# Systems Support for Preemptive Disk Scheduling

Zoran Dimitrijević, *Member*, *IEEE*, Raju Rangaswami, *Member*, *IEEE*, and Edward Y. Chang, *Senior Member*, *IEEE*

**Abstract**—Allowing higher-priority requests to preempt ongoing disk IOs is of particular benefit to delay-sensitive and real-time systems. In this paper, we present Semi-preemptible IO, which divides disk IO requests into small temporal units of disk commands to improve the preemptibility of disk access. We first lay out main design strategies to allow preemption of each component of a disk access—seek, rotation, and data transfer, namely, seek-splitting, JIT-seek, and chunking. We then present the preemption mechanisms for single and multidisk systems—JIT-preemption and JIT-migration. The evaluation of our prototype system showed that Semi-preemptible IO substantially improved the preemptibility of disk access with little loss in disk throughput and that preemptive disk scheduling could improve the response time for high-priority interactive requests.

**Index Terms**—Storage, preemptible disk access, preemptive disk scheduling, real-time, QoS, disk IO preemption.

## 1 Introduction

Traditionally, disk IOs have been thought of as nonpreemptible operations. Once initiated, they cannot be stopped until completed. Over the years, disk-storage designers have learned to live with this restriction. However, nonpreemptible IOs can be a stumbling block when designing applications requiring short, interactive responses. In this paper, our main goal is to investigate disk IO preemptibility. We propose a framework to make disk IOs semi-preemptible without changing the existing disks, thus providing system designers with a finer control over disk access. The mechanisms presented in this paper can also be used to enable IO preemption at the disk-firmware level. The firmware-based implementation would provide stronger real-time guarantees for higher-priority requests compared to our software-based prototype.

In addition to high-throughput, short response time is desirable and even required in certain application domains. One such domain is that of real-time disk scheduling. Real-time scheduling theoreticians have developed schedulability tests (the test of whether a task set is schedulable such that all deadlines are met) in various settings [1], [2], [3]. In real-time scheduling theory, *blocking*, or priority inversion, is defined as the time spent when a higher-priority task is prevented from running due to the nonpreemptibility of a low-priority task (in this paper, we refer to blocking as the *waiting time*). Blocking is undesirable since it degrades the schedulability of real-time tasks. Making disk IOs preemptible would reduce blocking and improve the schedulability of real-time disk IOs.

Real-world applications of preemptive IO scheduling include guaranteeing response time for interactive virtual reality, audio and video streaming, Web services running on shared storage, and complex multilevel database systems. For instance, storage systems often schedule background operations [4] like logging, defragmentation, indexing, or low-priority data mining. To handle cache misses due to foreground tasks, it is critical to preempt long-running noninteractive IOs to promptly service interactive IO requests. The latency tolerance for interactive operations in large Web databases is around 100 to 200 milliseconds (reponses to the user queries for all major Internet search engines are of this order). As another example, in an immersive virtual world, the latency tolerance between a head movement and the rendering of the next scene (which may involve a disk IO to retrieve relevant data) is around 15 milliseconds [5]. IO preemption enables delivery of the response time guarantees required by these applications.

In summary, the contributions of this paper are as follows:

- *Semi-preemptible IO.* We propose methods to abstract both read and write IO requests and make them preemptible [6]. This enables storage systems to reduce the response time for higher-priority requests.
- *Preemption mechanisms.* We explain two methods to preempt disk IOs in single and multidisk systems—JIT-preemption and JIT-migration. The preemption methods for RAID rely on our preemptible RAID architecture [7].
- *Prototype implementation.* We present a feasible path to implement Semi-preemptible IO. The implementation was possible through the use of a low-level disk profiler [8].

- *Z. Dimitrijević is with Google, Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043. E-mail: zorand@gmail.com.*
- *R. Rangaswami is with the School of Computing and Information Science, Florida International University, 11200 S.W. 8th St., Miami, FL 33199. E-mail: raju@cs.fiu.edu.*
- *E.Y. Chang is with the Department of Electrical and Computer Engineering, Eng I Building, University of California, Santa Barbara, CA 93106. E-mail: echang@ece.ucsb.edu.*

*Manuscript received 20 June 2004; revised 9 Feb. 2005; accepted 28 Apr. 2005; published online 16 Aug. 2005.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0208-0604.*

The rest of this paper is organized as follows: Section 2 introduces Semi-preemptible IO and describes its three core components. Section 3 introduces JIT-preemption and JIT-migration methods that facilitate effective disk IO preemption. In Section 4, we evaluate Semi-preemptible IO and our preemption mechanisms. Section 5 presents related research. In Section 6, we make concluding remarks and suggest directions for future work.

## 2 Semi-Preemptible IO

Before introducing the concept of Semi-preemptible IO [6], we first define some terms which we use throughout the rest of this paper:

- A *logical disk block* is the smallest unit of data that can be accessed on a disk drive (typically, 512 B). Each logical block resides at a physical disk location depicted by a physical address 3-tuple (cylinder, track, sector).
- A *disk command* is a nonpreemptible request issued to the disk over the IO bus (for example, the read, write, seek, and interrogative commands).
- A *disk IO request* is a request for read or write access to a sequential set of logical disk blocks (in this paper, we use the terms "disk IO request," "disk IO," and "IO" interchangeably).
- The *waiting time* is the time between the arrival of an IO request and the moment the disk starts servicing it.
- The *service time* is the sum of seek time, rotational delay, and data-transfer time for an IO request.
- The *response time* is then the sum of the waiting time and the service time.

In order to understand the magnitude of the waiting time, let us consider a typical read IO request, depicted in Fig. 1. The disk first performs a seek to the destination cylinder requiring $T_{seek}$ time. Then, the disk must wait for a rotational delay, denoted by $T_{rot}$, so that the target disk block comes under the disk arm. The final stage is the data transfer stage, requiring $T_{transfer}$ time, when the data is read from the disk media to the disk buffer. This data is simultaneously transferred over the IO bus to the system memory. (The IO bus data-transfer rate is typically greater than the internal disk transfer rate.)

**Overview.** For a typical commodity system, once a disk command is issued on the IO bus, it cannot be stopped. Traditionally, a disk IO is serviced using a single disk command. Consequently, the operating system must wait until the ongoing IO is completed before it can service the next IO request on the same disk. The system can issue multiple queued disk commands, but the disk firmware will only try to schedule them in a more efficient order. Once the disk starts the sequential data-transfer required to complete a disk command, it usually does not preempt the operation in order to favor another disk command in the command queue.

*Semi-preemptible IO* maps each IO request into multiple fast-executing disk commands using three methods. Each method addresses the reduction of one of the three components of the waiting time—the ongoing IO's rotational delay ($T_{rot}$), seek time ($T_{seek}$), and transfer time ($T_{transfer}$). In this paper, we present a software approach where we do not change the disk firmware to enable preemptions. It is also possible to provide a higher level of preemptibility and stronger real-time guarantees by changing the disk firmware. We briefly discuss the differences between software and firmware-based approaches in Sections 2.1, 2.2, and 2.3.

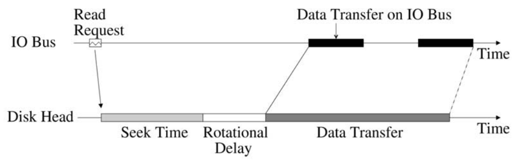


Fig. 1. Timing diagram for a disk read request.

- *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 (Section 2.1).
- *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 preseek 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 (Section 2.2).
- *Splitting $T_{seek}$.* Semi-preemptible IO can split a long seek into subseeks, and allows a preemption between two subseeks (Section 2.3).

In order to implement these methods, Semi-preemptible IO relies on accurate disk profiling, presented in Section 2.4.

**Waiting Time.** We use *waiting time* to quantify the preemptibility of disk access, a lower value indicating a more preemptible system. Let us assume that a higher priority request may arrive at any time during the execution of an ongoing IO request with equal probability. The waiting time for the higher priority request varies between zero and the duration of the on-going IO. The expected waiting time of a higher priority IO request can then be expressed in terms of seek time, rotational delay, and data transfer time required for on-going IO request as

$$E(T_{waiting}) = \frac{1}{2}(T_{seek} + T_{rot} + T_{transfer}). \quad (1)$$

Let $(V_1..V_n)$ be a sequence of $n$ fine-grained disk commands we use to service an IO request. Let the time required to execute disk-command $V_i$ be $T_i$. Let $T_{idle}$ be the total time during the servicing of the IO request when the disk was idle (i.e., no disk command is issued before JIT-seek). Using the above assumption that the higher priority request can arrive at any time with equal probability, the probability that it will arrive during the execution of the $i$th command $V_i$ can be expressed as

$$p_i = \frac{T_i}{\sum_{i=1}^{i=n} T_i + T_{idle}}.$$

Consequently, the expected waiting time of a higher priority request in Semi-preemptible IO can be expressed as

$$E(T'_{waiting}) = \frac{1}{2}\sum_{i=1}^{i=n}(p_i T_i) = \frac{1}{2}\frac{\sum_{i=1}^{i=n} T_i^2}{\left(\sum_{i=1}^{i=n} T_i + T_{idle}\right)}. \qquad (2)$$

The following example illustrates how Semi-preemptible IO can reduce the waiting time for higher-priority IOs (and, hence, improve the preemptibility of disk access).

**Illustrative Example.** Suppose a 500 kB read-request[1] has to seek 20,000 cylinders requiring $T_{seek}$ of 14 ms, must wait for a $T_{rot}$ of 7 ms, and requires $T_{transfer}$ of 25 ms at a transfer rate of 20 MBps. The expected waiting time, $E(T_{waiting})$, for a higher-priority request arriving during the execution of this request, is 23 ms, while the maximum waiting time is 46 ms. Semi-preemptible IO can reduce the waiting time by performing the following operations: It first predicts both the seek time and rotational delay using methods presented in Section 2.4. Since the predicted seek time is long ($T_{seek} = 14$ ms), it decides to split the seek operation into two subseeks, each of 10,000 cylinders, requiring $T'_{seek} = 9$ ms each. This seek splitting does not cause extra overhead in this case because the $T_{rot} = 7$ can mask the 4 ms increased total seek time ($2 \times T'_{seek} - T_{seek} = 2 \times 9 - 14 = 4$). The rotational delay is now $T'_{rot} = T_{rot} - (2 \times T'_{seek} - T_{seek} = 3$ ms.

With this knowledge, the disk driver waits for 3 ms before performing a JIT-seek. This JIT-seek method makes $T'_{rot}$ preemptible since no disk operation is being performed. The disk then performs the two subseek disk commands and then 25 successive read commands, each of size 20 kB, requiring 1 ms each. A higher priority IO request could be serviced immediately after each disk-command. Semi-preemptible IO thus enables preemption of an originally nonpreemptible read IO request. Now, during the service of this IO, we have two scenarios:

- *No higher-priority IO arrives.* In this case, the disk does not incur additional overhead for transferring data due to disk prefetching (discussed in Section 2.1). (If $T_{rot}$ cannot mask seek-splitting, the system can choose not to perform seek-splitting.)
- *A higher-priority IO arrives.* In this case, the maximum waiting time for the higher-priority request is now 9 ms in case it arrives during one of the two seek disk commands. However, if the on-going request is at the stage of transferring data, the longest stall for the higher priority request is just 1 ms. The expected value for waiting time is only $\frac{1}{2}\ \frac{2\times 9^2+25\times 1^2}{2\times 9+25\times 1+3} = 2.03$ ms, a significant reduction from 23 ms.

This example shows that Semi-preemptible IO substantially reduces the expected waiting time and, hence, increases the preemptibility of disk access. However, if an IO request is preempted to service a higher priority request, an extra seek operation may be required to resume service for the preempted IO. The distinction between *IO preemptibility* and *IO preemption* is an important one. Preemptibility enables preemption and incurs little overhead itself. Preemption always incurs overhead, but it reduces the service time for higher-priority requests. Preemptibility provides the system with the choice of trading throughput for short response time when such a trade-off is desirable. We present one such preemptive scheduling approach in our related work [7].

## 2.1 Chunking: Preempting $T_{transfer}$

The data transfer component ($T_{transfer}$) in disk IOs can be large. For example, the current maximum disk IO size used by Linux and FreeBSD is 128 kB and it can be larger for some specialized video-on-demand systems.[2] Furthermore, operating systems typically do not preempt a sequence of disk IOs accessing a larger set of consecutive disk blocks [12]. To make the $T_{transfer}$ component preemptible, Semi-preemptible IO uses *chunking*.

**Definition 2.1.** Chunking *is a method for splitting the data transfer component of an IO request into multiple smaller* chunk *transfers. The chunk transfers are serviced using separate disk commands, issued sequentially.*

**Benefits.** Chunking reduces the transfer component of $T_{waiting}$. A higher-priority request can be serviced after a chunk transfer is completed instead of after the entire IO is completed. For example, suppose a 500 kB IO request requires a $T_{transfer}$ of 25 ms at a transfer rate of 20 MBps. Using a chunk size of 20 kB, the expected waiting time for a higher priority request is reduced from 12.5 ms to 0.5 ms.

**Overhead.** For small chunk sizes, the IO bus can become a performance bottleneck due to the overhead of issuing a large number of disk commands. As a result, the disk throughput degrades. In the disk firmware-based implementation, the chunk size can be as small as one sector since there is no IO bus activity.

### 2.1.1 The Method

To perform chunking, the system must decide on the chunk size. Based on the profiling, Semi-preemptible IO chooses the minimum chunk size for which the sequential disk throughput is optimal without excessive IO bus activity (Section 2.4). The chunking relies on the existence of a read cache and a write buffer on the disk.

We now present the chunking for read and write IO requests separately.

**The Read Case.** Disk drives are optimized for sequential access and they continue prefetching data into the disk cache even after a read operation is completed [13]. Chunking for a read IO requests is illustrated in Fig. 2. The x-axis shows time and the two horizontal time lines depict the activity on the IO bus and the disk head, respectively. Employing chunking, a large $T_{transfer}$ is divided into smaller chunk transfers issued in succession. The first read command issued on the IO bus is for the first chunk. Due to the prefetching mechanism, all chunk transfers following the first one are serviced from the disk cache rather than the disk media. Thus, the data transfers on the IO bus (the small dark bars shown on the IO bus line in the figure) and the data transfer into the disk cache (the

1. Examples of schedulers that issue nonpreemptible access of the order of 500 kB and more are schedulers presented in Schindler et al. [9] work on track-aligned extents, Google File System [10], and Xtream [11].

2. These values are likely to vary in the future. Semi-preemptible IO provides a technique that does not deter disk preemptibility with the increased IO sizes.

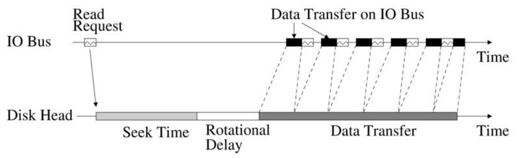


Fig. 2. Preemptibility of the data transfer.

dark shaded bar on the disk-head line in the figure) occur concurrently. The disk head continuously transfers data after the first read command, thereby fully utilizing the internal disk throughput.

Fig. 3 illustrates the effect of the chunk size on the disk throughput using a mock disk. The optimal chunk size lies between $a$ and $b$. A smaller chunk size reduces the waiting time for a higher-priority request. Hence, Semi-preemptible IO uses a chunk size close to but larger than $a$. For chunk sizes smaller than $a$, due to the overhead associated with issuing a disk command, the IO bus is a bottleneck. Point $b$ in Fig. 3 denotes the point beyond which the performance of the cache may be suboptimal due to the implementation of disk prefetching algorithms.

**The Write Case.** Semi-preemptible IO performs chunking for write IOs similarly to chunking for read requests. However, the implications of chunking in the write case are different. When a write IO is performed, the disk command can complete as soon as all the data is transferred to the disk write buffer. As soon as the write command is completed, the operating system can issue a disk command to service a higher-priority IO. However, the disk may choose to schedule a write-back operation for disk write buffers before servicing a new disk command. We refer to this delay as the *external waiting time*. Since the disk can buffer multiple write requests, the write-back operation can include multiple disk seeks. Consequently, the waiting time for a higher priority request can be substantially increased when the disk services write IOs.

In order to increase the preemptibility of write requests, we must take into account this external waiting time. External waiting can be reduced to zero by disabling write buffering. However, in the absence of write buffering, chunking would severely degrade disk performance. The disk would suffer from an overhead of one disk rotation after performing an IO for each chunk. To remedy external waiting, our prototype forces the disk to write only the last chunk of the write IO to disk media by setting the force-unit-access flag in the SCSI write command. Using this simple technique, the write-back operation to the disk medium is triggered at the end of each write IO. Consequently, the external waiting time is reduced since the write-back operation does not include multiple disk seeks (we present more details in our related work [6]).

### 2.2 JIT-Seek: Preempting $T_{rot}$

After the reduction of the $T_{transfer}$ component of the waiting time, the rotational delay and seek time components become significant even for large IOs. The rotational period ($T_P$) can be as much as 10 ms in current-day disk drives. To reduce the rotational delay component ($T_{rot}$) of the waiting

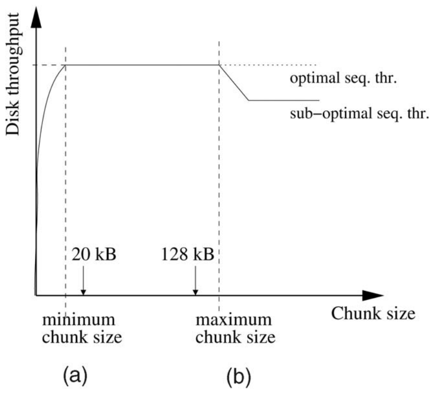


Fig. 3. Effect of chunk size on disk sequential throughput.

time, we propose a *just-in-time seek* (*JIT-seek*) technique for IO operations.

**Definition 2.2.** *The* JIT-seek *technique delays the servicing of the next IO request in such a way that the rotational delay to be incurred is minimized. We refer to the delay between two IO requests, due to JIT-seek, as* slack time.

**Benefits.**

1. The slack time between two IO requests is fully preemptible. For example, suppose that an IO request must incur a $T_{rot}$ of 5 ms and JIT-seek delays the issuing of the disk command by 4 ms. The disk is thus idle for $T_{idle} = 4$ ms. Then, the expected waiting time is reduced from 2.5 ms to $\frac{1}{2}\frac{1\times1}{1+4} = 0.1$ ms.
2. The slack obtained due to JIT-seek can also be used to perform data prefetching for the previous IO or to service a background request [14] and, hence, potentially increase the disk throughput.

**Overhead.** Semi-preemptible IO predicts the rotational delay and seek time between two IO operations in order to perform JIT-seek. If there is an error in prediction, then the penalty for JIT-seek is at most one extra disk rotation and some wasted cache space for unused prefetched data. In the firmware-based implementation, disks can predict JIT-seek more accurately because they control internal disk activity and have knowledge about the current head position (the similar access-time prediction is already implemented in certain disk firmwares in order to reduce noise [15]).

#### 2.2.1 The Method

The JIT-seek method is illustrated in Fig. 4. The x-axis depicts time and the two horizontal lines depict a regular IO and an IO with JIT-seek, respectively. With JIT-seek, the read command for an IO operation is delayed and issued just-in-time so that the seek operation takes the disk head directly to the destination block, without incurring any rotational delay at the destination track. Hence, data transfer immediately follows the seek operation. The available rotational slack, before issuing the JIT-seek command, is now preemptible. We can make two key observations about the JIT-seek method. First, an accurate JIT-seek operation reduces the $T_{rot}$ component of the waiting time without any loss in performance. Second,

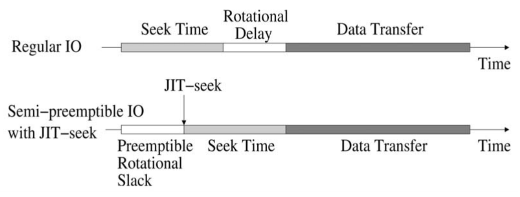


Fig. 4. JIT-seek.

and perhaps more significantly, the ongoing IO request can be serviced as much as possible, or even completely, if sufficient slack is available before the JIT-seek operation for a higher priority request.

The preseek slack made available due to the JIT-seek operation can be used in three possible ways:

- The preseek slack can be simply left unused. In this case, a higher-priority request arriving during the slack time can be serviced immediately.
- The slack can be used to perform additional data transfers. Operating systems can perform data prefetching for the current IO beyond the necessary data transfer. We refer to it as *free prefetching* [14]. Chunking is used for the prefetched data to reduce the waiting time of a higher priority request. Free prefetching can increase the disk throughput. We must point out, however, that free prefetching is useful only for sequential data streams where the prefetched data will be consumed within a short time. Operating systems can also perform another background request, as proposed elsewhere [14], [16].
- The slack can be used to mask the overhead incurred in performing *seek-splitting*, which we discuss next.

## 2.3 Seek Splitting: Preempting $T_{seek}$

The seek delay ($T_{seek}$) becomes the dominant component when the $T_{transfer}$ and $T_{rot}$ components are reduced. A full stroke of the disk arm may require as much as 15 to 20 ms in current-day disk drives. It may then be necessary to reduce the $T_{seek}$ component to further reduce the waiting time.

**Definition 2.3.** Seek-splitting *breaks a long, nonpreemptible seek of the disk arm into multiple smaller subseeks.*

**Benefits.** The *seek-splitting* method reduces the $T_{seek}$ component of the waiting time. A long nonpreemptible seek can be transformed into multiple shorter subseeks. A higher priority request can now be serviced at the end of a subseek, instead of being delayed until the entire seek operation is completed. For example, suppose an IO request involves a seek of 20,000 cylinders, requiring a $T_{seek}$ of 14 ms. Using seek-splitting, this seek operation can be divided into two 9 ms subseeks of 10,000 cylinders each. Then, the expected waiting time for a higher priority request is reduced from 7 ms to 4.5 ms.

**Overhead.**

1. Due to the mechanics of the disk arm, the total time required to perform multiple subseeks is greater than that of a single seek for a given seek distance. As a consequence, the seek-splitting method can degrade disk throughput. On the other hand, if we perform seek-splitting only when the available rotational slack (from JIT-seek) masks seek-splitting, then we avoid throughput degradation but incur slight degradation in expected waiting time (since the preseek slack is fully preemptible but the subseeks are not). The main benefit of seek-splitting is reducing the maximum value for waiting time. In case we implement Semi-preemptible IO in the disk firmware, the on-going long seek can be preempted at any time and the seek-splitting method is not necessary (instead, we can perform the actual seek preemption).
2. Splitting the seek into multiple subseeks increases the number of disk head accelerations and decelerations, consequently increasing the power usage and noise.

### 2.3.1 The Method

To split seek operations, Semi-preemptible IO uses a tunable parameter, the maximum subseek distance. The *maximum subseek distance* decides whether to split a seek operation. For seek distances smaller than the maximum subseek distance, seek-splitting is not employed. A smaller value for the maximum subseek distance provides higher responsiveness at the cost of possible throughput degradation.

Unlike the previous two methods, seek-splitting may degrade disk performance. However, we note that the overhead due to seek-splitting can, in some cases, be masked. If the preseek slack obtained due to JIT-seek is greater than the seek overhead, then the slack can be used to mask this overhead. A specific example of this phenomenon was presented in Illustrative Example. If the slack is insufficient to mask the overhead, seek-splitting can be aborted to avoid throughput degradation. Making such a trade-off, of course, depends on the requirements of the application.

## 2.4 Disk Profiling

As mentioned in the beginning of this section, Semi-preemptible IO greatly relies on disk profiling to obtain accurate disk parameters. The extraction of these disk parameters is described in the Diskbench technical report [8]. Semi-preemptible IO requires the following disk parameters:

- *The optimal chunk size.* In order to efficiently perform chunking, Semi-preemptible IO chooses the chunk size from the optimal range extracted using the disk profiler.
- *Seek time prediction.* In order to perform JIT-seek and seek-splitting, Semi-preemptible IO relies on the profiler-extracted seek curve.
- *Rotational delay prediction.* In order to implement JIT-seek, Semi-preemptible IO uses disk block mappings, rotational factors, and seek curve to accurately predict the rotational delay between two disk accesses [8].

Fig. 5 depicts the effect of chunk size on the read throughput performance for the SCSI disk used in our prototype system. The optimal range for chunk size (between the points $a$ and $b$ illustrated previously in Fig. 3) can be automatically extracted from these figures (in this case, between 8 and 250 kB). The disk profiler implementation was

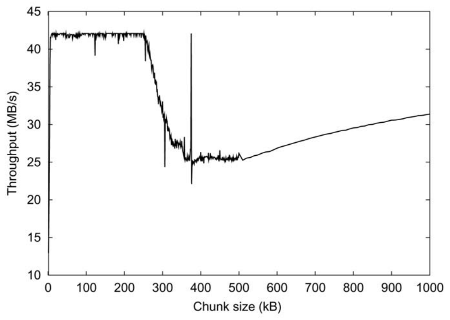


Fig. 5. Sequential read throughput versus chunk size (Seagate ST318437LW).

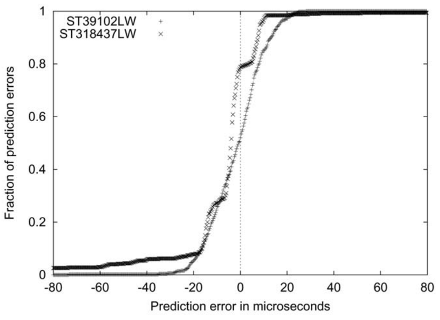


Fig. 6. Rotational delay prediction accuracy for ST39102LW and ST318437LW.

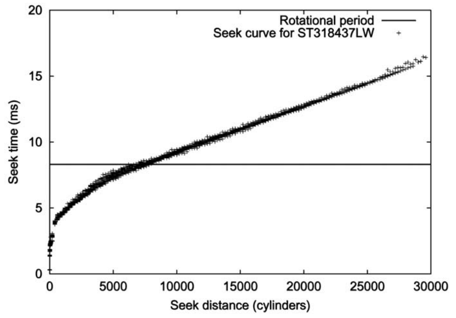


Fig. 7. Seek curve for ST318437LW.

successful in extracting the optimal chunk size for several SCSI and IDE disk drives [8].

In Fig. 6, we present the error distribution of rotational delay prediction for a large number of random request-pairs for two SCSI disks (ST318437LW is used in our prototype). Our prediction is accurate within 80 $\mu$s for 99 percent of the requests. In order to predict disk access time, our prototype must also predict the seek time. Fig. 7 depicts the seek curve for ST318437LW. The accuracy of the access-time prediction is bounded by the prediction accuracy of seek time, which is an error-bound of one millisecond.

Commodity disks support issuing multiple disk requests in an asynchronous fashion that can be rescheduled internally (known as disk queuing). Disks can also issue additional internal commands to handle bad-blocks or issue write-back operations. In order to control this external (for the Semi-preemptible IO) waiting time, we restrict disk queues to one request in our prototype. Since Semi-preemptible IO relies on the detailed disk profile, it can perform near-optimal scheduling itself. (To increase the reliability of JIT-seek, one possible improvement is queuing the last chunk with the next disk operation.)

## 3 Preemption Mechanisms

In this section, we introduce methods for IO preemption and resumption for single-disk and multiple-disk (RAID) systems.[3] We propose three mechanisms for IO preemption: 1) JIT-preemption with IO resumption at the same disk, 2) JIT-preemption with migration of the on-going IO to a different disk (favoring the newly arrived IO), and 3) preemption with JIT-migration of the on-going IO (favoring the on-going IO).

### 3.1 JIT-Preemption

When the local-disk scheduler (controlling the access to a single disk in a RAID system) decides that preempting and delaying an on-going IO would yield a better overall schedule, the IO should be preempted using *JIT-preemption*. This is a local decision, meaning that a request for the remaining portion of the preempted IO is placed back in the local-disk queue and resumed later on the same disk (or dropped completely[4]).

**Definition 3.1.** JIT-preemption *is the method for preempting an on-going Semi-preemptible IO at the points that minimize the rotational delay at the destination track (for a higher priority IO). The scheduler decides when to preempt the on-going IO using the knowledge about the available JIT-preemption points $P_i$. These points are roughly one disk rotation apart.*

**Preemption:** The method relies on JIT-seek (described in Section 2.2), which requires rotational delay prediction (also required by other disk schedulers [14], [17]). JIT-preemption is similar to free-prefetching introduced in [14]. However, if the preempted IO requires later completion, then the JIT-preemption yields useful data transfer (prefetching may or may not be useful). Another difference is that JIT-preemption 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 [6].

Fig. 8 depicts the positions of possible JIT-preemption points $P_i$. If $IO_1$ is preempted anywhere between two adjacent such points, the resulting service time for $IO_2$ would be exactly the same as if the preemption is delayed until the nextJIT-preemption point. This is because only the rotational delay at the destination track varies, depending on when the seek operation starts. The rotational delay is expected to be minimum at the JIT-preemption points, which are roughly one disk rotation apart.

Fig. 9 depicts the case when the ongoing $IO_1$ is preempted during its data transfer phase in order to service

3. In this paper, we use the term RAID for any multidisk system implementing reliable storage as a redundant array of commodity, inexpensive disks. We do not rely on hardware-only or static-placement RAID implementations.

4. For example, the scheduler may drop unsuccessful speculative reads, cache-prefetch operations, or preempted IOs whose deadlines have expired.

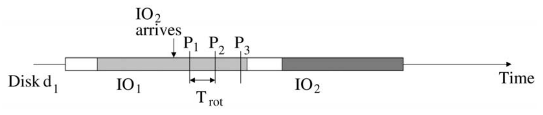


Fig. 8. Possible JIT-preemption points.

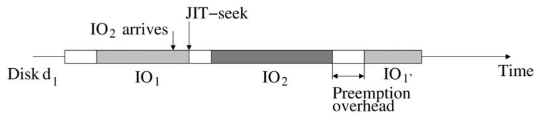


Fig. 9. JIT-preemption during data transfer.

$IO_2$. 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 $IO_2$, its access-time overhead is reduced compared to the no-preemption case depicted in Fig. 8. The preemption overhead when $IO_1$ is rescheduled immediately is likely to be longer than the JIT-seek duration because it also involves a rotational delay.

**Resumption.** The preempted IO is resumed later at the same disk. The preemption overhead (depicted in Fig. 9) 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).

## 3.2 JIT-Preemption with Migration

In terms of preemption, the main difference between single-disk and RAID systems is the RAID's ability to service multiple higher priority IOs at the same time (one at each disk). Another important difference is the ability to change the destination disk (migrate) or to postpone preempted write IOs without sacrificing reliability.

RAID systems duplicate data for deliberate redundancy. If an ongoing IO can also be serviced from some other disk which holds a copy of the data, then the scheduler has the option to preempt the IO and migrate its remaining portion to the other disk. In the traditional static RAIDs, this situation can happen in RAID levels 1 (mirrored) and 0/1 (striped+mirrored) [18]. It might also happen in reconfigurable RAID systems (for example, HP AutoRAID [19]), in object-based RAID storage [20], or in nontraditional large-scale software RAIDs [10].

**Definition 3.2.** JIT-preemption-with-migration *is the method for preempting and migrating an ongoing IO to a different disk to minimize the service time for the newly arrived IO.*

**Preemption.** For preemption, this method relies on the previously described JIT-preemption. Fig. 10 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 access time required for the completion of $IO_1$ at the replica disk.

**Resumption.** The preempted IO is resumed later at the disk to which it was migrated. The preempted IO enters the scheduling queue of the mirror disk and is serviced according to the single-disk scheduling policy. The preemption overhead exists only at the mirror disk. This

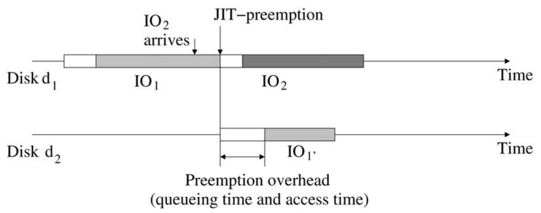


Fig. 10. JIT-preemption with migration.

suggests that this method may be able to improve the schedule when load balance is hard to achieve.

## 3.3 JIT-Migration

When a scheduler decides to migrate the preempted IO to another disk with a copy of the data, it can choose to favor the newly arrived IO or the ongoing IO. The former uses JIT-preemption introduced earlier. The latter uses *JIT-migration.*

**Definition 3.3.** JIT-migration *is the method for preempting and migrating an on-going IO in a fashion that minimizes the service time for the on-going IO. The on-going IO is preempted at the moment when the destination disk starts performing data-transfer for the remaining portion of the IO. The original IO is then preempted, but its completion time is not delayed.*

**Preemption.** JIT-migration also relies on JIT-seek and is used to preempt and migrate the on-going IO only if it does not increase its service time, thereby favoring the on-going IO.

Fig. 11 depicts the case when the on-going IO ($IO_1$) is more important than the newly arrived IO ($IO_2$). When the disk with $IO_1$ 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 to 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). JIT-migration is beneficial only when the mirror disk is not the same for all disk IOs and the RAID scheduler cannot schedule the low-priority $IO_2$ on the same disks as the high-priority $IO_1$ (for example, this is possible in object-based RAID storage [20] or in software RAIDs [10]).

**Resumption.** In the case of JIT-migration, $IO_1$ is not preempted until the disk with another replica is ready to

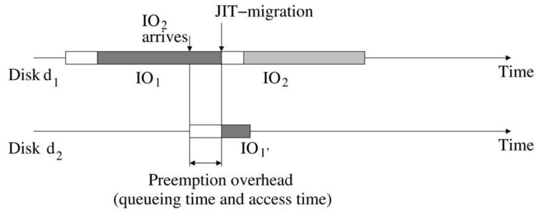


Fig. 11. Preemption with JIT-migration.

continue its data transfer. Again, the preemption overhead exists only at the mirror disk.

## 4 Experimental Evaluation

We now present the experimental evaluation of IO preemption methods based on the prototype implementation of Semi-preemptible IO and the simulator for preemptible RAID systems. Our experiments aimed to answer the following two questions:

- What is the level of *preemptibility* of Semi-preemptible IO and how does it influence the disk throughput?
- What is the effect of IO *preemption* on the average response time and the disk throughput for both single and multidisk systems?

### 4.1 Preemptibility

In order to answer the first question, we have implemented a prototype system [6] which can service IO requests using either the traditional nonpreemptible method (*nonpreemptible IO*) or Semi-preemptible IO.

#### 4.1.1 Experimental Setup

Our prototype runs as a user-level process in Linux and talks directly to a SCSI disk using the Linux SCSI-generic interface. The prototype uses the logical-to-physical block mapping of the disk, the seek curve, and the rotational skew times, all of which are automatically generated by the Diskbench profiler [8]. All experiments were performed on a Pentium III 800 MHz machine with a Seagate ST318437LW SCSI disk. This SCSI disk had two tracks per cylinder, with 437 to 750 blocks per track, depending on the disk zone. The total disk capacity was 18.4 GB. The rotational speed of the disk was 7,200 RPM. The maximum sequential disk throughput was between 24.3 and 41.7 MB/s.

For performance benchmarking, we performed two sets of experiments. First, we tested the preemptibility of the system using synthetic IO workload. For the synthetic workload, we used equal-sized IO requests within each experiment. The low-priority IOs were for data located at random positions on the disk. In the experiments where we actually performed preemption, the higher priority IO requests were also at random positions. However, their size was set to only one block in order to provide only the estimate for preemption overhead. We tested the preemptibility under the *first-come-first-serve (FCFS)* and *elevator* disk scheduling policies. In the second set of experiments, we used traces obtained using an instrumented Linux kernel disk-driver.

Nonpreemptible IOs were serviced using chunk sizes of at most 128 kB (smaller IOs were serviced as a single, smaller nonpreemptible IO). This is currently the maximum size used by Linux and FreeBSD for breaking up large IOs. We assumed that a large nonpreemptible IO cannot be preempted between chunks since this is the case for current nonpreemptive disk schedulers. Based on disk profiling, our prototype used the following parameters for Semi-preemptible IO: Chunking divided the data transfer into chunks of at most 50 disk blocks each (25 kB). JIT-seek used

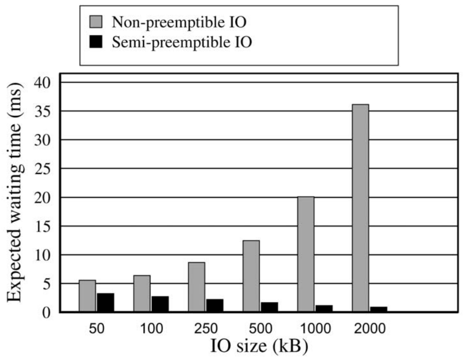


Fig. 12. Improvements in the expected waiting time (Elevator).

an offset of 1 ms to reduce the probability of prediction errors. Seeks for more than half of the disk size in cylinders were split into two equal-sized, smaller seeks. We used the SCSI *seek* command to perform subseeks.

The experiments for preemptibility of disk access measured the duration of (nonpreemptible) disk commands in the absence of higher-priority IO requests. The results include both detailed distribution of disk commands durations (and, hence, maximum possible waiting time) and the expected waiting time calculated using the measured durations of disk commands and (2), as explained in Section 2.

#### 4.1.2 Synthetic Workload

Fig. 12 depicts the difference in the expected waiting time between nonpreemptible IO and Semi-preemptible IO (calculated using the measured durations of nonpreemptible disk commands). In this experiment, IOs were serviced using elevator-based policy for data located at random positions on the disk. We can see that the expected waiting time for nonpreemptible IOs increases linearly with IO size due to increased data transfer time. The expected waiting time for Semi-preemptible IO actually decreases with IO size since the disk spends more time performing more-preemptible data transfer for larger IOs. For small IOs (less than 50 kB), the data-transfer component becomes even less significant and the preemptibility is similar to the 50 kB case.

Fig. 13 shows the effect of Semi-preemptible IO on the achieved disk throughput. The reduction of throughput for Semi-preemptible IO was due to the overhead of seek-splitting and misprediction of seek and rotational delay. Results for FCFS scheduling were published in [6].

Fig. 14 shows the individual contributions of the three strategies with respect to expected waiting time for the synthetic workload with the elevator scheduling policy. Fig. 15 summarizes the individual contributions of the three strategies with respect to the achieved disk throughput. Seek-splitting slightly degraded the disk throughput since, whenever a long seek was split, the disk required more time to perform multiple subseeks. Also, JIT-seek introduced overhead in the case of misprediction.

Since disk commands are nonpreemptible (even in Semi-preemptible IO), we can measure the duration of disk commands and then calculate the expected waiting time. A smaller value of waiting time implies a more preemptible system. Fig. 16 shows the distribution of the durations of

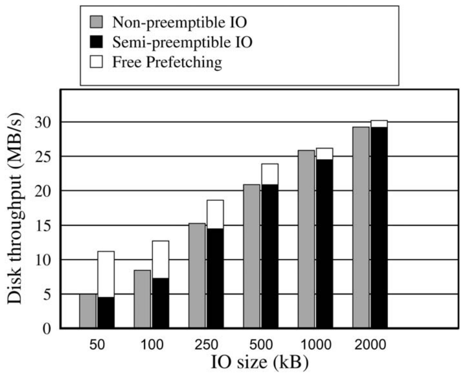


Fig. 13. Effect on achieved disk throughput (Elevator).

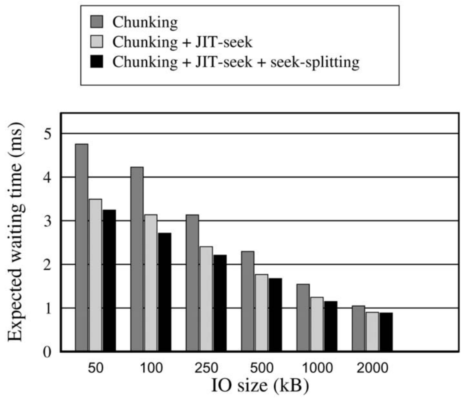


Fig. 14. Individual contributions of Semi-preemptible IO components on the expected waiting time (Elevator).

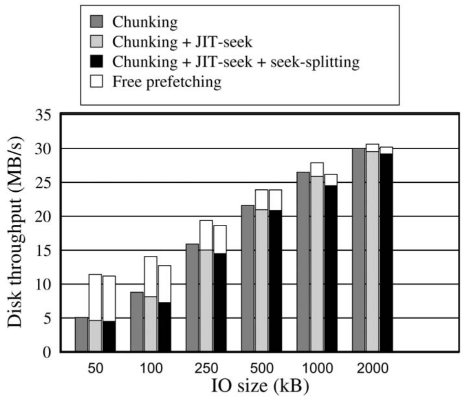


Fig. 15. Individual effects of Semi-preemptible IO strategies on disk throughput (Elevator).

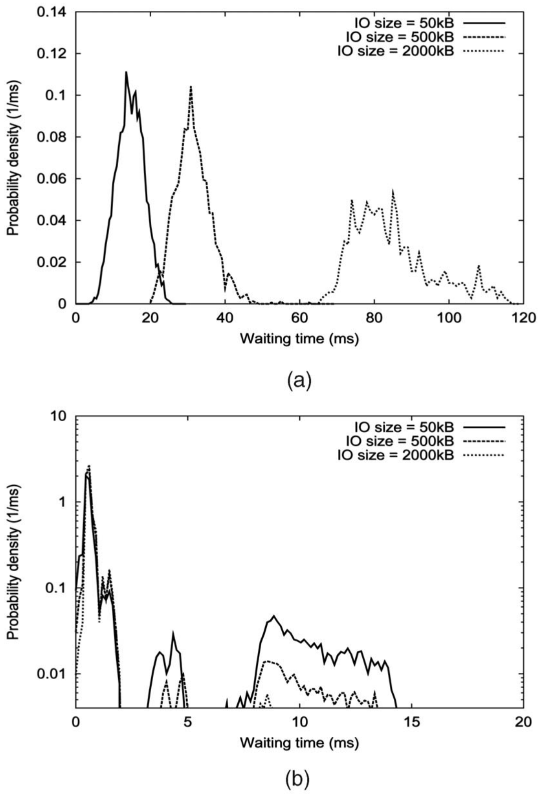


Fig. 16. Distribution of the disk command duration (FCFS). Smaller values of waiting time imply a higher preemptibility. (a) Nonpreemptible IO (linear scale). (b) Semi-preemptible IO (logarithmic scale).

disk commands for both nonpreemptible IO and Semi-preemptible IO (for exactly the same sequence of IO requests). In the case of nonpreemptible IO (Fig. 16a), the disk access can be preempted only when the current sequential IO is completed. The obtained distribution was dense near the sum of the average seek time, rotational delay, and transfer time required to service the entire IO request. The distribution was wider when the IO requests were larger because the duration of data transfer depended not only on the size of the IO request, but also on the throughput of the disk zone where the data resided.

In the case of Semi-preemptible IO, the distribution of the durations of disk commands did not directly depend on the IO request size, but on the sizes of individual disk commands used to service the IO request. (We plot the distribution for the Semi-preemptible IO case in logarithmic scale so that the probability density of longer disk commands can be better visualized.) In Fig. 16b, we see that, for Semi-preemptible IO, the largest probability density was around the time required to transfer a single chunk of data. If the chunk included the track or cylinder skew, the duration of the command was slightly longer. (The two peaks immediately to the right of the highest peak, at approximately 2 ms, have the same probability because the disk used in our experiments had two tracks per cylinder.) The part of the distribution between 3 ms and 16 ms in the figure is due to the combined effect of JIT-seek and seek-splitting on the seek and rotational delays. The probability for this range was small, approximately $0.168$, $0.056$, and $0.017$ for 50 kB, 500 kB, and 2 MB IO requests, respectively.

### 4.1.3 *Trace Workload*

IO traces were obtained from three applications. The first trace (DV15) was obtained when the Xtream multimedia system [11] was servicing 15 simultaneous video clients using the FCFS disk scheduler. The second trace (Elevator15) was obtained using a similar setup where Xtream used the native Linux elevator scheduler to handle concurrent disk IOs. The third was a disk trace of the TPC-C database benchmark with 20 warehouses obtained from [21]. Trace summary is presented in Table 1.

TABLE 1
Trace Summary

| Trace | Number of requests | Avg. req. size [*blocks*] | Max. disk block number |
|---|---|---|---|
| *DV15* | 10800 | 128.7 | 28442272 |
| *Elevator15* | 10180 | 127.6 | 28429968 |
| *TPC* | 1376482 | 126.5 | 8005312 |

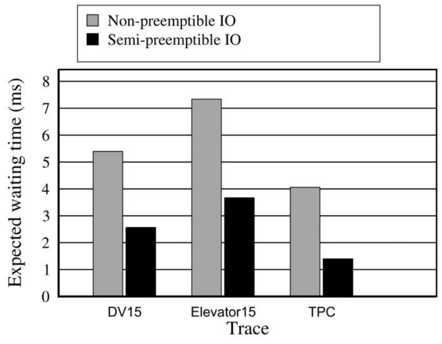


Fig. 17. Improvement in the expected waiting time (using disk traces).

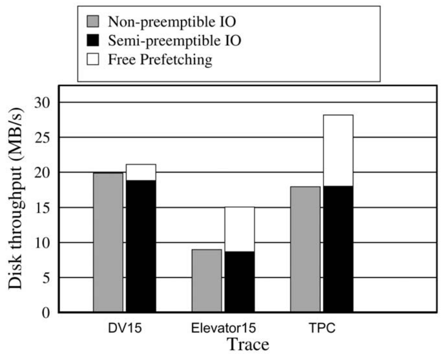


Fig. 18. Effect on the achieved disk throughput (using disk traces).

Figs. 17 and 18 show the expected waiting time and disk throughput for the trace experiments. In the case of Semi-preemptible IO, the expected waiting time was smaller by as much as 65 percent (Fig. 17) with less than 5.4 percent (Fig. 18, maximum 5.33 percent for DV15 traces) loss in disk throughput for all traces. Elevator15 had smaller throughput than DV15 because several processes were accessing the disk concurrently, which increased the total number of seeks.

## 4.2 Preemption

In order to answer the second question, we compared preemptive and nonpreemptive approaches using the preemptible RAID simulator PraidSim [7] as well as the Semi-preemptible IO prototype implementation explained previously.

### 4.2.1 *Experimental Setup*

We used PraidSim to evaluate preemptive RAID scheduling algorithms. PraidSim was implemented in C++ and used Disksim [22] to simulate disk accesses. We did not use the Disksim RAID support, but rather implemented our own simulator for QoS-aware RAID systems (based on the architecture presented in [7]). PraidSim can either generate a synthetic 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.

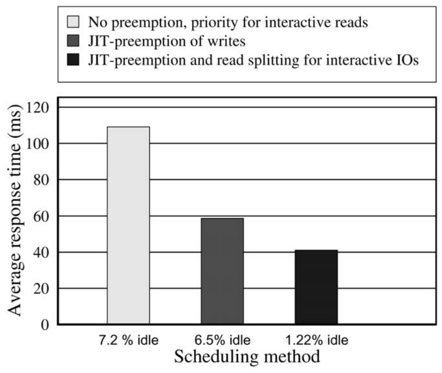


Fig. 19. Average interactive read response times for write-intensive applications.

To estimate the response time for higher priority IO requests and the preemption overhead, we conducted experiments wherein higher priority requests were inserted into the IO queue at arrival rate $\nu$ with normal distribution.

**Write-Intensive Real-Time 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 generated 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 was 10 req/s. The external non-interactive IOs were 2 MB each and interactive IOs were 1 MB each. The workload corresponded to a video surveillance system with 50 dvd-quality write video streams, 20 read streams, and 10 interactive operations performed each second.

Fig. 19 depicts the improvements in the response time for interactive IOs and the overhead in terms of reduced idle-time. The system was able to satisfy all real-time streaming requirements in this experiment. 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 percent to 6.5 percent. When the read IOs were split between mirror disks (read-splitting), we further decreased the response time (by reducing the data-transfer component on each disk) with overhead in terms of reduced disk idle time.

**Read-Intensive Applications.** Fig. 20 depicts the average response times for interactive read requests for read-intensive real-time streaming applications. The setup was the same as for write-intensive applications in the previous experiment, but the system serviced only read IOs. The streaming rate for noninteractive reads was 129 MB/s. The interactive IOs were 1 MB each and their arrival rate was 10 req/s. The improvements in average response times were similar to those in our write-intensive experiment. Although JIT-preemption with migration did not substantially improve the average response for interactive IOs, the better load-balance increased idle time. The read-splitting method split the IO into half and scheduled it to both replicas, which further decreased the response time, but

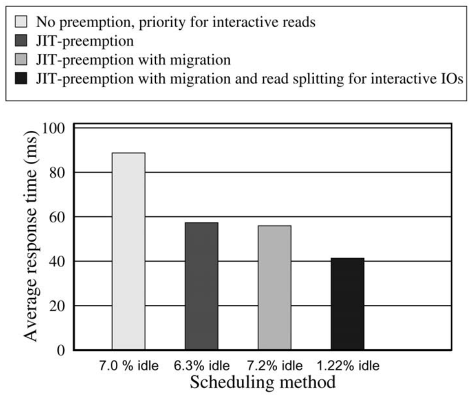


Fig. 20. Average interactive read response times for read-intensive applications.

substantially reduced the average disk idle time because of additional seeks.

### 4.2.2 Response Time for Cascading Interactive IOs

Interactive operations often require isuing multiple IOs for their completion. For example, a video-on-demand systems may first fetch metadata containing information about the position of requested frame in a file. Another example is a video surveillance system supporting complex interactive queries with data dependences.

In order to show how preemptions help when the interactive operation consists of issuing multiple IO requests in a cascade, we performed the following experiment. The background, noninteractive workload consisted of both read and write IOs, each external IO being 2 MB long. We used the RAID 0/1 configuration with eight disks. The sizes of internal IOs were between 0 and 2 MB and the interactive IOs were 100 kB each. As soon as one interactive IO completed, we issued the next IO in the cascade, measuring the time required to complete all cascading IOs. Fig. 21 depicts the effect of cascading interactive IOs on average response time for the entire operation. The preemptive approach could service six cascading IOs for interactive operations with 100 ms latency tolerance, whereas the nonpreemptive approach could service only two.

### 4.2.3 Preemption Overhead for Semi-Preemptible IO

In this section, we present results using the Semi-preemptible IO prototype with synthetic workload comprised of both lower priority and higher priority disk IOs. Table 2 presents the response time for a higher-priority request when using Semi-preemptible IO in two possible scenarios: 1) The higher-priority request was serviced after the ongoing IO was completed (nonpreemptible IO) and 2) the ongoing IO was preempted to service the higher priority IO request (Semi-preemptible IO). The results in Table 2 illustrate the case when the ongoing request was a read request.

Each time a higher priority request preempts a low priority IO request for disk access, an extra seek is required to resume servicing the preempted IO at a later time (implying a loss in disk throughput). Table 2 presents the average response time and the disk throughput for different arrival rates of higher-priority requests. For the same size of low priority IO requests, the average response time did not

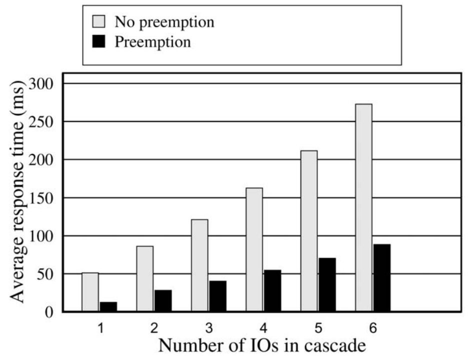


Fig. 21. Response time for cascading interactive IOs.

increase significantly with the increase in the arrival rate of higher priority requests. However, the disk throughput decreased with an increase in the arrival rate of higher priority requests, as expected.

Table 3 presents the average response time for higher priority requests depending on their arrival rate ($\nu$). The low-priority requests were 2 MB each. By preempting the ongoing Semi-preemptible IOs, the response time for a high priority request was reduced by a factor of four. The maximum response times for Semi-preemptible IO with and without preemption were measured as 34.6 ms and 150.1 ms, respectively.

### 4.2.4 External Waiting Time

In Section 2.1, we explained the difference in the preemptibility of read and write IO requests and introduced the notion of external waiting time. Table 4 summarizes the effect of external waiting time on the preemption of write IO requests. The arrival rate of higher-priority requests was set to $\nu = 1$ req/s.

As shown in Table 4, the average response time for higher priority requests in the presence of low priority write IOs is several times greater than for read experiments. Since the higher priority requests had the same arrival pattern in both experiments, the average seek time and rotational delay were the same for both read and write experiments. The large and often unpredictable external waiting time in the write case explains these results.

TABLE 2
The Average Response Time and Disk Throughput for Nonpreemptive IO (*npIO*) and Semi-Preemptive IO (*spIO*)

| *IO* | $\nu$ | *Avg. Resp.* $[ms]$ | | *Thr.* $[MB/s]$ | |
|---|---|---|---|---|---|
| $[kB]$ | $[req/s]$ | *npIO* | *spIO* | *npIO* | *spIO* |
| 50 | *0.5* | 19.2 | 19.4 | 3.39 | 2.83 |
| 50 | *1* | 21.8 | 16.0 | 3.36 | 2.89 |
| 50 | *2* | 20.8 | 17.6 | 3.32 | 2.82 |
| 50 | *5* | 21.0 | 18.2 | 3.18 | 2.62 |
| 50 | *10* | 21.2 | 18.3 | 2.95 | 2.30 |
| 50 | *20* | 21.1 | 18.4 | 2.49 | 1.68 |
| 500 | *0.5* | 29.2 | 15.7 | 16.25 | 16.40 |
| 500 | *1* | 28.1 | 15.5 | 16.15 | 16.20 |
| 500 | *2* | 28.2 | 16.7 | 15.94 | 15.77 |
| 500 | *5* | 28.6 | 16.0 | 15.28 | 14.58 |
| 500 | *10* | 28.9 | 16.3 | 14.24 | 12.48 |
| 500 | *20* | 29.4 | 16.8 | 11.96 | 8.57 |

TABLE 3
Response Time with and without Preemption for 2 MB IOs (Mean, Standard Deviation $\sigma$, and Maximum)

| $\nu$ | *npIO* | | | *spIO* | | |
|---|---|---|---|---|---|---|
| $[Hz]$ | mean | $\sigma$ | max | mean | $\sigma$ | max |
| 0.5 | 49.7 | 74.6 | 100.1 | 14.7 | 21.2 | 23.1 |
| 1 | 47.4 | 71.1 | 115.1 | 14.8 | 21.4 | 25.2 |
| 2 | 47.8 | 71.7 | 96.6 | 13.9 | 20.1 | 24.6 |
| 5 | 47.8 | 71.7 | 117.2 | 15.0 | 21.7 | 29.8 |
| 10 | 46.2 | 69.5 | 110.9 | 14.5 | 21.1 | 33.9 |
| 20 | 50.9 | 75.8 | 150.1 | 14.9 | 21.6 | 34.6 |

TABLE 4
The Expected Waiting Time and Average Response Time for Nonpreemptible and Semi-Preemptible IO ($\nu = 1$ Reqs)

| | *Exp. Waiting* $[ms]$ | | | | *Avg. Response* $[ms]$ | | | |
|---|---|---|---|---|---|---|---|---|
| *IO* | *npIO* | | *spIO* | | *npIO* | | *spIO* | |
| $[kB]$ | RD | WR | RD | WR | RD | WR | RD | WR |
| 50 | 8.2 | 11.4 | 3.9 | 9.5 | 21.8 | 105.8 | 16.0 | 24.6 |
| 250 | 11.8 | 12.9 | 3.1 | 5.6 | 25.5 | 27.2 | 16.1 | 21.2 |
| 500 | 16.4 | 18.7 | 2.5 | 4.7 | 28.1 | 36.0 | 15.5 | 20.3 |
| 1,000 | 25.9 | 33.3 | 1.9 | 3.7 | 36.8 | 45.7 | 14.4 | 19.5 |
| 2,000 | 45.4 | 60.9 | 1.4 | 2.9 | 58.3 | 70.0 | 14.7 | 18.3 |

## 5 Related Work

Before the pioneering work of [23], [24], [25], it was assumed that the nature of disk IOs was inherently nonpreemptible. Daigle and Strosnider [23] proposed breaking up a large IO into multiple smaller chunks to reduce the data transfer component ($T_{transfer}$) of the *waiting time* for higher priority requests. A minimum chunk size of one track was proposed. In this paper, we improve upon the conceptual model of [23] in three respects: 1) In addition to enabling preemption of the data transfer component, we show how to enable preemption of the $T_{rot}$ and $T_{seek}$ components, 2) we improve upon the bounds for zero-overhead preemptibility, and 3) we show that making write IOs preemptible is not as straightforward as it is for read IOs and propose one possible solution.

Semi-preemptible IO [6] uses a *just-in-time seek* (JIT-seek) technique to make the rotational delay preemptible. JIT-seek can also be used to mask the rotational delay with useful data prefetching. In order to implement both methods, our system relies on accurate disk profiling [8], [26], [27], [28], [29]. Rotational delay masking has been proposed in multiple forms. Worthington et al. [30] and Huang and Chiueh [31] present rotational-latency-sensitive schedulers that consider the rotational position of the disk arm to make better scheduling decisions. Ganger et al. [14], [16], [32] proposed *freeblock scheduling*, wherein the disk arm services background jobs during the rotational delay between foreground jobs. Seagate uses a variant of just-in-time seek [15] in some of its disk drives to reduce power consumption and noise. Semi-preemptible IO uses similar techniques to achieve a different goal—to make rotational delays preemptible.

There is a large body of literature proposing IO scheduling policies for multimedia and real-time systems that improve disk response time [33], [34], [35]. Semi-preemptible IO is orthogonal to these contributions. We believe that the existing methods can benefit from using preemptible IO to improve schedulability and further decrease response time for higher priority requests. For instance, to model real-time disk IOs, one can draw from real-time CPU scheduling theory. Molano et al. [24] adapt the *Earliest Deadline First* (EDF) algorithm from CPU scheduling to disk IO scheduling. Since EDF is a preemptive scheduling algorithm, a higher priority request must be able to preempt a lower priority request. However, an on-going disk request cannot be preempted instantaneously. Applying such classical real-time CPU scheduling theory is simplified if the preemption granularity is independent of system variables like IO sizes. Semi-preemptible IO provides such an ability. However, further investigation is necessary to address the nonlinear preemption overhead occurring in preemptible disk scheduling (we present two possible greedy preemptive scheduling approaches in our related work [7]).

Semi-preemptible IO suits best the systems issuing large sequential IO requests, of the order of 500 kB and more. For example, Schindler et al. [9] schedule accesses to track-aligned extents, which are of the order of 500 kB (one disk track for current disks). The Google File System [10] often accesses its 64 MB chunks in nonpreemptive IOs of the order of one megabyte. Many multimedia schedulers, including Xtream [11] and Bubbleup [33], use multimegabyte nonpreemptive IOs for storing and retrieving video data.

## 6 Conclusion and Future Work

In this paper, we have presented the design of Semi-preemptible IO and proposed three techniques for reducing IO waiting-time—chunking of data transfer, just-in-time seek, and seek-splitting. These techniques enable the preemption of a disk IO request and thus substantially reduce the waiting time for a competing higher priority disk IO. Our empirical study showed that Semi-preemptible IO reduced the waiting time for both read and write requests significantly when compared with nonpreemptible IOs. Using both synthetic and trace workloads, we have shown that these techniques can be efficiently implemented, given detailed disk parameters [6], [7], [8]. We have also presented two methods for deciding how to preempt ongoing IOs in multidisk systems—JIT-preemption and JIT-migration. These two methods ensure that disk IO preemptions happen in a rotationally efficient manner. The Semi-preemptible IO mostly benefits systems servicing noninteractive IOs of the order of 500 kB and larger, with a moderate number of higher priority, interactive requests. The examples of these systems are delay-sensitive multimedia and real-time systems. We plan to further study the impact of preemptibility on traditional and real-time disk-scheduling algorithms.

## Acknowledgments

This work was partially supported by US National Science Foundation grants IIS-0133802 (Career) and IIS-0219885 (ITR), and a University of California DiMI grant. The authors wish to thank the anonymous reviewers and Erik Riedel from Seagate Research for their valuable comments and suggestions. This research was conducted while all of the authors were with the University of California, Santa Barbara.

## REFERENCES

[1] K. Jeffay, D.F. Stanat, and C.U. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proc. 12th IEEE Real-Time Systems Symp.,* Dec. 1991.

[2] D.I. Katcher, H. Arakawa, and J.K. Strosnider, "Engineering and Analysis of Fixed Priority Schedulers," *Software Eng.,* vol. 19, no. 9, 1993.

[3] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *ACM J.,* Jan. 1973.

[4] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C.R. Lumb, and G.R. Ganger, "A Framework for Building Unobtrusive Disk Maintenance Applications," *Proc. Third Usenix Conf. File and Storage Technologies (FAST),* Mar. 2004.

[5] R.T. Azuma, "Tracking Requirements for Augmented Reality," *Comm. ACM,* vol. 36, no. 7, July 1993.

[6] Z. Dimitrijevic, R. Rangaswami, and E. Chang, "Design and Implementation of Semi-Preemptible IO," *Proc. Second Usenix Conf. File and Storage Technologies (FAST),* Mar. 2003.

[7] Z. Dimitrijevic, R. Rangaswami, and E. Chang, "Preemptive Raid Scheduling," Technical Report TR-2004-19, Univ. of California, Santa Barbara, Apr. 2004.

[8] Z. Dimitrijevic, R. Rangaswami, D. Watson, and A. Acharya, "Diskbench: User-Level Disk Feature Extraction Tool," Technical Report TR-2004-18, Univ. of California, Santa Barbara, Apr. 2004.

[9] J. Schindler, J.L. Griffin, C.R. Lumb, and G.R. Ganger, "Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics," *Proc. First Usenix Conf. File and Storage Technologies (FAST),* Jan. 2002.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *ACM Symp. Operating System Principles (SOSP),* Oct. 2003.

[11] Z. Dimitrijevic, R. Rangaswami, and E. Chang, "The XTREAM Multimedia System," *Proc. IEEE Conf. Multimedia and Expo,* Aug. 2002.

[12] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," *Proc. 18th Symp. Operating Systems Principles,* Sept. 2001.

[13] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer,* 1994.

[14] C.R. Lumb, J. Schindler, G.R. Ganger, and D.F. Nagle, "Towards Higher Disk Head Utilization: Extracting Free Bandwith from Busy Disk Drives," *Proc. Usenix Symp. Operating Systems Design and Implementation (OSDI),* Oct. 2000.

[15] Seagate Technology, "Seagate's Sound Barrier Technology," http://www.seagate.com/docs/pdf/whitepaper/sound_barrier.pdf, Nov. 2000.

[16] E. Riedel, C. Faloutsos, G.R. Ganger, and D.F. Nagle, "Data Mining on an OLTP System (Nearly) for Free," *Proc. ACM SIGMOD,* May 2000.

[17] D.M. Jacobson and J. Wilkes, "Disk Scheduling Algorithms Based on Rotational Position," Technical Report HPL-CSP-91-7rev1, Hewlett-Packard Laboratories, Mar. 1991.

[18] 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,* vol. 26, no. 2, June 1994.

[19] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID Hierarchical Storage System," *ACM Trans. Computer Systems,* vol. 14, no. 1, Feb. 1996.

[20] M. Mesnier, G.R. Ganger, and E. Riedel, "Object-Based Storage," *IEEE Comm. Magazine,* Aug. 2003.

[21] Performance Evaluation Laboratory, Brigham Young Univ., "Trace Distribution Center," http://tds.cs.byu.edu/tds/, 2002.

[22] G.R. Ganger, B.L. Worthington, and Y.N. Patt, "The DiskSim Simulation Environment Version 2. 0 Reference Manual," *Reference Manual,* Dec. 1999.

[23] S.J. Daigle and J.K. Strosnider, "Disk Scheduling for Multimedia Data Streams," *Proc. IS&T/SPIE,* Feb. 1994.

[24] A. Molano, K. Juvva, and R. Rajkumar, "Guaranteeing Timing Constraints for Disk Accesses in RT-Mach," *Proc. IEEE Real Time Systems Symp.,* Dec. 1997.

[25] A. Thomasian, "Priority Queueing in RAID5 Disk Arrays with an NVS Cache," *Proc. IEEE Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS),* Jan. 1995.

[26] M. Aboutabl, A. Agrawala, and J.-D. Decotignie, "Temporally Determinate Disk Access: An Experimental Approach," Technical Report CS-TR-3752, Univ. of Maryland, 1997.

[27] J. Schindler and G.R. Ganger, "Automated Disk Drive Characterization," Technical Report CMU-CS-00-176, Carnegie Mellon Univ., Dec. 1999.

[28] N. Talagala, R.H. Arpaci-Dusseau, and D. Patterson, "Microbenchmark-Based Extraction of Local and Global Disk Characteristics," technical report, Univ. of California, Berkeley, 1999.

[29] B.L. Worthington, G. Ganger, Y.N. Patt, and J. Wilkes, "Online Extraction of SCSI Disk Drive Parameters," *Proc. ACM Sigmetrics,* May 1995.

[30] B.L. Worthington, G.R. Ganger, and Y.N. Patt, "Scheduling Algorithms for Modern Disk Drives," *Proc. ACM Sigmetrics,* May 1994.

[31] L. Huang and T. Chiueh, "Implementation of a Rotation-Latency-Sensitive Disk Scheduler," technical report, State Univ. of New York at Stony Brook, May 2000.

[32] C.R. Lumb, J. Schindler, and G.R. Ganger, "Freeblock Scheduling Outside of Disk Firmware," *Proc. First Usenix Conf. File and Storage Technologies (FAST),* Jan. 2002.

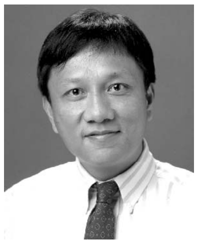
[33] E. Chang and H. Garcia-Molina, "Bubbleup—Low Latency Fast-Scan for Media Servers," *Proc. Fifth ACM Multimedia Conf.,* Nov. 1997.

[34] C. Shahabi, S. Ghandeharizadeh, and S. Chaudhuri, "On Scheduling Atomic and Composite Multimedia Objects," *IEEE Trans. Knowledge and Data Eng.,* vol. 14, no. 2, Mar./Apr. 2002.

[35] P.J. Shenoy and H.M. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems," *Proc. ACM Sigmetrics,* June 1998.

**Zoran Dimitrijević** received the PhD degree in computer science from the University of California, Santa Barbara, in 2004 and the Dipl.Ing. in electrical engineering from the School of Electrical Engineering, University of Belgrade, Serbia, in 1999. Currently, he is with Google, Inc. His research interests include operating systems, storage systems, large-scale parallel-computing architectures, and real-time streaming applications. He is a member of the IEEE and the IEEE Computer Society.

**Raju Rangaswami** received the PhD degree in computer science from the University of California, Santa Barbara, in 2004 and the BTech degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, in 1999. Currently, he is an assistant professor in the School of Computer Science at the Florida International University. His research interests include multimedia applications, file systems, operating systems, and storage. He is a member of the IEEE and the IEEE Computer Society.



**Edward Y. Chang** received the MS degree in computer science and the PhD degree in electrical engineering from Stanford University in 1994 and 1999, respectively. Since 2003, he has been an associate professor of electrical and computer engineering at the University of California, Santa Barbara. His recent research activities are in the areas of machine learning, data mining, high-dimensional data indexing, and their applications to image databases and video surveillance. Professor Chang has served on several ACM, IEEE, and SIAM conference program committees. He serves as an associate editor for the *IEEE Transactions on Knowledge and Data Engineering* and *ACM Multimedia Systems Journal*. He is a recipient of the IBM Faculty Partnership Award and the US National Science Foundation Career Award. He is a cofounder of VIMA Technologies, which provides image searching and filtering solutions. He is a senior member of the IEEE and a member of the IEEE Computer Society.