Design and Implementation of Semi-preemptible IO

Zoran Dimitrijević Raju Rangaswami Edward Chang
University of California, Santa Barbara
zoran@cs.ucsb.edu raju@cs.ucsb.edu echang@ece.ucsb.edu

Abstract

Allowing higher-priority requests to preempt ongoing disk IOs is of particular benefit to delay-sensitive multimedia and real-time systems. In this paper we propose *Semi-preemptible IO*, which divides an IO request into small temporal units of disk commands to enable preemptible disk access. We present main design strategies to allow preemption of each component of a disk access—seek, rotation, and data transfer. We analyze the performance and describe implementation challenges. Our evaluation shows that *Semi-preemptible IO* can substantially reduce IO waiting time with little loss in disk throughput. For example, expected waiting time for disk IOs in a video streaming system is reduced 2.1 times with the throughput loss of less than 6 percent.

1 Introduction

Traditionally, disk IOs have been thought of as non-preemptible operations. Once initiated, they cannot be stopped until completed. Over the years, operating system designers have learned to live with this restriction. However, non-preemptible IOs can be a stumbling block for applications that require short response time. In this paper, we propose methods to make disk IOs semi-preemptible, thus providing the operating system a finer level of control over the disk-drive.

Preemptible disk access is desirable in certain settings. 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 [9, 10, 11]. In real-time scheduling theory, *blocking*[1], or priority inversion, is defined as the time spent when a higher-priority task is prevented from running due to the non-preemptibility of a low-priority task. Blocking degrades schedulability of real-time tasks and is thus undesirable. Making disk IOs preemptible would reduce blocking and improve the schedulability of real-time disk IOs.

[1] In this paper, we refer to blocking as the *waiting time.*

Another domain where preemptible disk access is essential is that of interactive multimedia such as video, audio, and interactive virtual reality. Because of the large amount of memory required by these media data, they are stored on disks and are retrieved into main memory only when needed. For interactive multimedia applications that require short response time, a disk IO request must be serviced promptly. For 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 media data) is around 15 milliseconds [2]. Such interactive IOs can be modeled as higher-priority IO requests. However, due to the typically large IO size and the non-preemptible nature of ongoing disk commands, even such higher-priority IO requests can be kept waiting for tens, if not hundreds, of milliseconds before being serviced by the disk.

To reduce the response time for a higher-priority request, its waiting time must be reduced. The *waiting time* for an IO request is the amount of time it must wait, due to the non-preemptibility of the ongoing IO request, before being serviced by the disk. The response time for the higher-priority request is then the sum of its waiting time and service time. The *service time* is the sum of the seek time, rotational delay, and data transfer time for an IO request. (The service time can be reduced by intelligent data placement [27] and scheduling policies [26]. However, our focus is on reducing the waiting time by increasing the preemptibility of disk access.)

In this study, we explore *Semi-preemptible IO* (previously called Virtual IO [5]), an abstraction for disk IO, which provides highly preemptible disk access (average preemptibility of the order of one millisecond) with little loss in disk throughput. *Semi-preemptible IO* breaks the components of an IO job into fine-grained physical disk-commands and enables IO preemption between them. It thus separates the preemptibility from the size and duration of the operating system's IO requests.

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 possible components of the waiting time—ongoing IO's transfer time ($T_{transfer}$), rotational delay (T_{rot}), and seek time (T_{seek}).

- **Chunking $\mathbf{T_{transfer}}$.** 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 3.1).
- **Preempting $\mathbf{T_{rot}}$.** 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 (Section 3.2).
- **Splitting $\mathbf{T_{seek}}$.** *Semi-preemptible IO* can split a long seek into sub-seeks, and permits a preemption between two sub-seeks (Section 3.3).

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).

1.1 Illustrative Example

Suppose a 500 kB read-request 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 (please refer to Section 3 for equations). *Semi-preemptible IO* can reduce the waiting time by performing the following operations.

It first predicts both the seek time and rotational delay. Since the predicted seek time is long ($T_{seek} = 14$ ms), it decides to split the seek operation into two sub-seeks, 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 sub-seek 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 non-preemptible 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 Sections 3.1 and 3.4). (If T_{rot} cannot mask seek-splitting, the system can also 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 a mere 9 ms, if it arrives during one of the two seek disk commands. However, if the ongoing 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 $\frac{1}{2}\frac{2\times9^2+25\times1^2}{2\times9+25\times1+3} = 2.03$ ms, a significant reduction from 23 ms (refer to Section 3 for details).

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, but incurs little overhead itself. Preemption will always incur overhead, but it will reduce the service time for higher-priority requests. Preemptibility provides the system with the choice of trading throughput for short response time when such a tradeoff is desirable. We explore the effects of IO preemption further, in Section 4.3.

1.2 Contributions

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

- We introduce *Semi-preemptible IO*, which abstracts both read and write IO requests so as to make them preemptible. As a result, system can substantially reduce the waiting time for a higher-priority request at little or no extra cost.

- We show that making write IOs preemptible is not as straightforward as it is for read IOs. We propose one possible solution for making them preemptible.
- We present a feasible path to implement *Semi-preemptible IO*. We explain how the implementation is made possible through use of a detailed disk profiling tool.

The rest of this paper is organized as follows: Section 2 presents related research. Section 3 introduces *Semi-preemptible IO* and describes its three components. In Section 4, we evaluate our prototype. In Section 5, we make concluding remarks and suggest directions for future work.

2 Related Work

Before the pioneering work of [4, 14], it was assumed that the nature of disk IOs was inherently non-preemptible. In [4], the authors proposed breaking up a large IO into multiple smaller chunks to reduce the data transfer component ($T_{transfer}$) of the *waiting time* ($T_{waiting}$) for higher-priority requests. A minimum chunk size of one track was proposed. In this paper, we improve upon the conceptual model of [4] in three respects: 1) in addition to enabling preemption of the data transfer component, we show how to enable preemption of 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, but we propose one possible solution.

Weissel et al. [24] recently proposed Cooperative I/O, a novel IO semantics aimed to reduce the power consumption of storage subsystem by enabling applications to provide more information to OS scheduler. Similarly, in this paper we propose an IO abstraction to enable preemptive disk scheduling.

Semi-preemptible IO 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 [1, 7, 18, 22, 25]. Rotational delay masking has been proposed in multiple forms. In [8, 26], the authors present rotational-latency-sensitive schedulers, which consider the rotational position of the disk arm to make better scheduling decisions. In [13, 16, 12], the authors present *freeblock scheduling*, wherein the disk arm services background jobs using the rotational delay between foreground jobs. In [19], Seagate uses a variant of just-in-time seek in some of its disk drives to reduce power consumption and noise. *Semi-preemptible IO* uses similar techniques for 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 [3, 20, 21, 23]. *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. In [14], the authors 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 ongoing 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.

3 Semi-preemptible IO

Before introducing the concept of *Semi-preemptible IO*, we first define some terms which we will use throughout the rest of this paper. Then, we propose an abstraction for disk IO, which enables preemption of IO requests. Finally, we present our disk profiler and the disk parameters required for the implementation of *Semi-preemptible IO*.

Definitions:

- 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 (cylinder, track, sector).
- A *disk command* is a non-preemptible request issued to the disk over the IO bus (e.g., the read, write, seek, and interrogative commands).
- An *IO request* is a request for read or write access to a sequential set of logical disk blocks.

- The *waiting time* is the time between the arrival of a higher-priority IO request and the moment the disk starts servicing it.

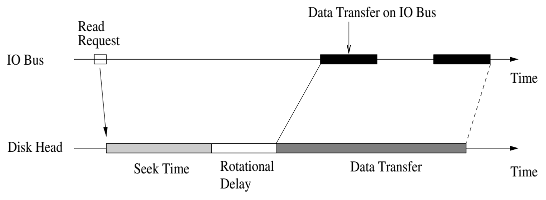


Figure 1: Timing diagram for a disk read request.

In order to understand the magnitude of the waiting time, let us consider a typical read IO request, depicted in Figure 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 a time of $T_{transfer}$, 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.

For a typical commodity system, once a disk command is issued on the IO bus, it cannot be stopped. Traditionally, an IO request 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. 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 can be as long as the duration of the ongoing 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 ongoing IO request as

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

Let V_i be the sequence of 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 duration of time during the servicing of the IO request, when the disk is idle (i.e., no disk command is issued). 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 $p_i = \frac{T_i}{\sum T_i + T_{idle}}$. Finally, the expected waiting time of a higher-priority request in *Semi-preemptible IO* can be expressed as

$$E(T'_{waiting}) = \frac{1}{2}\sum(p_i T_i) = \frac{1}{2}\frac{\sum T_i^2}{(\sum T_i + T_{idle})}. \tag{2}$$

In the remainder of this section, we present 1) *chunking*, which divides $T_{transfer}$ (Section 3.1); 2) *just-in-time seek*, which enables T_{rot} preemption (Section 3.2); and 3) *seek splitting*, which divides T_{seek} (Section 3.3). In addition, we present our disk profiler, Diskbench, and summarize all the disk parameters required for the implementation of *Semi-preemptible IO* (Section 3.4).

3.1 Chunking: Preempting $\mathbf{T_{transfer}}$

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]. To make the $T_{transfer}$ component preemptible, *Semi-preemptible IO* uses *chunking*.

Definition 3.1: *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. Issuing multiple disk commands instead of a single one also increases the CPU overhead for performing IO. However, for the range of chunk sizes, the disk throughput using chunking is optimal with negligible CPU overhead.

[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.

3.1.1 The Method

To perform chunking, the system must decide on the chunk size. *Semi-preemptible IO* chooses the minimum chunk size for which the disk throughput is optimal and the CPU overhead acceptable. Surprisingly, large chunk sizes can also suffer from throughput degradation due to the sub-optimal implementation of disk firmware (Section 3.4). Consequently, *Semi-preemptible IO* may achieve even better disk throughput than the traditional method where an IO request is serviced using a single disk command.

In order to perform chunking efficiently, *Semi-preemptible IO* relies on the existence of a read cache and a write buffer on the disk. It uses disk profiling to find the optimal chunk size. We now present the chunking for read and write IO requests separately.

3.1.2 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 [17]. Chunking for a read IO requests is illustrated in Figure 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 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 disk throughput.

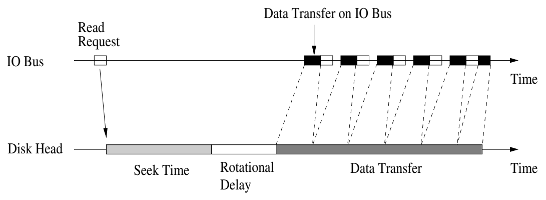


Figure 2: Virtual preemption of the data transfer.

Figure 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 Figure 3 denotes the point beyond which the performance of the cache may be sub-optimal[3].

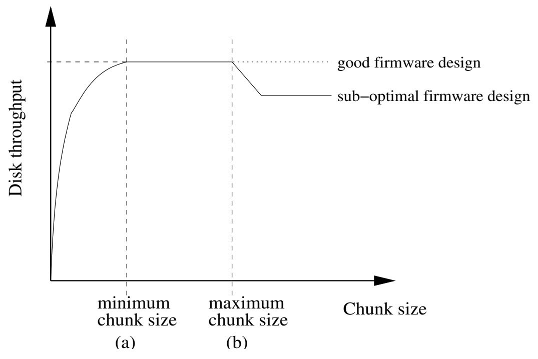


Figure 3: Effect of chunk size on disk throughput.

3.1.3 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[4]. 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 preemptibility of write requests, we must take into consideration the external waiting time for write IO requests. 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 force-unit-access flag

[3]We have not fully investigated the reasons for sub-optimal disk performance and it is the subject of our future work.

[4]If the size of the write IO is larger than the size of the write buffer, then the disk signals the end of the IO as soon as the excess amount of data (which cannot be fitted into the disk buffer) has been written to the disk media.

in SCSI write command. Using this simple technique, it triggers the write-back operation 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.

3.2 JIT-seek: Preempting $\mathbf{T_{rot}}$

After the reduction of the $T_{transfer}$ component of the waiting time, the rotational delay and seek time components become significant. 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 time, we propose a *just-in-time seek* (*JIT-seek*) technique for IO operations.

Definition 3.2: 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, 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.

3.2.1 The Method

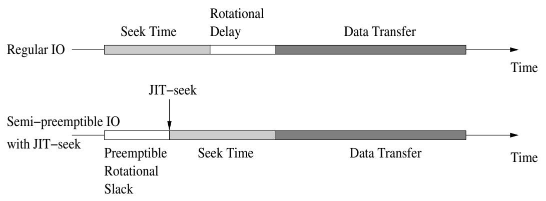


Figure 4: JIT-seek.

The JIT-seek method is illustrated in Figure 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, 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 pre-seek slack made available due to the JIT-seek operation can be used in three possible ways:

- The pree-seek 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* [13]. 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 [13, 16].
- The slack can be used to mask the overhead incurred in performing *seek-splitting*, which we shall discuss next.

3.3 Seek Splitting: Preempting $\mathbf{T_{seek}}$

The seek delay (T_{seek}) becomes the dominant component when the $T_{transfer}$ and T_{rot} components are reduced drastically. A full stroke of the disk arm may require as much as 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 3.3: *Seek-splitting* breaks a long, non-preemptible seek of the disk arm into multiple smaller sub-seeks.

Benefits: The *seek-splitting* method reduces the T_{seek} component of the waiting time. A long non-preemptible seek can be transformed into multiple shorter sub-seeks. A higher-priority request can now be serviced at the end of a sub-seek, instead of being delayed until the entire seek operation is finished. 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 sub-seeks 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 sub-seeks is greater than that for a single seek of a given seek distance. Thus, the seek-splitting method can degrade disk throughput. Later in this section, we discuss this issue further.

2. Splitting the seek into multiple sub-seeks increases the number of disk head accelerations and decelerations, consequently increasing the power usage and noise.

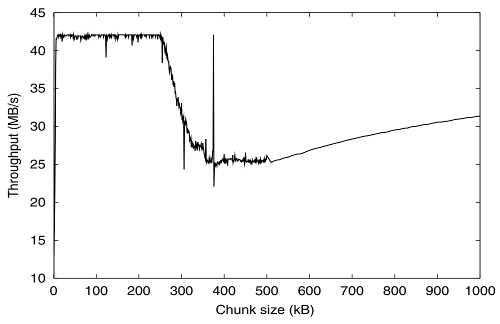
3.3.1 The Method

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

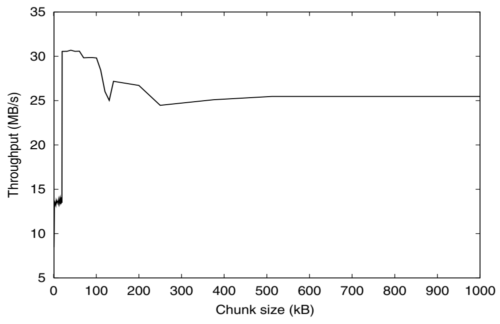
Unlike the previous two methods, seek-splitting may degrade disk performance. However, we note that the overhead due to seek-splitting can, in some cases, be masked. If the pre-seek slack obtained due to JIT-seek is greater than the seek overhead, then the slack can be used to mask this overhead. A specific example of this phenomenon was presented in Section 1. If the slack is insufficient to mask the overhead, seek-splitting can be aborted to avoid throughput degradation. Making such a tradeoff, of course, depends on the requirements of the application.

3.4 Disk Profiling

As mentioned in the beginning of this section, *Semi-preemptible IO* greatly relies on disk profiling to obtain accurate disk parameters. The disk profiler obtains the following required disk parameters:



(a) SCSI ST318437LW



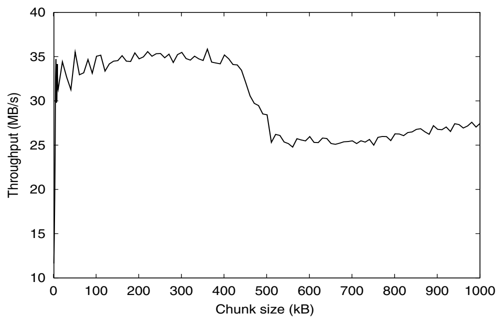
(b) IDE WD400BB

Figure 5: Sequential read throughput vs. chunk size.

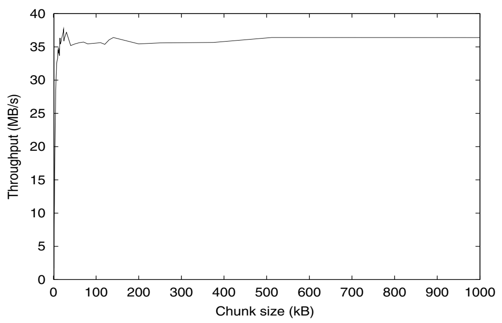
- **Disk block mappings.** System uses disk mappings for both logical-to-physical and physical-to-logical disk block address transformation.
- **The optimal chunk size.** In order to efficiently perform chunking, *Semi-preemptible IO* chooses the optimal chunk size from the optimal range extracted using disk profiler.
- **Disk rotational factors.** In order to perform JIT-seek, system requires accurate rotational delay prediction, which relies on disk rotation period and rotational skew factors for disk tracks.
- **Seek curve.** JIT-seek and seek-splitting methods rely on accurate seek time prediction.

The extraction of these disk parameters is described in [7].

As regards chunking, the disk profiler provides the optimal range for the chunk size. Figure 5 depicts the effect of chunk size on the read throughput performance for



(a) SCSI ST318437LW



(b) IDE WD400BB

Figure 6: Sequential write throughput vs. chunk size.

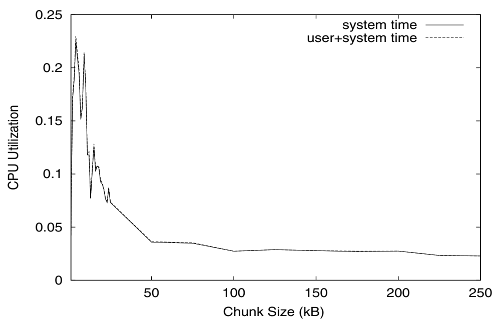


Figure 7: CPU utilization vs. chunk size for IDE WD400BB.

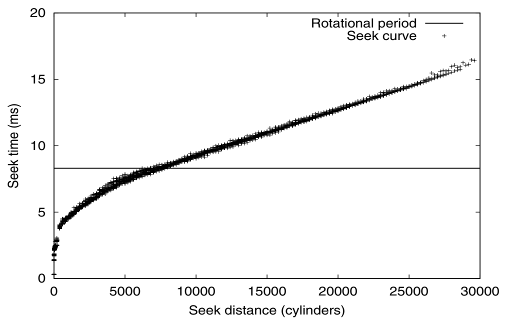


Figure 8: Seek curve for SCSI ST318437LW.

one SCSI and one IDE disk drive. Figure 6 shows the same for the write case. Clearly, the optimal range for the chunk size (between the points a and b illustrated previously in Figure 3) can be automatically extracted from these figures. The disk profiler implementation was successful in extracting the optimal chunk size for several SCSI and IDE disk drives with which we experimented. For those who might also be interested in the CPU overhead for performing chunking, we present the CPU utilization when transferring a large data segment from the disk, using different chunk sizes in Figure 7 for an IDE disk. The CPU utilization decreases rapidly with an increase in the chunk size. Beyond a chunk size of 50 kB, the CPU utilization remains relatively constant. This figure shows that chunking, using even small chunk size (50 kB), is feasible for IDE disk without incurring any significant CPU overhead. For SCSI disks, the CPU overhead of chunking is even less than that for IDE disks, since the bulk of the processing is done by the SCSI controller.

To perform JIT-seek, the system needs an accurate estimate of the seek delay between two disk blocks. The disk profiler provides the seek curve as well as the variations in seek time. The seek time curve (and variations in seek time) for a SCSI disk obtained by the disk profiler is presented in Figure 8. The disk profiler also obtains the required parameters for rotational delay prediction between accessing two disk blocks in succession with near-microsecond-level precision. However, the variations in seek time can be of the order of one millisecond, which restricts the possible accuracy of prediction. Finally, to perform JIT-seek, the system combines seek time and rotational delay prediction to predict T_{rot}. We have conducted more detailed study on T_{rot} prediction in [7].

4 Experimental Results

We now present the performance results for our implementation of *Semi-preemptible IO*. Our experiments aimed to answer the following questions:

- What is the level of *preemptibility* of *Semi-preemptible IO* and how does it influence the disk throughput?
- What are the *individual contributions* of the three components of *Semi-preemptible IO*?

- What is the effect of IO *preemption* on the average response time for higher-priority requests and the disk throughput?

In order to answer these questions, we have implemented a prototype system which can service IO requests using either the traditional non-preemptible method (*non-preemptible IO*) or *Semi-preemptible IO*. 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 curves, and the rotational skew times, all of which are automatically generated by the Diskbench [7]. All experiments were performed on a Pentium III 800 MHz machine with a Seagate ST318437LW SCSI disk. This SCSI disk has two tracks per cylinder, with 437 to 750 blocks per track depending on the disk zone. The total disk capacity is 18.4 GB. The rotational speed of the disk is 7200 RPM. The maximum sequential disk throughput is between 24.3 and 41.7 MBps.

For performance benchmarking, we performed two sets of experiments. First, we tested the preemptibility of the system using simulated IO workload. For the simulated workload, we used equal-sized IO requests within each experiment. The low-priority IOs are 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 the lower estimate for preemption overhead. We tested the preemptibility under *first-come-first-serve (FCFS)* and *elevator* disk scheduling policies. In the second set of experiments we used trace workload obtained on the tested Linux system. We obtained the traces from the instrumented Linux-kernel disk-driver. In the simulated experiments, non-preemptible IOs are serviced using chunk sizes of 128 kB. This is the size used by Linux and FreeBSD for breaking up large IOs. We assume that a large IO cannot be preempted between chunks, since such is the case for current operating systems. On the other hand, our prototype services larger IOs using multiple disk commands and preemption is possible after each disk command is completed. Based on disk profiling, our prototype used the following parameters for *Semi-preemptible IO*. Chunking divided the data transfer into chunks of 50 disk blocks each, except for the last chunk, which can be smaller. JIT-seek used an offset of 1 ms to reduce the probability of prediction errors. Seeks for more than a half of the disk size in cylinders were split into two equal-sized, smaller seeks. We used the SCSI *seek* command to perform sub-seeks.

4.1 Preemptibility

The experiments for preemptibility of disk access measure the duration of (non-preemptible) disk commands in both non-preemptible IO and *Semi-preemptible IO* 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 Equations 1 and 2, as explained in Section 3.

4.1.1 Random Workload

Figure 9 depicts the difference in the expected waiting time between non-preemptible IO and *Semi-preemptible IO*. In this experiment, IOs were serviced for data situated at random locations on the disk. The IOs were serviced using FCFS policy. We can see that the expected waiting time for non-preemptible IOs increases linearly with IO size due to increased data transfer time. However, the expected waiting time for *Semi-preemptible IO* actually decreases with IO size, since the disk spends more time in data transfer, which is more preemptible.

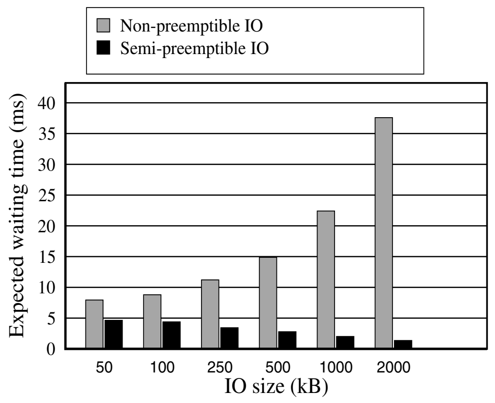


Figure 9: Improvements in the expected waiting time (FCFS).

Figure 10 depicts the improvements in the expected waiting time when the system uses an elevator-based scheduling policy. (The figure shows the results of randomly generated IO requests serviced in batches of 40.) The results are better than those of FCFS access since the elevator scheduler reduces the seek component that is the least-preemptible.

Figures 11 and 12 show the effect of improving IO preemptibility on the achieved disk throughput when an FCFS scheduling policy is used. There is a notice-

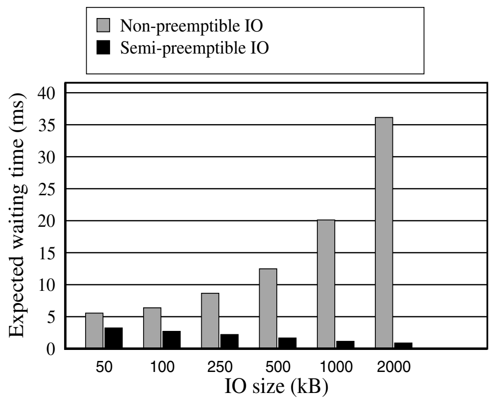


Figure 10: Improvements in the expected waiting time (Elevator).

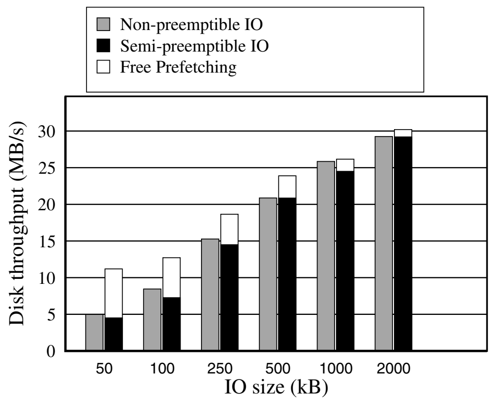


Figure 12: Effect on achieved disk throughput (Elevator).

able but minor reduction in disk throughput using *Semi-preemptible IO* (less than 15%). This reduction is due to the overhead of seek-splitting and mis-prediction of seek and rotational delay. More details on the accuracy of rotational delay predictions can be found in [7]. Another point worth mentioning is that the reduction in disk throughput in *Semi-preemptible IO* is smaller for large IOs than for small IOs due to the reduced number of seeks and hence the smaller overhead.

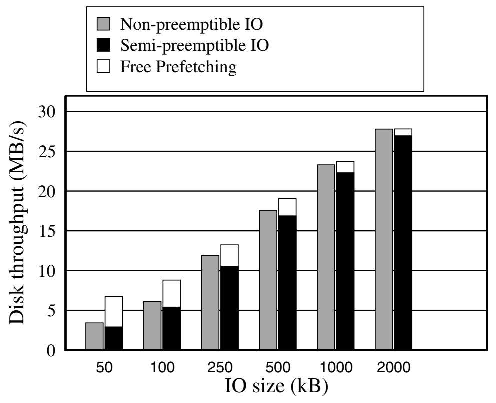


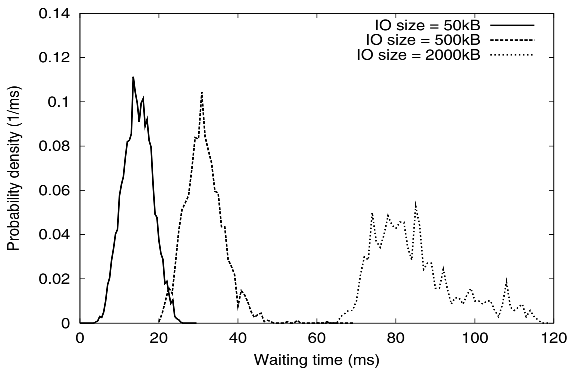
Figure 11: Effect on achieved disk throughput (FCFS).

Since disk commands are non-preemptible (even in *Semi-preemptible IO*), we can use the duration of disk commands to measure the expected waiting time. A smaller value implies a more preemptible system. Figure 13 shows the distribution of the durations of disk commands for both non-preemptible IO and *Semi-preemptible IO* (for exactly the same sequence of IO requests). In the case of non-preemptible IO (Figure 13 (a)), one IO request is serviced using a single disk command. Hence, the disk access can be preempted only when the current IO request is completed. The distribution is dense near the sum of the average seek time, rotational delay, and transfer time required to service an entire IO request. The distribution is wider when the IO requests are larger, because the duration of data transfer depends not only on the size of the IO request, but also on the throughput of the disk zone where the data resides.

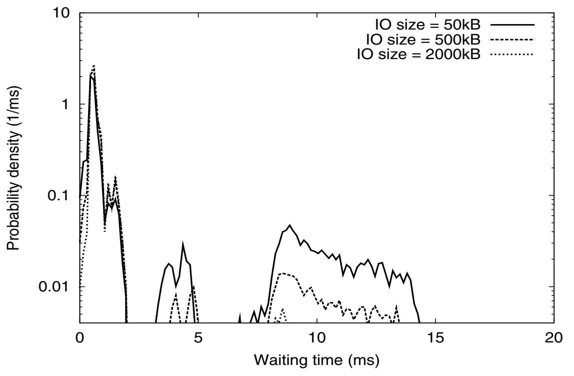
In the case of *Semi-preemptible IO*, the distribution of the durations of disk commands does not directly depend on the IO request size, but on individual disk commands used to perform an IO request. (We plot the distribution for the *Semi-preemptible IO* case in logarithmic scale, so that the probability density of longer disk commands can be better visualized.) In Figure 13 (b), we see that for *Semi-preemptible IO*, the largest probability density is around the time required to transfer a single chunk of data. If the chunk includes the track or cylinder skew, the duration of the command will be 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 has 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 is small, approximately 0.168, 0.056, and 0.017 for 50 kB, 500 kB, and 2,000 kB IO requests, respectively.

4.1.2 Trace Workload

We now present preemptibility results using IO traces obtained from a Linux system. IO traces were obtained



(a) Non-preemptible IO (linear scale)



(b) *Semi-preemptible IO* (logarithmic scale)

Figure 13: Distribution of the disk command duration (FCFS). Smaller values imply a higher preemptibility.

from three applications. The first trace (DV15) was obtained when the XTREAM multimedia system [6] was servicing 15 simultaneous video clients using the FCFS disk scheduler. The second trace (Elevator15) was obtained using the similar setup where XTREAM let Linux elevator scheduler handle concurrent disk IOs. The third was a disk trace of the TPC-C database benchmark with 20 warehouses obtained from [15]. Trace summary is presented in Table 1.

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

Table 1: Trace summary.

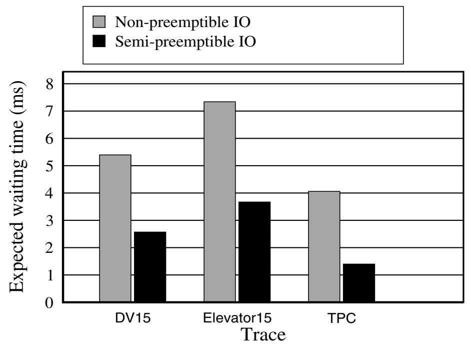


Figure 14: Improvement in the expected waiting time (using disk traces).

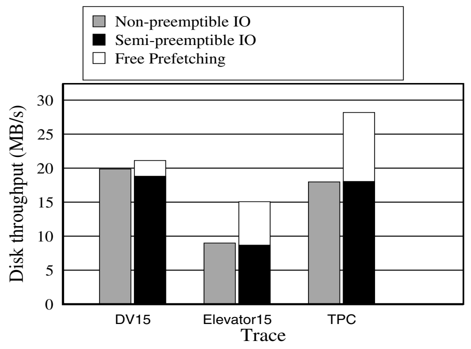


Figure 15: Effect on the achieved disk throughput (using disk traces).

Figures 14 and 15 show the expected waiting time and disk throughput for these trace experiments. The expected waiting time was reduced by as much as 65% (Figure 14) with less than 10% (Figure 15) 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 Individual Contributions

Figure 16 shows the individual contributions of the three strategies with respect to expected waiting time for the random workload with the elevator scheduling policy. In Section 4.1, we showed that the expected waiting time can be significantly smaller in *Semi-preemptible IO* than in non-preemptible IO. Here we compare only contributions within *Semi-preemptible IO* to show the importance of each strategy. Since the time to transfer a single chunk of data is small compared to the seek time (typically less than 1 ms for a chunk transfer and 10 ms for a seek), the expected waiting time decreases as the data transfer time becomes more dominant. When the data

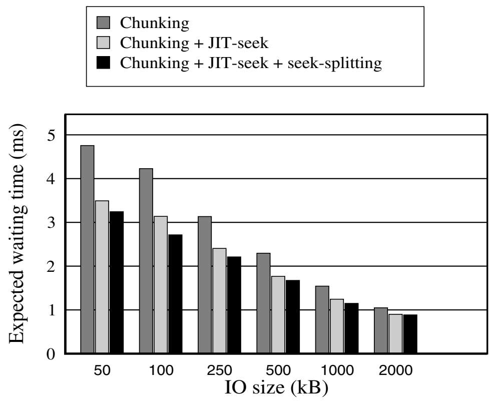


Figure 16: Individual contributions of *Semi-preemptible IO* components on the expected waiting time (Elevator).

transfer time dominates the seek and rotational delays, chunking is the most useful method for reducing the expected waiting time. When the seek and rotational delays are dominant, JIT-seek and seek-splitting become more effective for reducing the expected waiting time.

Figure 17 summarizes the individual contributions of the three strategies with respect to the achieved disk throughput. Seek-splitting can degrade disk throughput, since whenever a long seek is split, the disk requires more time to perform multiple sub-seeks. JIT-seek requires accurate prediction of the seek time and rotational delay. It introduces overhead in the case of mis-prediction. However, when the data transfer is dominant, benefits of chunking can mask both seek-splitting and JIT-seek overheads. JIT-seek aids the throughput with free prefetching. The potential free disk throughput acquired using free prefetching depends on the rate of JIT-seeks, which decreases with IO size. We believe

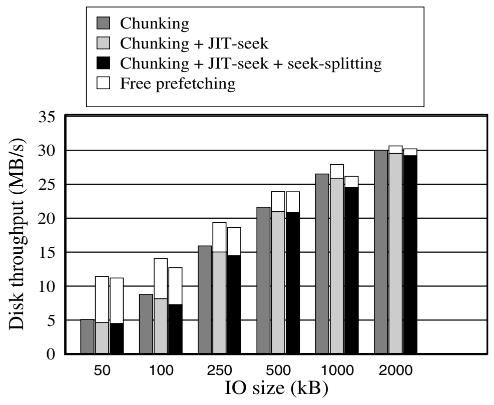


Figure 17: Individual effects of *Semi-preemptible IO* strategies on disk throughput (Elevator).

that the free prefetching is a useful strategy for multimedia systems that often access data sequentially and hence can use most of the potential free throughput.

4.3 Effect of Preemption

To estimate the response time for higher-priority IO requests, we conducted experiments wherein higher-priority requests were inserted into the IO queue at a constant rate (ν). While the constant arrival rate may seem unrealistic, the main purpose of this set of experiments is only to "estimate" the benefits and overheads associated with preempting an ongoing *Semi-preemptible IO* request to service a higher-priority IO request.

Table 2 presents the response time for a higher-priority request when using *Semi-preemptible IO* in two possible scenarios: (1) when the higher-priority request is serviced after the ongoing IO is completed (non-preemptible IO), and (2) when the ongoing IO is preempted to service the higher-priority IO request (*Semi-preemptible IO*). If the ongoing IO request is not preempted, then all higher-priority requests that arrive while it is being serviced, must wait until the IO is completed. The results in Table 2 illustrate the case when the ongoing request is a read request. The results for the write case are presented in Table 4.

IO $[kB]$	ν $[req/s]$	*Avg. Resp.* $[ms]$ *npIO*	*spIO*	*Throughput* $[MB/s]$ *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 2: The average response time and disk throughput for non-preemptible IO ($npIO$) and *Semi-preemptible IO* ($spIO$).

Preemption of IO requests is not possible without overhead. Each time a higher-priority request preempts a low-priority IO request for disk access, an extra seek is required to continue servicing the preempted request after the higher-priority request has been completed. 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 does not increase significantly with the increase in the arrival rate of higher-priority requests. However, the disk throughput does decrease with an increase in the arrival rate of higher-priority requests. As explained earlier, this reduction is expected since the overhead of IO preemption is an extra seek operation per preemption. For applications that require short response time, the performance penalty of IO preemption seems acceptable.

4.3.1 External Waiting Time

In Section 3.1, we explained the difference in the preemptibility of read and write IO requests and introduced the notion of external waiting time. Table 3 summarizes the effect of external waiting time on the preemption of write IO requests. The arrival rate of higher-priority requests is set to $\nu = 1$ req/s. As shown in Table 3, the average response time for higher-priority requests for write experiments is several times longer than for read experiments. Since the higher-priority requests have the same arrival pattern in both experiments, the average seek time and rotational delay are the same for both read and write experiments. The large and often unpredictable external waiting time in the write case explains these results.

	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

Table 3: The expected waiting time and average response time for non-preemptible and *Semi-preemptible IO* ($\nu = 1$ req/s).

Table 4 presents the results of our experiments aimed to find out the effect of write IO preemption on the average response time for higher-priority requests and disk write throughput. For example, in the case of 50 kB write IO requests, the disk can buffer multiple requests, and the write-back operation can include multiple seek operations. *Semi-preemptible IO* succeeds in reducing external waiting time and provides substantial improvement in the response time. However, since the disk is able to efficiently reorder the buffered write requests in the case of non-preemptible IO, it achieves better disk throughput. For large IO requests, *Semi-preemptible IO* achieves write throughput comparable to that of non-preemptible IO. We suggest that write preemption can be disabled when maintaining high system throughput is essential, and the disk reordering is useful (reordering could also be done in the operating system scheduler using the low-level disk knowledge).

IO	ν	*Avg. Response* $[ms]$		*Throughput* $[MB/s]$	
$[kB]$	$[req/s]$	*npIO*	*spIO*	*npIO*	*spIO*
50	*0.5*	93.1	26.9	4.85	1.98
50	*1*	105.8	24.6	4.75	1.96
50	*2*	91.1	22.7	4.68	1.94
50	*5*	102.2	24.4	4.40	1.84
50	*10*	87.5	23.7	3.95	1.70
50	*20*	81.3	23.3	3.09	1.42
500	*0.5*	32.4	20.3	13.71	11.41
500	*1*	36.0	20.3	13.64	11.24
500	*2*	35.0	20.8	13.45	11.02
500	*5*	34.9	20.5	12.82	10.36
500	*10*	36.6	20.3	11.67	9.13
500	*20*	34.6	20.7	9.64	6.92

Table 4: The average response time and disk write throughput for non-preemptible and *Semi-preemptible IO*.

5 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—data transfer chunking, 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 IO request. Using both synthetic and trace workloads, we have shown that these techniques can be efficiently implemented, given detailed disk parameters. Our empirical studies showed that *Semi-preemptible IO* can reduce the waiting time for both read and write requests significantly when compared with non-preemptible IOs.

We believe that preemptible IO can especially benefit multimedia and real-time systems, which are delay sensitive and which issue large-size IOs for meeting real-time constraints. We are currently implementing *Semi-preemptible IO* in Linux kernel. We plan to further study its performance impact on traditional and real-time disk-scheduling algorithms.

6 Acknowledgements

This project is partially supported by NSF grants IIS-0133802 (Career) and IIS-0219885 (ITR), and a UC DiMI grant. The third author is supported by an IBM Faculty Partnership Award. The authors wish to thank our reviewers and especially Erik Riedel from Seagate Research for their valuable comments and suggestions.

References

[1] M. Aboutabl, A. Agrawala, and J.-D. Decotignie. Temporally determinate disk access: An experimental approach. *Univ. of Maryland Technical Report CS-TR-3752*, 1997.

[2] R. T. Azuma. Tracking requirements for augmented reality. *Communications of the ACM*, 36(7):50–51, July 1993.

[3] E. Chang and H. Garcia-Molina. Bubbleup - Low latency fast-scan for media servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 87–98, November 1997.

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

[5] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Virtual IO: Preemptible disk access (poster). *Proceedings of the ACM Multimedia*, December 2002.

[6] Z. Dimitrijevic, R. Rangaswami, and E. Chang. The XTREAM multimedia system. *IEEE Conference on Multimedia and Expo*, August 2002.

[7] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya. Diskbench. *http://www.cs.ucsb.edu/∼zoran/papers/db01.pdf*, November 2001.

[8] L. Huang and T. Chiueh. Implementation of a rotation-latency-sensitive disk scheduler. *SUNY at Stony Brook Technical Report*, May 2000.

[9] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, December 1991.

[10] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *Software Engineering*, 19(9):920–934, 1993.

[11] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *ACM Journal*, January 1973.

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

[13] 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.

[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] Performance Evaluation Laboratory, Brigham Young University. Trace distribution center. *http://tds.cs.byu.edu/tds/*, 2002.

[16] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle. Data mining on an OLTP system (nearly) for free. *Proceedings of the ACM SIGMOD*, May 2000.

[17] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 2:17–28, 1994.

[18] J. Schindler and G. R. Ganger. Automated disk drive characterization. *CMU Technical Report CMU-CS-00-176*, December 1999.

[19] Seagate Technology. Seagate's sound barrier technology. *http://www.seagate.com/docs/pdf/whitepaper/sound_barrier.pdf*, November 2000.

[20] C. Shahabi, S. Ghandeharizadeh, and S. Chaudhuri. On scheduling atomic and composite multimedia objects. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):447–455, 2002.

[21] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Proceedings of the ACM Sigmetrics*, June 1998.

[22] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based extraction of local and global disk characteristics. *UC Berkeley Technical Report*, 1999.

[23] W. Tavanapong, K. Hua, and J. Wang. A framework for supporting previewing and vcr operations in a low bandwidth environment. *Proceedings of the 5th ACM Multimedia Conference*, November 1997.

[24] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O—A novel I/O semantics for energy-aware applications. *Proceedings of the OSDI*, December 2002.

[25] B. L. Worthington, G. Ganger, Y. N. Patt, and J. Wilkes. Online extraction of scsi disk drive parameters. *Proceedings of the ACM Sigmetrics*, pages 146–156, 1995.

[26] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. *Proceedings of the ACM Sigmetrics*, pages 241–251, May 1994.

[27] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. *Proceedings of the OSDI*, October 2000.