Virtual IO: Preemptible Disk Access

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

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

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

ABSTRACT

Supporting preemptible disk access is essential for interactive multimedia applications that require short response time. In this study, we propose Virtual IO, an abstraction for disk IO, that transforms a non-preemptible IO request into a preemptible one. In order to achieve its objective efficiently, Virtual IO uses disk profiling to obtain accurate and detailed knowledge about the disk. Upon implementation of Virtual IO, we show that not only does Virtual IO enable highly preemptible disk access, but it does so with little or no loss in disk throughput.

1. INTRODUCTION

Many varieties of media such as video, audio, and interactive virtual reality are proliferating. 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 which 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 [1]. Such interactive IO requests can be modeled as higher-priority IO requests. However, the disk might already be servicing an IO request when the higher-priority IO request arrives. Due to the typically large media-IO size and the non-preemptible nature of an ongoing disk IO, even higher-priority IO requests can be kept waiting for at least tens, if not hundreds, of milliseconds before they are 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 defined as 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 and can be reduced by intelligent data placement policies [19]. However, our focus is on reducing the waiting time by increasing the preemptibility of disk access.

In this study, we propose *Virtual IO*, an abstraction for disk IO, which provides highly preemptible disk access with little or no loss in disk throughput. Virtual IO breaks the components of an IO job into fine-grained physical disk-commands and enables IO preemption between any two disk commands. Let $T_{waiting}$ denote the waiting time for a higher-priority IO request which arrives when the disk is currently servicing an ongoing IO request. The waiting time for the higher-priority request, $T_{waiting}$, can include seek time (T_{seek}), rotational delay (T_{rot}), and data transfer time ($T_{transfer}$) for the ongoing IO request. If a higher-priority IO request can arrive at any time during the service time for the ongoing IO request with equal probability, then the expected value for the waiting time of the higher-priority request can be expressed as

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

Virtual IO maps each IO request into multiple fast-executing disk commands using three methods. The ongoing IO request can now be preempted to service another higher-priority IO between two disk commands. Each method within Virtual IO addresses the reduction of one of the components of the waiting time.

- **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 $\mathbf{T_{seek}}$.** Virtual IO splits a long seek into sub-seeks, and permits a preemption between two sub-seeks (Section 3.3).

Virtual IO services a single IO request using multiple disk commands. Let V_i be the sequence of disk commands used by Virtual IO to execute the IO request. Let the time required to execute the disk command V_i be T_i. The expected waiting time[1] using Virtual IO can then be expressed as:

$$E(T'_{waiting}) = \frac{1}{2}\frac{\sum T_i^2}{\sum T_i}.$$

The following example illustrates how Virtual IO improves the expected waiting time for a high-priority request.

[Illustrative Example] Suppose a 2 MB read-request has to seek $20,000$ cylinders requiring T_{seek} of 14 ms, must wait for a T_{rot} of 8 ms, and requires $T_{transfer}$ of 100 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 61 ms, while the maximum waiting time is 122 ms. Virtual 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

[1]Please refer to Section 3 for the derivation of this equation.

the seek operation into two sub-seeks, each of $10,000$ cylinders, requiring $T'_{seek} = 9$ ms each. Please note that this seek splitting does not cause extra overhead because the $T_{rot} = 8$ can mask the 4 ms increased total seek time ($2 \times T'_{seek} - T_{seek} = 2 \times 9 - 14 = 4$) incurred by seek splitting. The rotational delay is now $T'_{rot} = T_{rot} - (2 \times T'_{seek} - T_{seek}) = 4$ ms.

With this $T'_{rot} = 4$ ms knowledge, Virtual IO can wait for 4 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 100 successive read commands, each of size 20 kB, requiring 1 ms each. A high-priority IO request can be serviced immediately after each disk-command. Virtual IO thus makes preemptible the 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) nor additional disk latency. (Please note that if T_{rot} cannot mask seek-splitting, we can choose not to perform seek-splitting.)

- *A higher-priority IO arrives.*
  In this case, the maximum waiting time for the high-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 high-priority request is just 1 ms. The expected value for waiting time is only $\frac{1}{2}\ \frac{0\times4^2+2\times9^2+100\times1^2}{4+2\times9+100} = 1.1$ ms, a significant reduction from 61 ms.

This example shows that Virtual IO drastically reduces $E(T_{waiting})^2$.

As the above example may reveal, in order to implement Virtual IO, we must know accurate disk parameters (e.g., the chunk size, seek time, rotational delay between disk blocks, etc.) to perform effective chunking, just-in-time seek, and seek splitting. These disk parameters differ between disk models. Moreover, even disks which are of the same model and from the same vendor can be different [5]. We show that the disk profiler we have implemented, Diskbench [5], can extract essential disk information so that accurate disk-performance prediction is feasible within Virtual IO. In addition, as we will show in Section 3.1, write IOs behave differently from read IOs. In summary, the contributions of this paper are as follows:

- We introduce Virtual IO, which abstracts both read and write IO requests so as to make them highly preemptible. More significantly, it achieves this objective with little or no loss in disk throughput. As a result, Virtual IO can drastically reduce the waiting time for a higher-priority request at little or no extra cost.

- We show a feasible path to implement Virtual IO. We explain how the implementation of Virtual IO is made possible through Diskbench, a disk profiling tool that we have previously devised [5].

[2]Virtual IO 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 itself has little overhead. IO preemption may result in degradation in disk throughput and might not always be desirable. We explore the effects of IO preemption further, in Section 4.3.

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

2. RELATED WORK

In the past, little need has been expressed for highly preemptible disk access. However, new applications like immersive virtual reality, 3D gaming, etc., require a high level of responsiveness from the system, thus necessitating preemption of ongoing disk IO requests to retrieve more urgently required data.

Before the pioneering work of [3, 9], it was assumed that the nature of disk IOs was inherently non-preemptible. In [3], 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 high-priority requests. A minimum chunk size of one track was proposed. In this paper, we present the implementation of Virtual IO, which improves upon the conceptual model in the three major aspects:

1. In addition to reducing the data transfer component of the waiting time, we show how the T_{rot} and T_{seek} components can also be reduced. This further improves the preemptibility of a system, and reduces the waiting time for a high-priority request.

2. Even for the data transfer component, we show that the bounds for zero-overhead preemptibility proposed by [3] are too tight and do not apply to current disk drive technology. We propose bounds which are far more relaxed.

3. To the best of our knowledge, the preemptibility of write IOs has not received sufficient attention in the past. It is more difficult to make write IO requests preemptible than to do so with read IOs. We propose one possible solution for making write IOs preemptible.

Virtual IO uses a *just-in-time seek* technique to make the rotational delay preemptible. In addition, JIT-seek can mask the rotational delay with useful data prefetching. In order to implement both methods, Virtual IO relies on the accurate disk profiling [17, 11, 15, 5]. Rotational delay masking has been proposed in multiple forms. In [18, 7], the authors present the rotational-latency-sensitive schedulers, which consider the rotational position of the disk arm to make better scheduling decisions. In [8], the authors present *freeblock scheduling*, wherein the disk arm services background jobs using the rotational delay between foreground jobs. Virtual IO differs from these approaches not only in its primary goal, which is to make rotational delays preemptible, but also in the method employed for increasing throughput. Virtual IO uses *free prefetching* (introduced in Section 3.2) to increase disk throughput.

There is a large body of literature proposing IO scheduling policies for multimedia and real-time systems that improve disk response time [6, 16, 2, 14, 13, 12]. Virtual IO, however, is orthogonal to these contributions. We believe that the existing methods can benefit from using preemptible Virtual IO, to further decrease response time for high-priority requests. For instance, to model real-time disk IOs, one can draw from real-time CPU scheduling theories. In [9], 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 requires a certain preemption granularity which must be independent of system variables like IO sizes. Virtual IO provides exactly such an ability.

3. VIRTUAL IO

Before introducing the concept of *Virtual IO*, we first define some terms which we will use throughout the rest of this paper. Then, we propose Virtual IO, an abstraction for disk IO, which enables preemption of IO requests. Finally, we present our disk profiler [5] and the disk parameters required for the implementation of Virtual 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. Examples of disk commands are the read, write, and seek 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.
- The *expected waiting time* is the expected value for the waiting time for a higher-priority IO request.

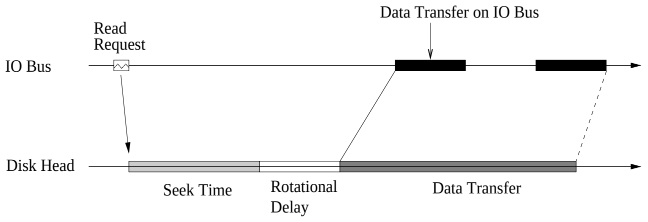


Figure 1: Timing diagram for a disk read request.

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 via the IO controller.

For a typical commodity system, once a disk command is issued on the IO bus, it cannot be stopped[3]. Traditionally, an IO request is serviced using a single disk command. Consequently, the 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

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

[3]In this paper we investigate the possible level of preemptibility of IO requests when the disk commands are non-preemptible.

To reduce the waiting time, we propose Virtual IO, which judiciously services an IO request using fine-grained disk commands. Virtual IO enables preemption of each of the above waiting-time components using three techniques: chunking $T_{transfer}$, preempting T_{rot}, and splitting T_{seek}, with little or no loss in disk throughput. Let V_i be the sequence of fine-grained disk commands used by Virtual IO to service an IO request. Let the time required to execute disk-command V_i be T_i. Using 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 $p_i = \frac{T_i}{\sum T_i}$. Finally, the expected waiting time of a higher-priority request in Virtual IO can be expressed as

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

In the remainder of this section, we present 1) *chunking*, which divides $T_{transfer}$ (Section 3.1); 2) *just-in-time seek*, which masks T_{rot} (Section 3.2); and 3) *seek splitting*, which divides T_{seek} (Section 3.3). In addition, we present our disk profiler, Diskbench [5], and summarize all the disk parameters required for the implementation of Virtual IO (Section 3.4).

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

Preemption of the data transfer component ($T_{transfer}$) in disk IOs is important since it can be large (e.g., in multimedia applications). A 2 MB IO requires 100 ms at a data transfer rate of 20 MBps. To make the $T_{transfer}$ component preemptible, Virtual 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 having to wait for the entire IO to complete. For example, suppose a 2 MB IO request requires a $T_{transfer}$ of 100 ms at a transfer rate of 20 MBps. Using a chunk size of 20 kB, the expected waiting time for a high priority request is reduced from 50 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.

3.1.1 The Method

To perform chunking, Virtual IO must decide on the chunk size. Virtual IO chooses the minimum chunk size for which the disk throughput is optimal and CPU overhead acceptable. Surprisingly, very large chunk sizes can also suffer from throughput degradation due to the sub-optimal implementation of disk firmware. Consequently, Virtual 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, Virtual IO relies on the existence of a read cache and a write buffer on the disk. To extract the optimal range for the chunk size, Virtual IO uses disk profiling. Since the chunking method is different for read and write IO requests, we now present these methods 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 [10]. 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 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.

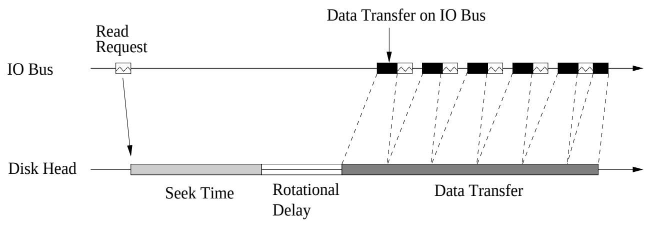


Figure 2: Virtual preemption of the data transfer component.

The effect of the chunk size on the disk throughput is explained using a mock disk in Figure 3. The optimal chunk size lies between a and b. A smaller chunk size reduces the waiting time for a higher-priority request. Hence, Virtual 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 can become a bottleneck. We explain this behavior further in an extended version [4] using an analytical model. Point b in Figure 3 denotes the point beyond which the performance of the cache may be sub-optimal. We observed surprising effect in many disks that we experimented with. We believe that this behavior can be mostly attributed to the design of the disk firmware. Points a and b in Figure 3 can both be extracted using our disk profiler (please see Section 3.4 for details).

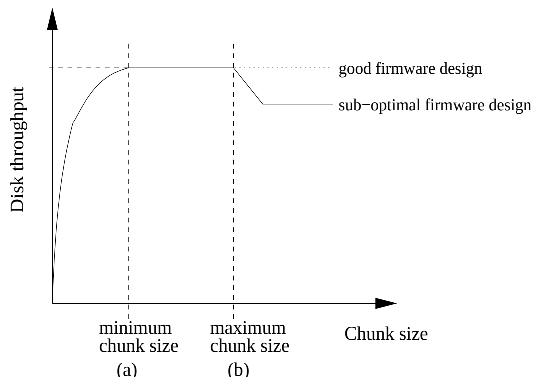


Figure 3: Effect of chunk size on disk throughput.

The Write Case

Virtual IO performs chunking for write IO requests in the same manner as it does 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 request. However, the disk may choose to schedule a write-back operation for disk write buffers before servicing a new disk command issued by the operating system. We refer to this delay as the *external waiting time* (T_{ext}). 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 IO requests.

In order to increase preemptibility of write requests, Virtual IO must take into consideration the external waiting time for write IO requests[5]. To remedy external waiting, Virtual IO forces the disk to write the last chunk (obtained after chunking) of the write IO request to disk media. Using this simple technique, Virtual IO triggers the write-back operation at the end of each write IO request. 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 the 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 IO request by 4 ms. Then, the expected waiting time is reduced from 2.5 ms to $\frac{1}{2}\frac{0\times4+1\times1}{1+4} = 0.1$ ms.

2. The slack obtained due to JIT-seek can also be used to perform data prefetching for the previous IO stream. If prefetching for the previous IO stream is useful, then JIT-seek can increase the disk throughput.

Overhead: Virtual 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 can be one extra rotation of the disk.

3.2.1 The Method

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,

[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.

[5]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.

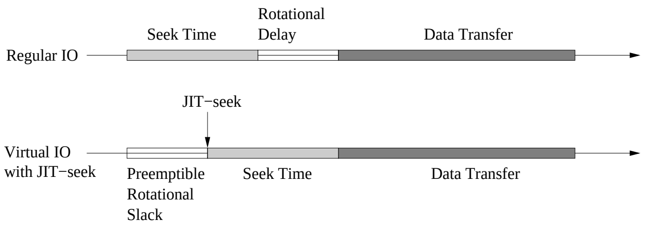


Figure 4: JIT-seek.

data transfer immediately follows the seek operation. The rotational slack available, 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 slack can be simply left unused. In this case, a higher-priority request arriving during the slack time can be serviced immediately.
- The pre-seek slack time can be used to perform useful data prefetching for the current IO request beyond the necessary data transfer. We refer to it as *free prefetching*. Chunking is used for the prefetched data, to reduce the waiting time of a higher-priority request. Free prefetching thus employed can increase the disk throughput considerably. 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.
- The slack can also be used to mask the overhead incurred in performing *seek-splitting*, which we shall discuss next.

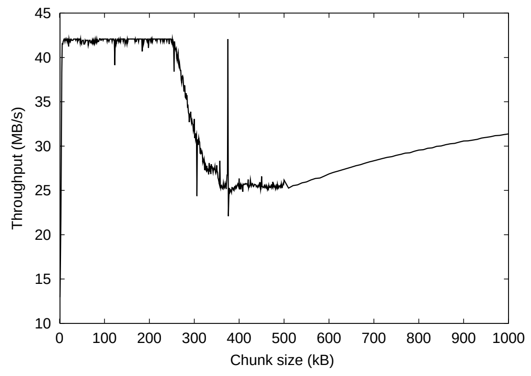
3.3 Seek Splitting: Preempting T_{seek}

The seek delay (T_{seek}) becomes the dominant component when the $T_{transfer}$ and T_{rot} components are reduced 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.

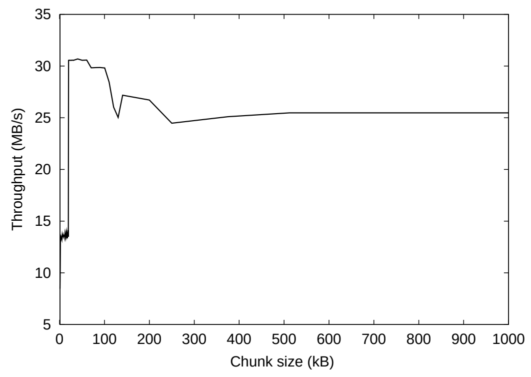
Definition 3.3: *Seek-splitting* breaks a long, non-preemptible seek of the disk arm into multiple smaller sub-seeks.

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 waiting for the entire seek operation to finish. 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 sub-seeks, each of $10,000$ cylinders, requiring 9 ms each. Then the expected waiting time for a higher-priority request is reduced from 7 ms to 4.5 ms.

Overhead: There is a downside to using the seek-splitting method. 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 given seek distance. Thus, the seek-splitting method can degrade disk throughput. We discuss this issue further later in this section.



(a) SCSI ST318437LW



(b) IDE WD400BB

Figure 5: Sequential read throughput vs. chunk size.

3.3.1 The Method

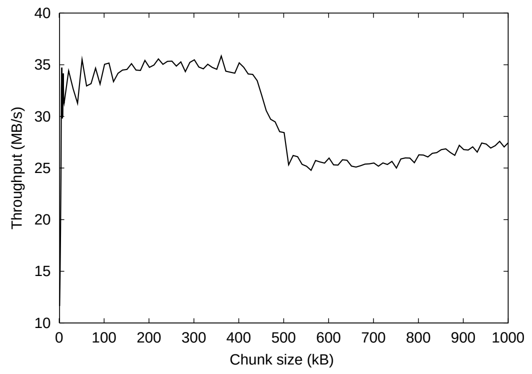
To split seek operations, virtual IO uses a tunable parameter, the maximum sub-seek distance. The *maximum sub-seek distance* decides whether to split a seek operation. For seek distances smaller than the maximum sub-seek distance, seek-splitting is not employed. A smaller value for the maximum sub-seek 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. Such a tradeoff, of course, depends on the requirements of the application.

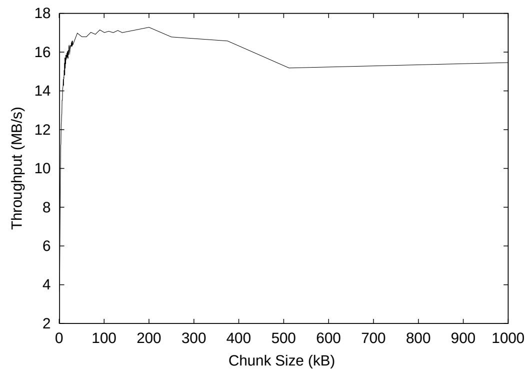
3.4 Disk Profiling

As mentioned in the beginning of this section, Virtual IO greatly relies on disk profiling to obtain accurate disk parameters. The disk profiler runs once before Virtual IO is used for the first time to obtain the following required disk parameters:

- **Disk block mappings.** Virtual IO uses disk mappings for both logical-to-physical and physical-to-logical disk block address transformation.



(a) SCSI ST318437LW



(b) IDE WD400BB

Figure 6: Sequential write throughput vs. chunk size.

- **The optimal chunk size.** In order to efficiently perform chunking, Virtual IO uses the optimal range for chunk sizes.

- **Disk rotational factors.** In order to perform JIT-seek, Virtual IO requires accurate rotational delay prediction, which requires disk rotation period and rotational skew factors for disk tracks [5].

- **Seek curve.** JIT-seek and seek-splitting methods rely on accurate seek time prediction.

One of the main parameters which is essential to Virtual IO are the disk block mappings (logical-to-physical and vice-versa). These block mappings aid in predicting disk behavior accurately in various scenarios. The extraction of these disk mappings is described in [5]. We now discuss the parameters specific to each of the three methods described earlier in this section.

As regards chunking, the disk profiler provides virtual IO the optimal range for the chunk size. Figure 5 depicts the effect of chunk size on the read throughput performance for one SCSI and one IDE disk drive. Figure 6 shows the same for the write case. Clearly, the optimal range for the chunk size 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. One might also be interested in the CPU overhead for performing chunking. We present the CPU utilization to transfer a large data segment from the disk,

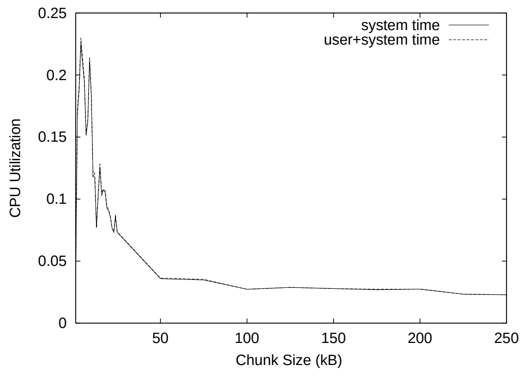


Figure 7: CPU utilization vs. chunk size for IDE WD400BB.

using different chunk sizes in Figure 7 for an IDE disk. We note that 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. Then, the overhead for small chunk sizes can be estimated using this constant value which depicts the CPU utilization for large IO requests. 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.

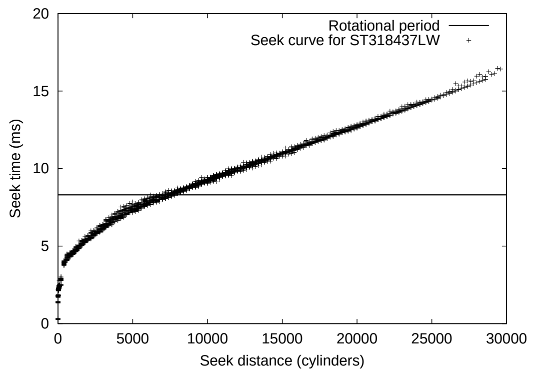


Figure 8: Sample seek curve.

To perform JIT-seek, Virtual IO 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 [5]. 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, Virtual IO combines seek time and rotational delay prediction to predict T_{rot}. We have conducted more detailed study on T_{rot} prediction in [5].

To perform seek-splitting, the disk profiler extracts the seek time curve for the disk. The seek time curve and variations in seek time for a SCSI disk obtained by the disk profiler is presented in Figure 8. Apart from the seek curve, seek-splitting also uses the disk block mappings provided by the profiler to calculate the seek distance in cylinders between two IO requests. Based on the disk block mappings, intermediate physical block destinations for subseeks are obtained for a given seek operation.

4. EXPERIMENTAL RESULTS

In this section, we present the performance results for Virtual IO. Our experiments aimed to answer the following questions:

1. What is the level of *preemptibility* of Virtual IO and how does it influence the disk throughput?

2. What are the *individual contributions* of the three components of Virtual IO?

3. 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 Virtual 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 [5]. 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 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 use equal-sized IO requests within each experiment. The low-priority IO requests are for data located at random positions on the disk. No scheduling algorithm is employed for the purposes of generality. The Virtual IO prototype services a non-preemptible IO request using a single disk command. Based on the disk profiling, our prototype uses the following parameters for Virtual IO. Chunking divides the data transfer into chunks of 50 disk blocks each, except for the last chunk, which can be smaller. JIT-seek uses an offset of 1.5 ms to reduce the probability of prediction misses. Seeks for more than half of a disk size in cylinders are split into two equal-sized, smaller seeks.

4.1 Preemptibility of Virtual IO

In this section, we aim to answer our first question: What is the level of preemptibility of Virtual IO and how does it influence the disk throughput? The experiments for preemptibility of disk access measure the duration of (non-preemptible) disk commands in both non-preemptible IO and Virtual 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 Equation 1 and 2, as explained in Section 3.

Figure 9 depicts the difference in the expected waiting time between non-preemptible IO and Virtual IO. We can see that the expected waiting time in non-preemptible IO depends linearly on the size of IO requests. This is to be expected, since the time needed to complete one IO request increases due to the larger data transfer time for a larger IO requests. However, the expected waiting time in Virtual IO does not depend on the IO size. The expected waiting time actually decreases for large IOs, since a disk spends more time in data transfer, which has a higher preemptibility than the seek component. We note that Virtual IO can reduce the expected waiting time by more than an order of magnitude, especially in systems with large IO requests.

Figure 10 shows disk throughput results in our experiments for non-preemptible and Virtual IO. Virtual IO provides IO preemptibility with little or no loss in disk throughput. Furthermore, the disk throughput for large IOs in Virtual IO is better than in non-preemptible IO. This seemingly unexpected result occurs because Virtual IO uses optimal chunk size to transfer data (see Sections 3.1 and 3.4 for details). For the individual contributions of each of the three components of Virtual IO, please refer to Section 4.2.

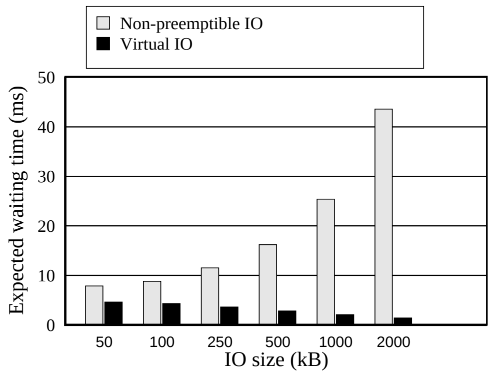


Figure 9: Improvements in the expected waiting time.

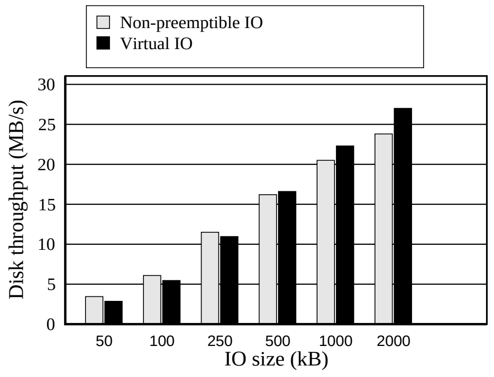
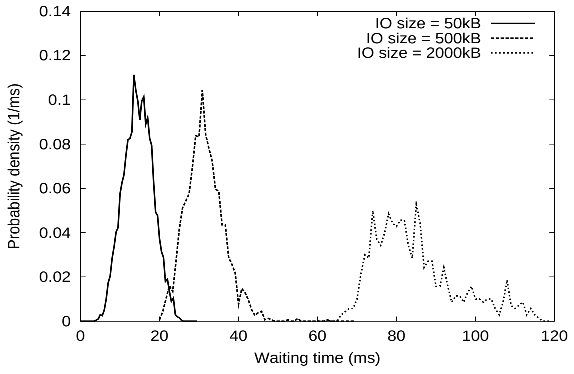


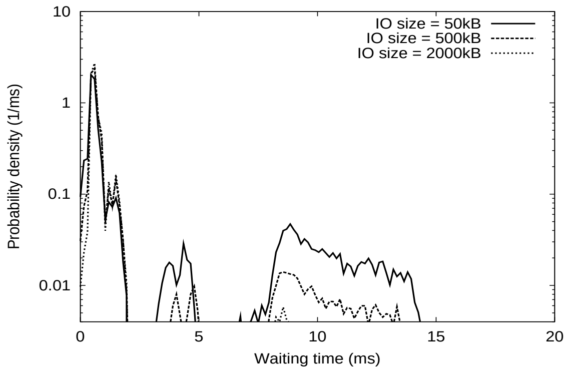
Figure 10: Effects on the achieved disk throughput.

Since disk commands are non-preemptible (even in Virtual IO), we use the duration of disk commands to measure the preemptibility of the disk access, a smaller value implying a more preemptible system. Figure 11 shows the distribution of the durations of disk commands for both non-preemptible IO and Virtual IO (for exactly the same sequence of IO requests). In the case of non-preemptible IO (Figure 11 (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. We note that 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 [10, 5].

In the case of Virtual IO, the distribution of the durations of disk commands does not depend on the IO request size, but on individual disk commands used to perform an IO request. (Please note that we plot the distribution for the Virtual IO case in logarithmic scale, so that the probability density of longer disk commands can be better comprehended.) In Figure 11 (b), we see that for Virtual 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 second part of the distribution (between approximately 3 ms and 16 ms) is due to the combined effect of JIT-seek and seek-splitting on the seek and rotational delays. However, the probability for this range is small, approximately 0.168, 0.056, and 0.017 for 50 kB, 500 kB, and 2,000 kB equal-sized IO requests in our experiments.



(a) Non-preemptible IO (linear scale)



(b) Virtual IO (logaritmic scale)

Figure 11: Distribution of the disk command duration. Smaller values imply a higher preemptibility of disk access.

The results of our experiments show that Virtual IO indeed succeeds in providing a high-level of preemptibility of disk access without significant degradation in disk throughput.

4.2 Individual Contributions within Virtual IO

In this section, we aim to answer our second question: What are the *individual contributions* of the three components of Virtual IO? Figure 12 shows the individual contributions of the three Virtual IO components with respect to expected waiting time. In Section 4.1, we showed that the expected waiting time can be significantly smaller in Virtual IO than in non-preemptible IO. Here we compare only contributions within Virtual IO to show the importance of each component. 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 transfer time dominates the seek and rotational delays, chunking is the most important method for reducing the expected waiting time. When the seek and rotational delays are dominant, JIT-seek and seek-splitting become more important for reducing the expected waiting time. Virtual IO provides more than an order of magnitude reduction in waiting time when IO requests are large, which is often the case in multimedia systems.

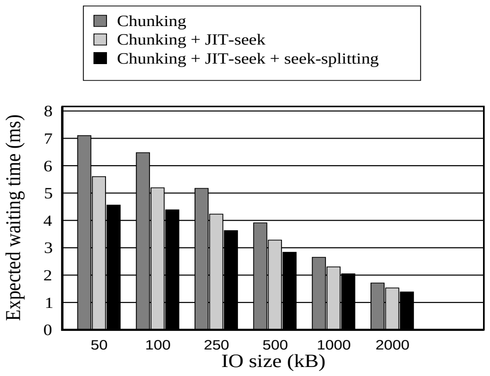


Figure 12: Individual contributions of Virtual IO components on the expected waiting time.

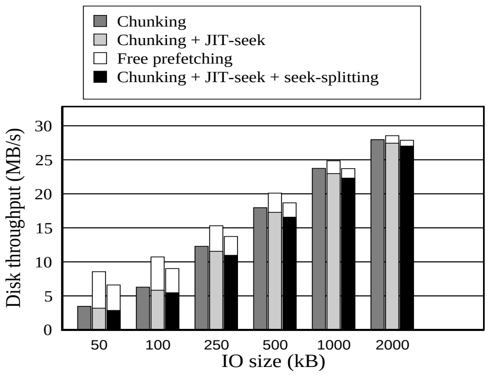


Figure 13: Individual effects of Virtual IO components on disk throughput.

Figure 13 summarizes the individual contributions of the Virtual IO components 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. Our prototype implementation of JIT-seek may introduce overhead when it makes an incorrect 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 of Virtual IO with free prefetching. The free disk throughput acquired using free prefetching depends on the rate of JIT-seeks, which decreases with an increase in IO size. We believe that the free prefetching can be useful for multimedia systems that often access data sequentially.

4.3 Effect of Preemption in Virtual IO

In this section, we aim to answer our third question: What is the effect of IO *preemption* on the average response time for higher-priority requests and the disk throughput? 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 Virtual IO request to service a higher-priority IO request.

Figure 14 presents the response time for a higher-priority request when using Virtual IO in two possible scenarios: (1) when the higher-priority request is serviced after the ongoing IO is completed, and (2) when the ongoing IO is preempted to service the higher-priority IO request. If the ongoing Virtual IO request is not preempted, then all higher-priority requests that arrive while it is being serviced, must wait until the ongoing IO is completed. The results presented in Figure 14 are for $2,000$ kB IO requests in order to emphasize the benefits of IO preemption for systems that require high disk throughput. By preempting the ongoing Virtual IO, the response time for a high priority request reduces by a factor of 4. The maximum response times (not shown) for Virtual IO with and without preemption were measured as 34.6 ms and 150.1 ms respectively. More extensive experimental results are presented in Tables 1 and 3.

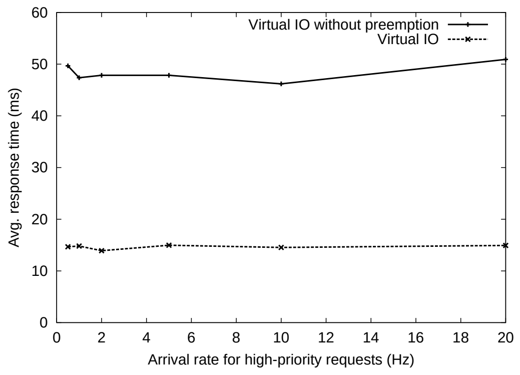


Figure 14: The average response time for higher-priority requests depending on their arrival rate (ν). Ongoing IO requests are $2,000$ kB each.

Preemption of IO requests does not come without costs. Each time a higher-priority request preempts a low-priority Virtual IO request for disk access, an extra seek is required to continue servicing the preempted request after the high-priority request is completed. Figure 15 presents disk throughput in the presence of higher-priority requests with different arrival rates. For applications that require high responsiveness, the performance penalty of IO preemption seems acceptable, since the response time can be substantially reduced.

Table 1 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 is expected since the overhead of IO preemption is an extra seek operation per preemption. Depending on the application, preemption may or may not be desirable.

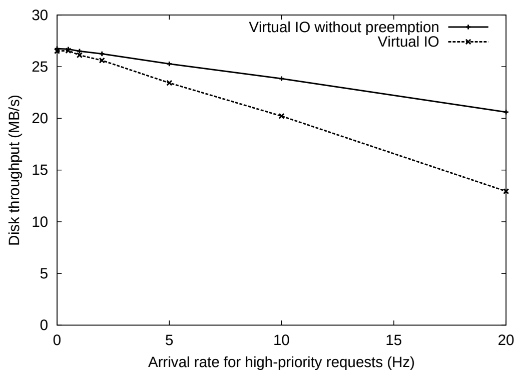


Figure 15: Disk throughput for $2,000$ kB IOs depending on the arrival rate of higher-priority requests (ν).

IO $[kB]$	ν $[Hz]$	*Avg. Resp.* $[ms]$ *npIO*	*vIO*	*Throughput* $[MB/s]$ *npIO*	*vIO*	*vIO+FP*
50	*0.5*	19.2	19.4	3.386	2.833	6.535
50	*1*	21.8	16.0	3.359	2.887	6.671
50	*2*	20.8	17.6	3.319	2.824	6.540
50	*5*	21.0	18.2	3.178	2.623	6.068
50	*10*	21.2	18.3	2.949	2.301	5.375
50	*20*	21.1	18.4	2.488	1.676	3.838
500	*0.5*	29.2	15.7	16.250	16.400	18.481
500	*1*	28.1	15.5	16.145	16.202	18.245
500	*2*	28.2	16.7	15.943	15.774	17.755
500	*5*	28.6	16.0	15.279	14.575	16.420
500	*10*	28.9	16.3	14.239	12.483	14.064
500	*20*	29.4	16.8	11.963	8.566	9.650
$2,000$	*0.5*	54.4	14.0	24.025	26.700	27.526
$2,000$	*1*	58.3	14.7	23.867	26.319	27.144
$2,000$	*2*	55.8	14.3	23.524	25.603	26.411
$2,000$	*5*	55.2	14.7	22.581	23.620	24.363
$2,000$	*10*	56.9	14.4	20.982	20.289	20.916
$2,000$	*20*	55.8	14.7	17.858	13.212	13.626

Table 1: The average response time and disk throughput for non-preemptible IO ($npIO$) and Virtual IO with free prefetching (vIO and $vIO + FP$).

External Waiting Time

In Section 3.1, we introduced the notion of external waiting time to explain the difference in the preemptibility of read and write IO requests. Table 2 summarizes the results of our experiments aimed to find out the effect of external waiting time on the preemption of write IO requests. The arrival rate of higher-priority requests is set to $\nu = 1$ Hz. As shown in Table 2, the average response time for higher-priority requests for write experiments is several times longer than in 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 then explains these seemingly unexpected results.

Table 3 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. Virtual 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

	Exp. Waiting $[ms]$				*Avg. Response* $[ms]$			
IO	*npIO*		*vIO*		*npIO*		*vIO*	
$[kB]$	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

Table 2: The expected waiting time and average response time for non-preemptible and Virtual IO ($\nu = 1$ Hz).

requests in the case of non-preemptible IO, it achieves better disk throughput. For large IO requests, Virtual IO also achieves better write throughput than that of non-preemptible IO because it uses optimal chunk size for data transfer. The effects of the external waiting time is not the focus of this paper, but will be the subject of our future work.

IO	ν	*Avg. Response* $[ms]$		*Throughput* $[MB/s]$	
$[kB]$	$[Hz]$	*npIO*	*vIO*	*npIO*	*vIO*
50	*0.5*	93.1	26.9	4.849	1.980
50	*1*	105.8	24.6	4.746	1.964
50	*2*	91.1	22.7	4.683	1.935
50	*5*	102.2	24.4	4.397	1.843
50	*10*	87.5	23.7	3.947	1.698
50	*20*	81.3	23.3	3.088	1.420
500	*0.5*	32.4	20.3	13.708	11.406
500	*1*	36.0	20.3	13.643	11.237
500	*2*	35.0	20.8	13.450	11.016
500	*5*	34.9	20.5	12.824	10.357
500	*10*	36.6	20.3	11.672	9.132
500	*20*	34.6	20.7	9.643	6.915
$2,000$	*0.5*	79.3	18.2	17.568	19.952
$2,000$	*1*	70.0	18.3	17.681	19.742
$2,000$	*2*	77.2	19.1	17.489	19.319
$2,000$	*5*	73.6	18.9	16.603	17.828
$2,000$	*10*	70.7	19.2	15.364	15.245
$2,000$	*20*	72.5	19.0	12.961	10.165

Table 3: The average response time and disk write throughput for non-preemptible and Virtual IO.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented the design and implementation of Virtual IO, and proposed three techniques— data transfer chunking, just-in-time seek, and seek-splitting. These techniques enable the preemption of a disk IO request, and thus drastically reduce the waiting time for a higher-priority IO request. More significantly, we have shown that an efficient implementation of Virtual IO can achieve this objective with little or no loss in disk throughput. In order to efficiently implement Virtual IO, disk parameters such as optimal chunk size, seek time, and rotational delay must be obtained accurately. We have explained how our disk profiler extracts these parameters, effectively assisting our implementation of Virtual IO. Empirical study conducted on our Virtual IO prototype showed that the expected waiting time of higher-priority IO requests can be reduced by an order of magnitude (from more than tens, or even hundreds, of milliseconds to less than a couple of milliseconds). We believe that delay-sensitive multimedia applications such as virtual reality and interactive games can take advantage of Virtual IO to improve the quality of service significantly.

We plan to further our research in three directions. First, we plan to investigate how preemptible Virtual IO can be used to improve disk scheduling algorithms for multimedia applications. Second, believing that Virtual IO can benefit from a more detailed profiling of the disk write buffer, we will continue to improve the preemptibility of write disk access. Additionally, we plan to investigate the benefits of free prefetching for different classes of multimedia applications.

6. REFERENCES

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

[2] 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.

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

[4] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Virtual IO: Preemptible disk access. *http://www.cs.ucsb.edu/∼zoran/papers/vio02x.pdf*, April 2002.

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

[6] S. Ghandeharizadeh, A. Dashti, and C. Shahabi. A pipelining mechanism to minimize the latency time in hierarchical multimedia storage managers. *Computer Communication*, 18(3):170–184, March 1995.

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

[8] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. *Usenix Conference on File and Storage Technologies*, January 2002.

[9] A. Molano, K. Juvva, and R. Rajkumar. Guaranteeing timing constraints for disk accesses in rt-mach. *Real Time Systems Symposium*, 1997.

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

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

[12] 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.

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

[14] P. J. Shenoy and H. M. Vin. Efficient support for interactive operations in multi-resolution video servers. *ACM Multimedia Systems*, 7(3), May 1999.

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

[16] 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.

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

[18] 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.

[19] 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 Operating Systems Design and Implementation*, October 2000.