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 data) is around 15 milliseconds [1]. Such interactive requests can be modeled as higher-priority IO requests. However, the disk might already be servicing an IO 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 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*, $T_{waiting}$, for an IO request is defined as the amount of time it must wait, due to the non-preemptibility of the ongoing IO, 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 [12]. 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 into fine-grained physical disk-commands and enables IO preemption between any two disk commands. If a higher-priority IO request can arrive at any time during the service time for the ongoing IO request with equal probability, then $T_{waiting}$ of the higher-priority request can include seek time (T_{seek}), rotational delay (T_{rot}), and data transfer time ($T_{transfer}$) for the ongoing IO request, with the expected value of

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

Virtual IO maps each IO request into multiple fast-executing disk commands using three methods. The ongoing IO 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 $\mathbf{T_{transfer}}$.** Virtual IO divides a large IO transfer into a number of small chunk transfers, and makes preemption possible between the chunk transfers. If the IO is not preempted, chunking does not incur any overhead. This is due to the prefetching mechanism in current disk drives (Section 2.1).
- **Preempting $\mathbf{T_{rot}}$.** Virtual IO performs just-in-time (JIT) seek, which converts the rotational delay at the destination track into a fully preemptible pre-seek slack. This slack can also be used to perform prefetching for the ongoing IO request, or/and to mask seek splitting overhead (Section 2.2).
- **Splitting $\mathbf{T_{seek}}$.** Virtual IO splits a long seek into sub-seeks, and permits preemption between two sub-seeks (Section 2.3).

Virtual IO services a single IO request using multiple disk commands. Let the time required to execute i^{th} disk command be T_i. Let T_{idle} be the idle time before JIT-seek. The expected waiting time[1] using Virtual IO can then be expressed as

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

[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.

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 the seek operation into two sub-seeks, each of $10,000$ cylinders, requiring $T'_{seek} = 9$ ms each. In this case, the seek splitting

[1]Please refer to Section 2 for the derivation of this equation.

does not cause extra overhead because the $T_{rot} = 7$ ms can mask the 4 ms increased total seek time ($2 \times T'_{seek} - T_{seek} = 2 \times 9 - 14 = 4$ ms) incurred by seek splitting. The rotational delay is now $T'_{rot} = T_{rot} - (2 \times T'_{seek} - T_{seek}) = 3$ ms.

With this $T'_{rot} = 3$ ms knowledge, Virtual IO can wait for 3 ms before performing a JIT-seek. This JIT-seek method makes T'_{rot} preemptible. 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 can be serviced immediately after each disk-command. Virtual IO thus makes preemptible the originally non-preemptible read IO request. 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 2.1 and 2.4) nor additional disk latency.

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, the longest stall for the higher-priority request during data-transfer is just 1 ms. Thus, 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.

This example shows that Virtual IO drastically reduces the expected waiting time[2]. 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 with little or no loss in disk throughput.
- We show a feasible path to implement Virtual IO. We explain how the implementation of Virtual IO is made possible through Diskbench, our disk profiling tool [5].

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

2. VIRTUAL IO

Before introducing the concept of *Virtual IO*, we first define some terms which we will use throughout the rest of this paper. A *logical disk block* is the smallest unit of data that can be accessed on a disk drive (typically 512 B). Each logical block resides at a physical disk location, depicted by a physical address (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* ($T_{waiting}$) 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.

In order to understand the magnitude of the waiting time, let us consider a typical read IO request. 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 $T_{transfer}$. 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 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 access times required for ongoing IO as

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

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} 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. Let T_{idle} be the idle time before JIT-seek. 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 $p_i = \frac{T_i}{\sum T_i + T_{idle}}$. Finally, the expected waiting time of a higher-priority request in Virtual IO can be expressed as

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

2.1 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., for multimedia). A 500 kB IO requires 25 ms at a data transfer rate of 20 MBps. To make the $T_{transfer}$ preemptible, Virtual IO uses *chunking*.

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

Benefits: Chunking reduces the transfer component of $T_{waiting}$. A higher-priority request can be serviced after a chunk transfer is completed instead of having to wait for the entire IO to complete. 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: Chuck size must be carefully chosen or disk throughput may degrade. We show in Section 2.4 that an optimal range of chunk sizes can be obtained by our disk profiler, in a disk-dependent way.

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

The rotational period 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.

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

Benefits:

1. The slack time between two IO requests is fully preemptible. For example, if an IO request must incur a T_{rot} of 5 ms and JIT-seek delays the issuing of the IO by 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 stream (*free preefetching*).

3. The slack can also be used to mask the overhead incurred in performing *seek-splitting*, which we shall discuss in next section.

[2]Virtual IO increases the preemptibility of disk access with little or no overhead. However, if an IO is preempted, an extra seek operation may be required to resume service for the preempted IO.

Overhead: Virtual IO predicts T_{rot} and T_{seek} between two IO operations in order to perform JIT-seek. If there is an error in prediction, then the penalty can be one extra disk rotation.

2.3 Seek Splitting: Preempting T_{seek}

The seek time (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 2.3: *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.

Overhead: 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 can degrade disk throughput. However, when the obtained slack in JIT-seek is large, it can be used to mask the seek overhead.

2.4 Disk Profiling

Virtual IO greatly relies on disk profiling to obtain accurate disk parameters. The extraction of these disk mappings is described in [5]. The disk profiler runs once before Virtual IO is used for the first time to obtain the following required disk parameters: *disk block mappings* (both logical-to-physical and physical-to-logical block address transformation), *the optimal chunk size* (required to efficiently perform chunking), *disk rotational factors* (rotation period and rotational skew factors for disk tracks), and *seek curve* (required for accurate JIT-seek and seek-splitting).

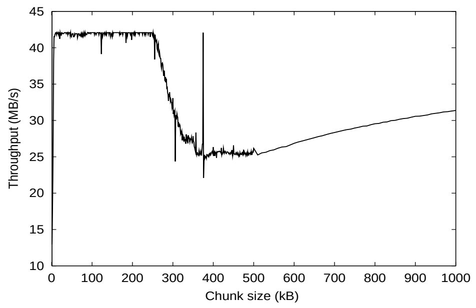


Figure 1: Sequential throughput (SCSI ST318437LW).

The disk profiler provides Virtual IO the optimal range for the chunk size. Figure 1 depicts the effect of chunk size on the sequential read throughput for one SCSI disk drive. On y-axis we show achieved sequential disk throughput for the read operation using the chunk size of x. For this particular disk, the range for the optimal chunk size is between 6 kB and 263 kB, and can be automatically extracted. Our disk profiler implementation was successful in extracting the optimal chunk size for several SCSI and IDE disk drives with which we experimented [4].

3. EXPERIMENTAL RESULTS

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. All experiments were performed on a Seagate ST318437LW SCSI disk. The prototype uses Diskbench [5] to profile the disk. For performance benchmarking, we use equal-sized IO requests at random positions on the disk. The Virtual IO prototype services a non-preemptible IO request using the smallest number of disk commands that provide the optimal disk throughput. Based on the disk profiling, chunking in Virtual IO divides the data transfer into 25 kB chunks, except for the last chunk, which can be smaller. JIT-seek uses an offset of 1 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.

3.1 Preemptibility of Virtual IO

In this section, we aim to answer *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.

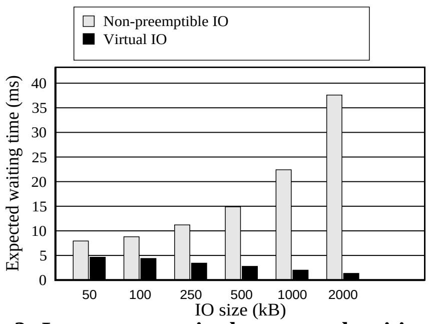


Figure 2: Improvements in the expected waiting time.

Figure 2 depicts the difference in the expected waiting time between non-preemptible IO and Virtual IO (calculated using Equations 1 and 2). 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 increases due to the larger data transfer time for larger IO requests. However, the expected waiting time in Virtual IO actually decreases for large IOs, since a disk spends more time in data transfer, which has a higher preemptibility. Virtual IO can reduce the expected waiting time by more than an order of magnitude in systems with large IOs. Figure 3 shows that Virtual IO provides IO preemptibility with little loss in disk throughput.

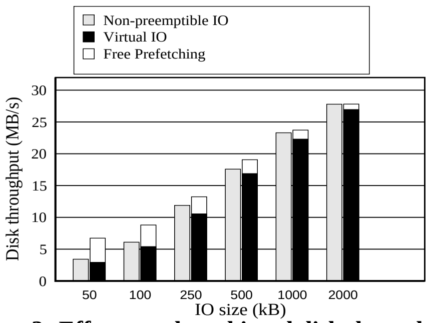
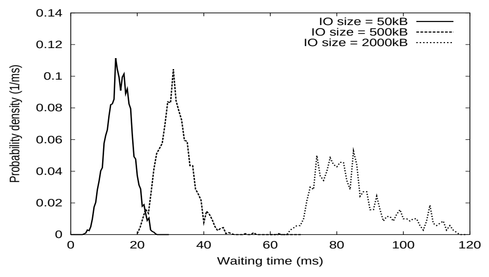
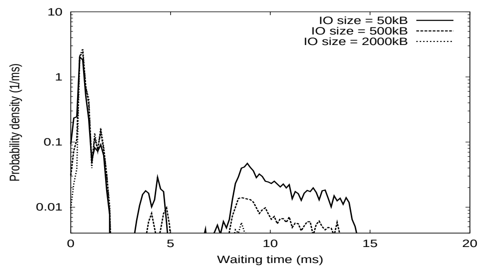


Figure 3: Effects on the achieved disk throughput.

We use the duration of disk commands to measure the preemptibility of the disk access, a smaller value implying a more preemptible system. Figure 4 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 4a), 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. In the case of Virtual IO (Figure 4b), the distribution does not depend on the IO request size, but on individual disk commands used to perform an IO request. For detailed study please refer to [4].



(a) Non-preemptible IO (linear scale)



(b) Virtual IO (logaritmic scale)

Figure 4: Distribution of disk commands duration.

3.2 Individual Contributions within Virtual IO

In this section, we aim to answer *what are the individual contributions of the three components of Virtual IO*. Figure 5 shows the individual contributions of the three Virtual IO components with respect to expected waiting time. The data transfer in Virtual IO is highly preemptible, and the expected waiting time decreases as the duration of the transfer increases. When the transfer component dominates the seek and rotational components, chunking is the most important method for reducing the expected waiting time. Otherwise, JIT-seek and seek-splitting are more important. 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 6 summarizes the individual contributions of the Virtual IO components with respect to the achieved disk throughput.

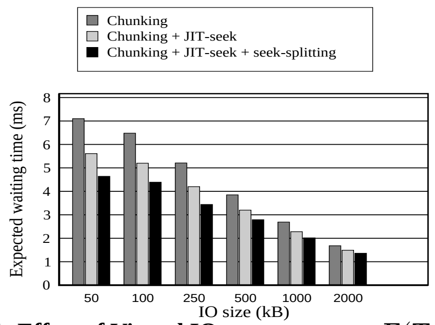


Figure 5: Effect of Virtual IO components on $\mathbf{E(T_{waiting})}$.

4. RELATED WORK

In the past, little need has been expressed for highly preemptible disk access. Before the pioneering work of [3, 7], 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 of the waiting time. A minimum chunk size of one track was proposed. 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. Even for the data transfer component, we show that the bounds for zero-overhead preemptibility proposed in [3] are too tight.

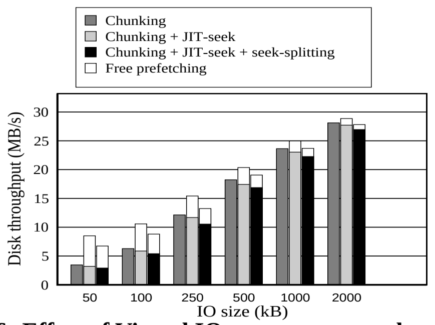


Figure 6: Effect of Virtual IO components on throughput.

There is a large body of literature proposing IO scheduling policies for multimedia and real-time systems that improve disk response time [11, 10, 2, 9, 6, 8]. 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.

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 with little or no loss in disk throughput. 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 two directions. First, we plan to investigate how preemptible Virtual IO can be used to improve disk scheduling algorithms for multimedia applications. Second, we plan to implement Virtual IO in Linux kernel.

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

[4] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Design, analysis, and implementation of Virtual IO. *http://www.cs.ucsb.edu/∼zoran/papers/vio02.pdf*, September 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] 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.

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

[8] 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.

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

[10] 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*, 1997.

[11] 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.

[12] 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.