Implementing Range Writes in DiskSim Simulator

Zhenxiao Luo
Department of Computer Sciences, University of Wisconsin Madison
zhenxiao@cs.wisc.edu

ABSTRACT

Writing to disk in a rotationally-optimal manner is challenging. Much of the problem arises from the traditional interface, which demands an exact address for each write. In this project, I change this traditional interface by building what we call *Range Writes*. A range write takes a data block and a list of possible destination addresses; the disk then internally chooses the best possible address and writes the data to it, returning the address to the caller when finished. In this project, I explore range writes via a system level storage subsystem simulator – *DiskSim*. I introduce two types of *Range Writes*: *Continuous Range Writes* and *Discrete Range Writes*. I also implement *Anticipatory Scheduling* in DiskSim. Experimental results show that, with my manually generated workload, *Range Write* always outperforms Common Write, and with carefully chosen parameters, *Range Write* could achieve more than 40% improvement over Common Write.

Keywords

Range Writes, Rotational Latency, DiskSim Simulator, Anticipatory Scheduling

1. INTRODUCTION

The performance of the input/output (I/O) subsystem plays a large role in determining overall system performance in many environments. The relative importance of this role has increased steadily for the past years and should continue to do so for two reasons. First, the components that comprise the I/O subsystem have improved at a much slower rate than other system components. For example, microprocessor performance grows at a rate of 35-50 percent per year[3], while disk drive performance grows at only 5-20 percent per year[6]. As this trend continues, applications that utilize any quantity of I/O will become more and more limited by the I/O subsystem[1]. Second, advances in technology enable new applications and expansions of existing applications, many of which rely on increased I/O capability.

In response to the growing importance of I/O subsystem performance, a number of research are focusing on optimizing Disk Access. Typically, *Disk Access Time* consists of three components: *Seek Time*, *Rotational Latency*, and *Transfer Time*. In my project, I focus on minimizing Rotational Latency in Disk Write.

A common disk write has the interface:

$$write\ data\ address$$

Using this interface, lots of time is spent on positioning disk head to the exact address.

The basic idea of *Range Writes* is to make a new interface for disk write:

$$write\ data\ \{address\ list\}$$

Using this new interface, disk head could internally chooses the best possible address from address list and writes the data to it. In this way, Rotational Latency will be reduced. In my project, I implement two types of *Range Writes*, say, *Continuous Range Writes* and *Discrete Range Writes*. Experimental results show that, with my manually generated workload, both of these two types of *Range Writes* outperform the common write interface. When the range varies from 0 to one track, disk write benefits significantly and at last its positioning time reached 0.

When incorporating *Range Writes* into the *DiskSim*[4] simulator, I implemented *Anticiparoty Scheduling*[5]: Before choosing a Disk Access for service, the scheduler introduces a short, controlled delay period, during which it waits for additional Disk Accesses to arrive. In the experiment, I varied the delay, and found the optimal delay that could minimize Rotational Latency. Results show that with *Anticipatory Scheduling*, *Range Writes*'s performance could be further improved by about 40%.

Section 2 is related work. After an exposition of the new Disk Write interface in Section 3, I describe the implementation issues and *Anticipatory Scheduling* in Section 4, and delve into a detailed experimental evaluation in Section 5. I will talk about my conclusion and experience in Section 6.

2. RELATED WORK

A number of previous storage systems were designed to optimize disk access performance. MimdRAID[10] is a prototype which uses disk head position prediction to increase the

performance of a disk array.

In MimdRAID[10], Xiang Yu et al. proposes an analytical model that can guide a designer how to design disk arrays that can flexibly and systematically reduce seek and rotational delay in a balanced manner. Given a fixed budget of disks, MimdRAID could intelligently choose a combination of techniques(striping, mirroring, and rotational data replication) to have best performance.

MimdRAID mainly focuses on how to employ performance-enhancing techniques in a large disk array scenario. It does nothing with the traditional write interface. Using the traditional interface, performance improvement is rather expensive. Usually, we have to pay lots of extra disk space for striping and replication.

Dishon and Liu[2] consider latency reduction on either synchronized or unsynchronized D-way mirrors. A synchronized mirror can reduce foreground propagation latency because the multiple copies can be written at nearly the same time if we insist that the replicas are placed at rotationally identical positions.

The *distorted mirror*[7] provided an alternative way of improving the performance of writes in a mirror. It performed writes initially to rotationally optimal but variable locations and propagated them to fixed locations later.

Both [2] and *distorted mirror* focus on rotational latency reduction, but they does not change the traditional write interface. Their performance advantage comes at the cost of extra disk space for mirroring.

Disk Mimic[8] proposed a simple table-based approach, using a shortest-mimicked-time-first(SMTF) for I/O scheduling that performs on-line simulation of the underlying disk. To reduce the prohibitively large input space, Disk Mimic used two input parameters, say, the logical distance tetween two requests and the request type, to predict the positioning time. At the first step of my project, I read this paper and got the fundamental knowledge for disk simulation.

In [4], Greg Ganger et al. defines three distinct classes of request criticality and points out the weakness of conventional disk simulator. In order to simulate disk system accurately, simulator must have all the modules of a computer system and individual request response time could affect system behavior. DiskSim[4] is implemented with these guidance. Unlike the conventional approach which evaluate the performance of a subsystem based on standalone subsystem models, DiskSim is a system level disk simulator which includes modules that simulate disks, intermediate controllers, buses, device drivers, request schedulers, disk block caches and disk array data organizations. My project is to implement *Range Writes* in DiskSim simulator. I will introduce its internal structure in the experiment section.

Anticipatory Scheduling[5] makes the assumption that there is likely to be locality in a stream of requests from a given process. It suggests that instead of choosing a request for service, the scheduler could introduce a short, controlled delay period, during which it could waits for additional requests to arrive. I implement *Anticipatory Scheduling* when I incorporate *Range Writes* in DiskSim.

3. RANGE WRITES

Traditionally, disk write has the interface:

$$write\ \ data\ \ address$$

Using this interface, lots of time is spent on positioning disk head to the exact address.

The basic idea of *Range Writes* is to make a new interface for disk write:

$$write\ \ data\ \ \{address\ \ list\}$$

Using this new interface, disk head could internally chooses the best possible address from address list and writes the data to it. In this way, Rotational Latency will be reduced. In my project, I implement two types of *Range Writes*, say, *Continuous Range Writes* and *Discrete Range Writes*.

3.1 Continuous Range Writes

Continuous Range Writes has continuous addresses in {address list}, which starts with *startBlk*(start block number) and ends with *endBlk*(end block number). Using this interface, a disk write could write to any of the blocks that lie within range: [startBlk .. endBlk].

The write interface is:

$$write\ \ data\ \ [startBlk\ ..\ endBlk]$$

If the head position(head block number) of the current disk is *headBlk*. Using this new interface, instead of moving to an exact block number, the head could move to the nearest block number in order to reduce rotational latency. Take *optimalBlk* as the nearest block number:

$$optimalBlk = \begin{cases} startBlk & \text{if (headBlk} < \text{startBlk)} \\ headBlk & \text{if (startBlk} <= \text{headBlk} <= \text{endBlk)} \\ startBlk & \text{if (headBlk} > \text{endBlk)} \end{cases}$$

Note: Since disk could only rotate in one direction, when headBlk > endBlk, disk head could not rotate backward to endBlk(though endBlk has the minimum disktance with disk head). It has to rotate forward and move to headBlk.

3.2 Discrete Range Writes

Discrete Range Writes has discrete addresses in {address list}.

The write interface is:

$$write\ \ data\ \ \{addrBlk1,\ addrBlk2\ ..\ addrBlkN\}$$

Using this interface, a disk write could write to any of the blocks that lie in the address list {addrBlk1, addrBlk2 .. addrBlkN}.

Definition 1. If there are M blocks in a track, and N addresses in {address list}: $\{addrBlk_1,\ \ addrBlk_2\ \ ..$

$addrBlk_N\}$, then by d_i we mean the disktance between $address_i$ and $address_{i+1}$:

$$d_i = \begin{cases} |addrBlk_{i+1} - addrBlk_i| & \text{if (1 <= i <= N - 1)} \\ |addrBlk_1 - addrBlk_N| & \text{if (i = N)} \end{cases}$$

And,

$$\sum_{i=1}^{N} d_i = M$$

If the head position of a disk is random, we have the following theorem:

THEOREM 1. *A disk write has the minimum rotational delay if and only if:*

$$d_1 = d_2 = d_3 = \cdots = d_N$$

PROOF. Since the head position of a disk is random, *headBlk* could be any number within the range [1 .. M]. The probability of *headBlk* being $i(1 <= i <= M)$ is: $\frac{1}{M}$.

If the minimum address that is greater than *headBlk* is $address_k$, then the rotational delay for this head position is:

$$r_k = |address_k - headBlk|$$

Taking all the possible *headBlk* together, the rotational delay is:

$$\begin{aligned} rotational\ \ delay &= \sum_{k=1}^{M} \frac{r_k}{M} \\ &= \frac{1}{M} \sum_{i=1}^{N} \sum_{j=1}^{d_i} (d_i - 1) \\ &= \frac{1}{2M} \sum_{i=1}^{N} d_i(d_i - 1) \\ &= \frac{1}{2M} (\sum_{i=1}^{N} d_i^2 - M) \\ &>= \frac{1}{4M} (\sum_{i=1}^{N} d_i)^2 - \frac{1}{2} \qquad (1) \\ &= \frac{1}{4M} M^2 - \frac{1}{2} \\ &= \frac{M}{4} - \frac{1}{2} \end{aligned}$$

For any positive numbers, their *Quadratic Mean* is no less than *Arithmetic Mean*. The two means are equal if and only if all the numbers are of the same value. So, for equation (1), rotational delay reaches its minimum if and only if:

$$d_1 = d_2 = d_3 = \cdots = d_N$$

This proves the *Theorem*. □

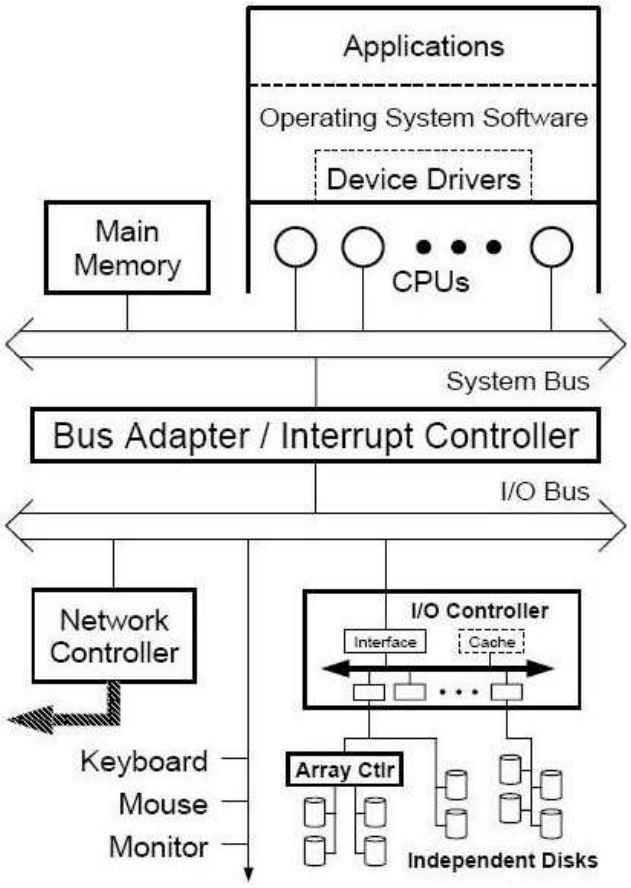

Figure 1: Block Diagram of DiskSim

From theorem 1, we could see that we only need to specify a start address($startAddr$) and an address number(N) for *Discrete Range Writes*. Since the distance between each neighbouring address is the same: $\frac{M}{N}$(there are M blocks in a track). Knowing these two values, disk could internally set the address list.

The new interface is:

$$write\ \ data\ \ startAddr\ \ addrNumber$$

For example, if $addrNumber = 3$, disk could set the three address as following:

$$address_i = \begin{cases} startAddr & \text{if (i = 1)} \\ startAddr + \frac{M}{3} & \text{if (i = 2)} \\ startAddr + \frac{2M}{3} & \text{if (i = 3)} \end{cases}$$

4. IMPLEMENTATION

I implemented *Range Writes* in DiskSim simulator. DiskSim simulator is a system level storage subsystem simulator. Rather than focusing on the storage subsystem in a vacuum, DiskSim is expanded to include all major system components, so as to incorporate the complex performance/workload feedback effects.

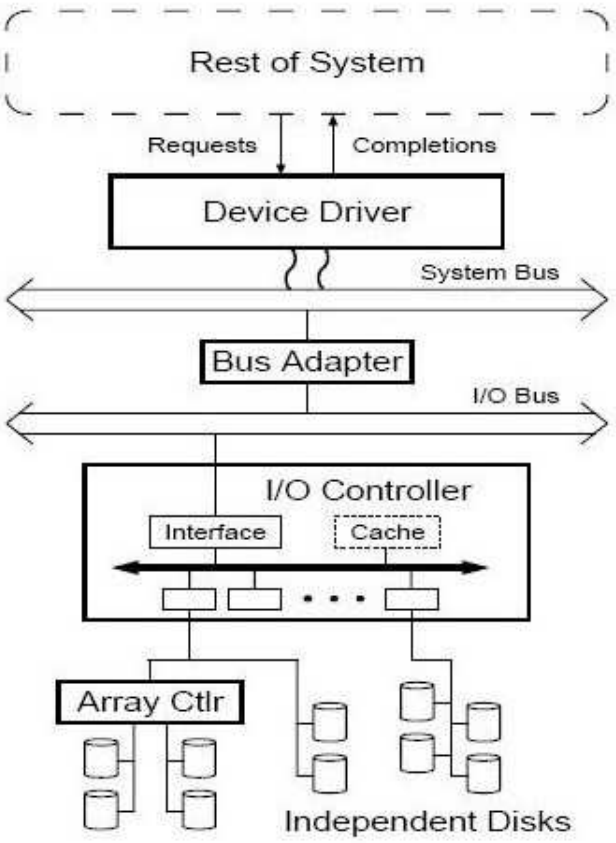

Figure 2: Block Diagram of Storage Subsystem

4.1 DiskSim

DiskSim consists of modules for each major system component and interfaces to the outside world(e.g., users and other systems). Processes execute within a system-level simulator in a manner that imitates the behavior of the corresponding system. Also, external interrupts may arrive at the interfaces, triggering additional work for the system.

Figure 1 shows DiskSim simulator's structure. It includes processors, main memory, applications, operating system software, buses, a bus adapter that also handles interrupt control activities, storage sybsystem components and some other I/O devices(network and user interface). The applications are the inputs to the simulator(together with externally generated interrupts and data), dictating the tasks performed. An application consists of one or more processes. In DiskSim, process refers to any instruction stream other than an interrupt service routine, independent of the virtual memory context. So, multiple threads that share a common context are each processes. System call and exception service routines are part of the process, because they are generally executed within the context and flow of the process. A process is modeled as a sequence of events separated by computation times. An interrupt controller tracks the pending interrupts in the system and routes them to the CPUs for service. The state of the interrupt controller is updated when new interrupts are generated and when a CPU begins handling an interrupt.

4.2 Request Queue Scheduler

In DiskSim, request queues and the corresponding schedulers can be present in several different storage subsystem components (e.g., device drivers, intelligent controllers and disk drives), so the request queue scheduler functionality is implemented as a separate module that is incorporated into various components as appropriate. New requests are referred to the queue module by queue-containing components. When such a component is ready to initiate an access, it calls the queue module, which selects (i.e., schedules) one of the pending accesses according to the configured policies. When an access completes, the component informs the queue module. In response, the queue module returns a list of requests that are now complete. This list may contain multiple requests because some scheduling policies combine sequential requests into a single larger storage access. The queue module collects a variety of useful statistics (e.g., response times, service times, inter-arrival times, idle times, request sizes and queue lengths), obviating the need to replicate such collection at each component.

4.3 Storage Subsystem

Figure 2 shows the structure of DiskSim's Disk module. The disk module includes disk drives, a small disk array, and intelligent cached I/O controller, a simple bus adapter, several buses, a device driver and an interface to the rest of the system. Requests are issued to the disk module via the interface and are serviced in a manner that imitates the behavior of the corresponding storage subsystem.

The interface between the storage subsystem and the remainder of the system is very simple. Requests are issued by the system and completion is reported for each request when appropriate. A request is defined by five values:

- **Device Number**: the logical storage device to be accessed. The device number is from the system's viewpoint and may be remapped several times by different components as it is routed to the final physical storage device. This field is unnecessary if there is only one device.
- **Starting Block Number**: the logical starting address to be accessed. The starting block number is from the system's viewpoint and may be remapped several times by different components as it is routed to the final physical storage device.
- **Size**: the number of bytes to be accessed.
- **Flags**: control bits that define the type of access requested and related characteristics. The most important request flag component indicates whether the request is a read or a write. Other possible components might indicate whether written data should be re-read (to verify correctness), whether a process will immediately block and wait for the request to complete, and whether completion can be reported before newly written data are safely in non-volatile storage.
- **Main Memory Address**: the physical starting address in main memory acting as the destination (or

source). The main memory address may be represented by a vector of memory regions in systems that support gather/scatter I/O. This field is often not included in request traces and is only useful for extremely detailed simulators.

4.4 Continuous Range Writes Implementation

To implement *Continuous Range Writes* in DiskSim, I add two more values to the request interface:

- **Ending Block Number**: the logical ending address to be accessed for continues range writes. Having the Ending Block Number, request could write to any block that falls in range [Starting Block Number .. Ending Block Number].
- **Range Write Signal**: a control bit that define whether the current request is a range write.

4.5 Discrete Range Writes Inplementation

To implement *Continuous Range Writes* in DiskSim, I add two more values to the request interface:

- **Address Number**: The number of addresses in {address list}. Having the Address Number, disk will internally set the appropriate address for *Continuous Range Writes*(as discussed in section3.2).
- **Range Write Signal**: a control bit that define whether the current request is a range write.

4.6 Anticipatory Scheduling

To further improve the performance of *Range Writes*, I implement *Anticipatory Scheduling* in DiskSim's *Request Queue Scheduler*. Before choosing a request for service, the *Request Queue Scheduler* waits for additional requests to arrive. After waiting, it chooses the request that has the minimum rotational delay to service.

I implement two types of *Anticipatory Scheduling* in my project:

- **Think Time**: Before selecting the next request, the scheduler will think for some time(*Think Time*), and then choose the request with minimum rotational delay from the queue. Within *Think Time*, a number of requests may arrive. Selecting from a larger number of requests will definitely increase the possibility of choosing a request with less rotational delay.
- **Wait Number**: Before selecting the next request, the scheduler will wait for a number of requests to arrive(*Wait Number*), and then choose the request with minimum rotational delay from the queue. The increased number of requests will definitely increase the possibility of choosing a request with less rotational delay.

5. EXPERIMENTAL RESULTS

This section evluates *Range Writes*'s performance on DiskSim simulator. Firstly, I vary the range and show the performance of *Continuous Range Writes*. Secondly, I vary the address number and show the performance of *Discrete Range Writes*. Finally, I implement the two types of *Anticipatory Scheduling* on both types of *Range Writes* and show the performance improvement.

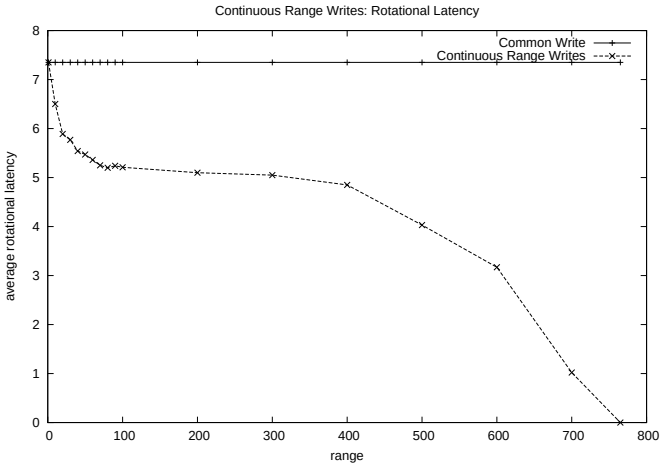

Figure 3: Continuous Range Writes: Rotational Latency

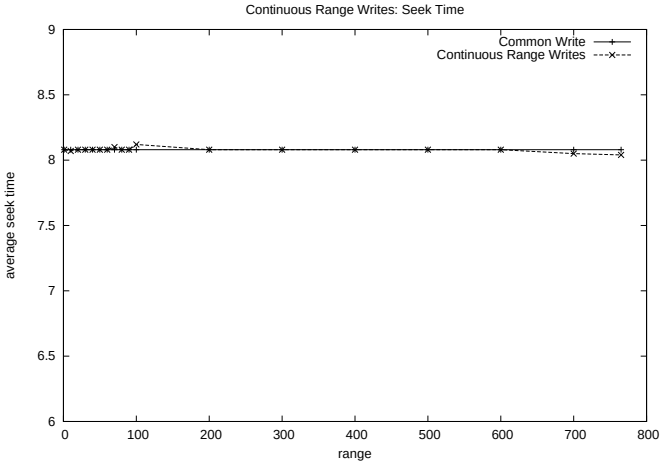

Figure 4: Continuous Range Writes: Seek Time

In DiskSim, each track has 765 blocks. The workload for my experiment is 30 manually generated disk writes, with their *Starting Block Number* spans from 0 to 730. Since *Range Writes* does not change *Transfer Time*, I simply set all the writes' *Size* equal to 10 blocks. All experimental results measuring rotational latency and seek time are reported in millisenconds.

5.1 Continuous Range Writes: Varying Range

In this experiment, I vary the range for *Continuous Range Writes*, and collect the 30 writes' average rotational latency and seek time. When range varies from 1 block to 100 blocks, I set the step to 10 blocks. When range varies from 100 blocks to 700 blocks, I set the step to 100 blocks. Finally, I collect data when range comes to the whole track(there are 756 blocks in a track).

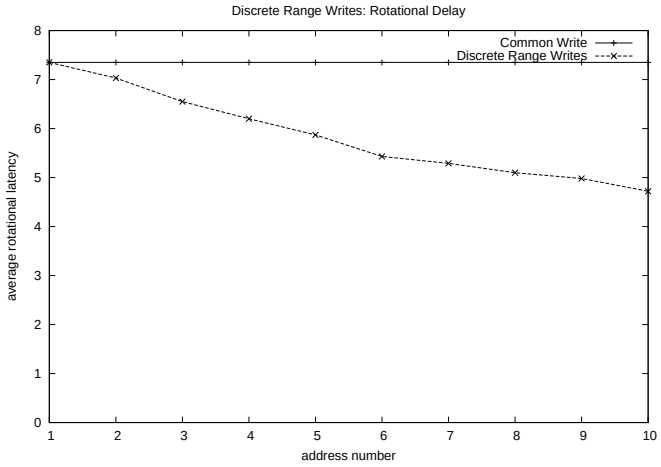

Figure 5: Discrete Range Writes: Rotational Latency

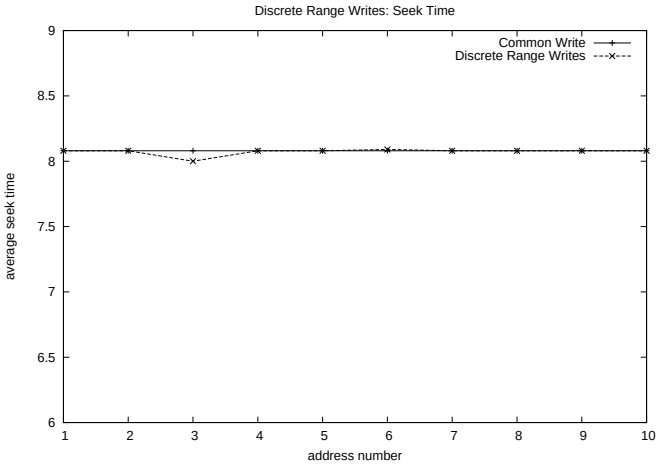

Figure 6: Discrete Range Writes: Seek Time

Figure 3 compares the rotational latency of *Continuous Range Writes* and that of *Common Writes* as range becoming larger and larger. Intuitively, *Continuous Range Writes* should have better performance with larger ranges. Since with a larger range, the possibility that disk head has a smaller rotational latency will be increased.

The results show that when range varies from 1 block to 100 blocks, rotational latency decreases rapidly. While, after 100 blocks, rotational latency will decrease at a much slower speed untill 600 blocks, and then it will decrease sharply towards 0. Here we see *Continuous Range Writes* outperforms *Common Writes*, and their performance difference increases as range increases. The results match our intuition.

The results also implies that for small ranges(range less than 100 blocks) and big ranges(range bigger than 600 blocks), a small range increase will cause a big difference in rotational latency. While, for medium ranges(between 100 blocks and 600 blocks), rotational latency does not change much when range increases.

Figure 4 illustrates that, with *Continuous Range Writes*, the seek time of disk writes does not change. This is exactly what we expected, for *Range Writes* only focusing on reducing rotational latency.

5.2 Discrete Range Writes: Varying Address Number

In this experiment, I vary the address number for *Discrete Range Writes*, and collect the 30 writes' average rotational latency and seek time. When address number increases from 1 to 10, I collect data for each address number.

Figure 5 compares the rotational latency of *Discrete Range Writes* and that of *Common Writes* as adress number becoming larger and larger. Intuitively, *Discrete Range Writes* should have better performance with larger address numbers. Since having a larger address number, it is easier for disk head to find a nearer address to write.

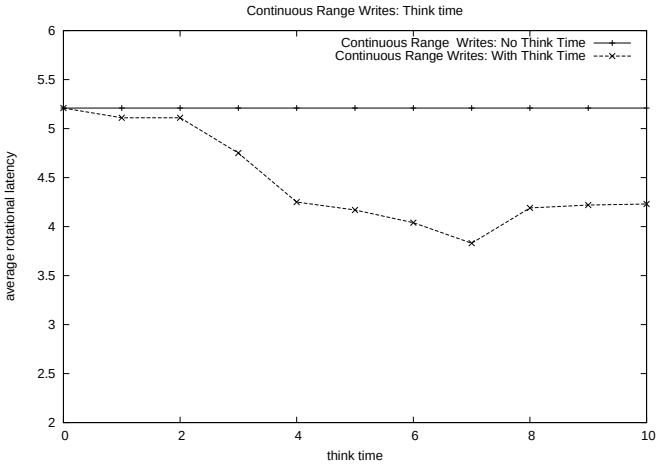

Figure 7: Continuous Range Writes: Think Time

The results show that when address number varies from 1 to 10, rotational latency decreases almost at the same speed. As we expected, figure 6 illustrates that, with *Discrete Range Writes*, the seek time of disk writes does not change.

Practically, I think *Discrete Range Writes* is better. Although *Continuous Range Writes* could achieve smaller rotational latency, it is highly impossible for a disk to have continuous free space for disk write. While, *Discrete Range Writes* could make good use of disk fragments, which is more economical.

5.3 Anticipatory Scheduling: Think Time

In this experiment, I try one type of *Anticipatory Scheduling*, *Think Time*, on both *Continuous Range Writes* and *Discrete Range Writes*. For *Continuous Range Writes*, I use the result with range 100 as baseline. The reason why I choose range 100 is that the rotational latency will stay statically and decrease at a smaller speed between range 100 and range 600. For *Discrete Range Writes*, I use the result with address number 3 as baseline. This is just for simplicity. For each experiment, I vary the think time from 1 millisecond to

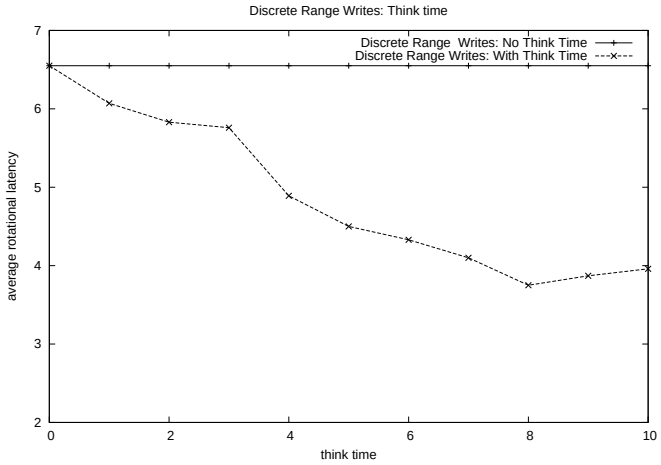

Figure 8: Discrete Range Writes: Think Time

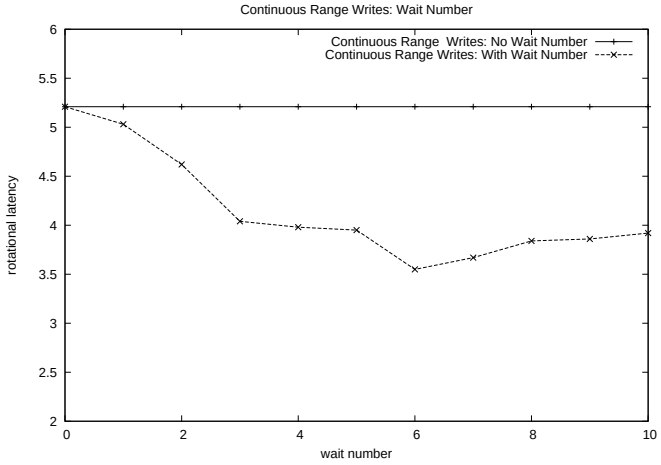

Figure 9: Continuous Range Writes: Wait Time

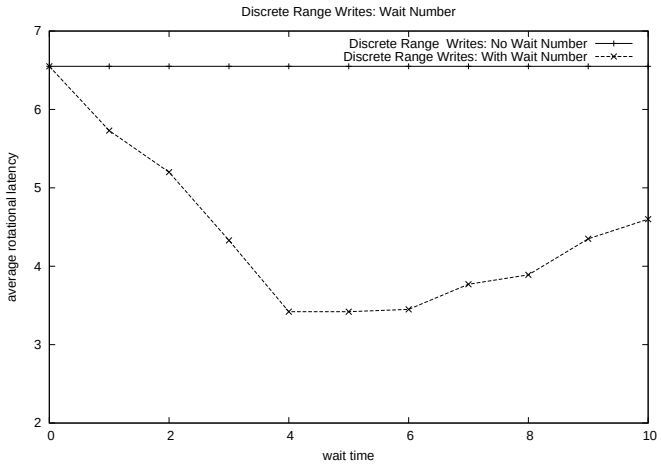

Figure 10: Discrete Range Writes: Wait Time

10 milliseconds, and collect the 30 writes' average rotational latency.

Figure 7 compares the rotational latency of *Continuous Range Writes* without *Think Time* and that with *Think Time*. From the results, we could see that when think time increases from 0 milliseconds to 2 milliseconds, rotational latency for the two are almost the same. While after 2 milliseconds, the rotational latency with *Think Time* will decrease rapidly. And when *Think Time* reaches 7 milliseconds, *Continuous Range Writes* with *Think Time* gets its minimum rotational latency. After 7 milliseconds, surprisingly, its rotational latency will increase. This implies that waiting for a longer time does not always guarantee better performance. Sometimes, the scheduler may switch frequently, which will increase its rotational latency. While, in general, we could see that *Anticipatory Scheduling* is effective for *Continuous Range Writes*. It could improve its performance by more than 40 percentage.

Figure 8 compares the rotational latency of *Discrete Range Writes* without *Think Time* and that with *Think Time*. Here we could see that rotational latency decreases at a constant speed when *Think Time* varies from 1 millisecond to 8 milliseconds. And after than, it will remain the same. Compared with figure 7, I think *Anticipatory Scheduling* is more effective for *Discrete Range Writes*. With *Anticipatory Scheduling*, *Discrete Range Writes* decreases at a larger speed and it could improve its performance by almost 40 percentage.

5.4 Anticipatory Scheduling: Wait Number

In this experiment, I try another type of *Anticipatory Scheduling*, *Wait Number*, on both *Continuous Range Writes* and *Discrete Range Writes*. I use the same baseline as that of *Think Time*. For each experiment, I vary the wait number from 0 to 10, and collect the 30 writes' average rotational latency.

Figure 9 compares the rotational latency of *Continuous Range Writes* without *Wait Number* and that with *Wait Number*. From the results, we could see that rotational latency decrease rapidly when *Wait Number* varies from 0 to 3. After 3, rotational latency will remain at a constant level. This implies that if we wait for more than three additional requests, the performance will be almost the same.

Figure 10 compares the rotational latency of *Discrete Range Writes* without *Wait Number* and that with *Wait Number*. From the results, we could see that rotational latency decrease rapidly when *Wait Number* varies from 0 to 4. When *Wait Number* varies from 4 to 6, rotational latency will remain at a constant level. Then it will increase a little bit. This implies that if we wait for more than six additional requests, the performance will be penalized.

The effects of *Think Time* and *Wait Number* are similar. While, I think *Think Time* is better in practice. For realistic workloads, it is highly possible that some request will arrive after a long time. If the scheduler insist doing nothing until that request comes, its performance will be significantly penalized.

6. CONCLUSION AND EXPERIENCE

In this project, I explored the issue of implementing *Range Writes* in DiskSim simulator. In particular, I focused on whether the new write interface could reduce rotational latency, and how I could further improve its performance. From the experimental results, I think *Range Writes* has a promising future. The new interface could reduce rotational latency significantly. The results also show that using *Anticipatory Scheduling* in DiskSim could further improve *Range Writes*' performance.

There are some lessons I learned during the implementation of *Range Writes*:

- If the experiment is based on others' system, make good preparation for it. I spent almost 3 weeks on learning how DiskSim works. I learned a lot of lessions during that time. Now I see that before implementing my own idea, I should read the paper which introduces the system, and use a debugger to see what's going on inside the system.
- Focus on one experiment, and do it as good as I can. Do not dream to do many things at the same time. I remembered when I gave my presentation, I even havn't finished the *Vary Range* experiment. While working on some other problems. If there is not an additional week after presentation, I may fail this project. Next time, focus on the most important experiment.
- If I would like to do a project myself, make sure it is within my ability. Working on one project alone is a good experience, but it has a big risk. Were I have a partner, I may do more things and do things more quickly. How I wish I could ask someone for advice during the hard old days.

For future work, I think firstly I should run some realistic workloads. Someone suggested me to do *Range Writes* on RAID system. I think that is a good idea.

7. ACKNOWLEDGMENTS

I would like to thank Prof. Remzi for his advising and encouragement during all the semester. Many times I came to his office just having no experiemntal results. His good words and encouragement helped me a lot.

This is my first semester in Wisconsin Madison. And CS736 is my first graduate course. The hard old days have gone away, but I will always remember them.

8. REFERENCES

[1] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computing Conference*, pages 483–485, 1967.

[2] L. T. Dishon Y. Disk dual copy methods and their performance. In *Eighteenth International Symposium on Fault-Tolerant Computing*. IEEE, 1988.

[3] D. H. G. Myers, A. Yu. Microprocessor technology trends. In *IEEE. Vol. 74*, pages 1605–1622. IEEE, 1986.

[4] G. Ganger. System-oriented evaluation of i/o subsystem performance. *Ph.D. Dissertation, Report number CSE-TR-243-95 by the University of Michigan, Ann Arbor*, 1995.

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

[6] E. Lee. Performance modeling and analysis of disk arrays. *Ph.D. Dissertation, University of California, Berkeley*, 1993.

[7] C. U. Orji and J. A. Solworth. Doubly distorted mirrors. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 307–316, New York, NY, USA, 1993. ACM.

[8] F. Popovici, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Robust, portable i/o scheduling with the disk mimic, 2003.

[9] S. Salas and E. Hille. *Calculus: One and Several Variable*. John Wiley and Sons, New York, 1978.

[10] 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. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 17–17, Berkeley, CA, USA, 2000. USENIX Association.