

IDEDUMP - A Low Level Disk Profiler

Prashant Kediyaal
Department of Computer Science
North Carolina State University
Raleigh, NC 27695

December 2006

Contents

1	Introduction	1
2	High-Level Discussion	2
2.1	Overview	2
2.2	Software	4
3	Detailed Description	4
3.1	SystemTap	4
3.2	The <i>request</i> and <i>bio</i> data structures	5
4	Results	6
4.1	The Different scenarios	6
4.2	Graphs	6
5	Conclusion	8

Abstract

While considerable effort and work has gone in perfecting the existing scheduling mechanisms and file systems for the general case, the intuition is that perhaps a file system that profiles your disk usage may be able to use artificial intelligence techniques to learn how to better layout your data on your hard disk so that better performance is achieved.

In existing unix command or tools available freely, data you observe does not have microscopic details such as the starting LBA, the size, the start and end time of every block access request. Work on a similar intuition [2, 1] have approached this opportunity at a

higher level. To track such low level disk usage statistics we introduce idedump (inspired from tcpdump) written using SystemTAP that provides us the necessary profiling data we need. Standard benchmarks such as Bonnie/Andrew Bench Mark and others, are run on the same machine with various i/o scheduling mechanisms and the disk usage statistics is gathered. Graphs are presented that illustrate the different behaviours of the i/o schedulers.

1 Introduction

A precursor to suggesting improvements in i/o scheduling mechanism or even a file system is to be able to observe performance indicators of disk usage. While considerable effort and work has gone in perfecting the existing scheduling mechanisms and file systems for the general case, the intuition is that perhaps a file system that profiles an individual disk usage may be able to use artificial intelligence techniques to learn how to better layout data on a particular hard disk so that better performance is achieved. Such performance enhancement may be faster request completion, reduced average seek time and lower power consumption among others.

Existing unix command such as vmstat do provide a wealth of information and can be used to compare performance of different scheduling mechanisms and file systems, but the data you observe is not detailed enough to offer any intuition towards problem areas or what we may call opportunity areas. We require a microscopic view of disk activity that can track

details such as the starting LBA, the size, the start and end time of every block access request. While work on a similar intuition has been done previously [2, 1] they too approach this opportunity at a higher level. No tools are readily available to track such low level disk usage statistics and hence we introduce idedump (inspired from tcpdump) written using SystemTAP that provides us the necessary profiling data we need. Since block device requests are asynchronous;idedump produces a data file that contains two records per request, one each for the initiation and completion of the request. The record output is in the following format:

```
activityType(Read/Write) diskName LBA request-Size startTime(in microsec)
```

In this paper POSTMARK a standard benchmark, and kernel compile are chosen as system tasks to run. For each of the benchmark the same machine is run with various i/o scheduling mechanisms and the disk usage statistics is gathered using idedump. An instance of a 10 hour log of the machine running its normal course and an instance of a highly memory intensive AI application is also profiled. The paper then presents graphs that illustrate the different behaviours of the benchmarks described above. This help us in gaining insight towards opportunities that exist in enhancing disk performance. Preliminary data suggests that there does exist some opportunity.

2 High-Level Discussion

2.1 Overview

There are several mediums for secondary memory storage, removable floppy, tape drive and the trusted hard disk. Even among hard disks you have the IDE and SCSI. With a proliferation of storage devices and file systems how does the kernel maintain device and filesystem independence?

VFS: Virtual File System (sometimes called the Virtual File Switch) is the kernel subsystem that implements an abstraction over all the specifics. The VFS provides a standard of interface for user-space programs to enable them to interact with data stored over various storage mediums and file systems. All

drivers and filesystems are then required to have an implementation of some required system calls such as `open()`, `read()` and `write()` that the VFS can call.

For the purpose of this article we were looking into the measurement of a hard disk. Since the VFS exists at a higher level of abstraction the data structures associated with the VFS are necessarily oblivious of low level events of any storage medium. To get information such as the exact time at which the hard disk received a request and the exact time at which it completes the request, it is obvious that we have to dig deeper.

The data and request flow can be seen in 1 when a program is running in user-space it does not request data directly from the secondary storage medium. The program only knows the path to the data it needs, it then requests the kernel to perform operations such as `read()`, `write()`. These requests are then relayed to the drivers of the devices and the data is transferred back. The kernel is in the business of

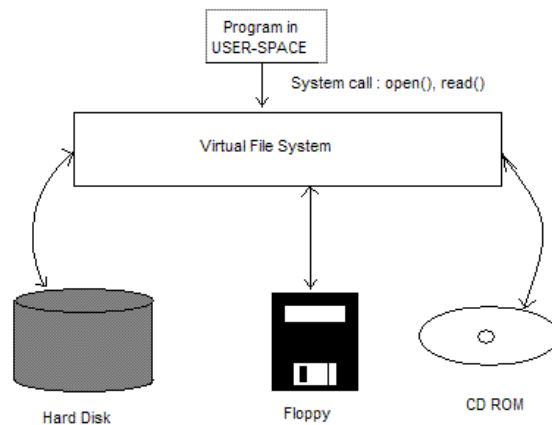


Figure 1: Virtual File System

managing the different hardware devices connected to the machine. For this, the kernel needs to communicate with all these devices. Most hardware devices

, such as hard disks are way slower when compared to the speed at which the CPU can operate. It would be a terrible waste if the kernel had to issue a data request to the IDE driver and then wait for the hard disk. Imagine a scenario where one can no longer type using the keyboard because the kernel is not available to service the keyboard since it is simply waiting for the hard disk to transfer data.

There are two solutions to such problems polling and interrupts. Polling is more wasteful than interrupt since it requires you to periodically give attention to resources that may not even need them, beyond this when the resource really wants to interface with you, you may be busy polling other things and the resource has to wait to be serviced. Interrupts on the other hand allow resources to initiate the conversation. For example, as I type this document the keyboard controller issues a signal to the processor to alert the operating system about the newly available key strokes. These signals are interrupts. Likewise hard disks are enabled with interrupts too. Requests for data are accomplished asynchronously by the kernel.

Hard disks are block devices. This means that accessing data off of this device requires random access of fixed-size block of data. Block devices are in contrast to character devices as character devices are accessed as a stream of data, one byte after another, as in a keyboard. Handling block devices is far more complex. The smallest addressable unit on a block device is called the sector. Sector size is typically 512 bytes but may be another value which is a power of 2. On the other hand filesystem use a different abstraction, the block. Although the block device is addressable at a sector level the filesystem performs all disk operations in blocks of data. Thus the block size can be no smaller than the sector and must be a multiple of it. The kernel has to deal with memory management where it deals with memory in terms of pages and hence the kernel place an additional requirement that the block be no larger than the page size. Common block sizes are 512bytes, 1kb and 4kb. After a read occurs (or when the write is pending), the blocks of memory accessed are stored in buffers. Each buffer is associated with exactly one block. The buffer then serves as the representation of (one or

more) sectors in memory. Since the block size is always a factor of 2 of the page size, the page can hold more than one blocks.

Now that we understand how memory from the hard disk is mapped to the RAM we can understand how request from an application in user-space is handled. When an application in user-space makes a system call and requests for data, the kernel simply places the request in a request queue and scurries off to handle other interrupts. The hard disk picks up the request. When the hard disk is done locating the appropriate block address it reads the data and using the Direct Memory Access Controller or bus mastering it completes the actual transferring of the data from the sector in the hard disk to the buffer in the RAM. Once the transfer request is completed an interrupt is raised, the request is marked complete and taken off the request queue. The hard disks, like other storage mediums have to provide implementation of their part of the process described above to interface with the VFS. This implementation is available under `/usr/src/"LINUX"/drivers/ide/ide-disk.c`.

One more detail, remember how we said the kernel simply places the request in the request queue for the hard disk to process? Well actually if the kernel were to simply issue requests in the order it received them it would result in awful performance. A FIFO handling of requests would entail that a request for data from a block on one part of the disk followed by another request to a completely other side of the disk be handled that way. Then if another request comes in that was adjacent to the first request the disk head has to scurry back. Instead the kernel subsystem called I/O scheduler manages adding requests to the request queue. Various scheduling mechanisms are available anticipatory, deadline, noop and completely fair queue. All of these mechanisms have at least on common goal, to merge different request such that request for data from adjacent blocks on the disk are issued as 1 request. When a request is completed the I/O scheduler, which manages the request queue, is responsible for taking the request off of the queue. This marks the end of request. The portion of the kernel code responsible for I/O scheduling is available at `usr/src/"LINUX"/block/elevator.c`

2.2 Software

We mentioned two source files in the earlier section `ide-disk.c` and `elevator.c`. While `ide-disk.c` contains the function `_ide_do_rw_disk` the I/O scheduler `elevator.c` contains the function `elv_completed_request`. The function names are self explanatory one marks the start of a read/write request the other the completion of request.

One way to write the tools for the IDEDUMP project would be to modify the above mentioned functions in the kernel source, but that would be too cumbersome. You are stuck with rebooting to a modified kernel that continuously profiles your machine, if you dont reboot to a normal kernel then your machine is continuously being profiled. A safer method exists. SystemTAP is a robust toolset available freely over the internet that allows you to write C like scripts that dynamically introduce "hooks" into kernel events such that your script is executed every time the identified event occurs. In our case these events are the two function calls.

Following is a list of all the files that constitute the IDEDUMP project related software. The list is indented to show dependency it is also ordered to show dependency. This means that from the list you can deduce that `idedump.sh` needs to be run (to collect profiling data) before `analyze.sh` can be used for analysis. Also it can be seen that `analyze.sh` needs `compact.sh`, `size.sh` etc to be able to execute. You get the idea

```
README.txt
idedump.sh
analyze.sh
  compact.sh
  size.sh
    size.gnu
    size.pct.gnu
  lba.sh
    lba.gnu
    lba_pct.gnu
  request_rate.sh
    request_rate.gnu
```

All of the shell scripts have a `-h` option that brings up help. While there are several scripts that make

up the project software, a user needs to interact with only two scripts (`idedump.sh` and `analyze.sh`).

To start profiling for 60 seconds and collecting data in a file name `IDEDUMP.out`

```
sh idedump.sh -t 60 -o IDEDUMP.out
```

Once profiled you can analyze your data simply by running

```
sh analyze.sh -all IDEDUMP.out
```

(this script outputs 5 graphs as gif files that are available in `./data` folder)

There are two more things to know. First, the output (`IDEDUMP.out`) of the `idedump.sh` shell script is raw data. Since I/O requests are asynchronous the profiling raw data contains two records for each request serviced. The first record is generated when the request is picked up by the ide driver. The second record is generated by the scheduler when it marks the request as complete and takes it off the request queue. This raw data is compacted by the `analyze` script before generating analysis data and graphs, this data is available in the `./data/stap.log` file. The second is the description of fields in the data files:

```
activityType(Read/Write) diskName LBA request-
Size startTime(in microsec)
```

3 Detailed Description

3.1 SystemTap

Due to the extreme importance of SystemTAP in enabling this project a brief description of the tool is covered here. Conventional route to modifying the kernel to test an idea or simply peeking at the variables in the kernel was a long process. SystemTAP provides us with a tool that easily allows us to write scripts that can examine every piece of the kernel without having to go through the modify, compile, reboot cycle.

Writing a SystemTAP script starts with naming an event. In our case these events are the function calls to `_ide_do_rw_disk` and the `_elv_completed_request`. Once we have the information about where in the kernel source we want to tap int, we insert a probe at that point in the event simply as:

```
probe kernel.function("elv_completed_request")
```

Simple as that. Follow the probe with any script that you want to be invoked whenever the kernel source traverses through that code. So in our case we wanted to print the activity type of the request that was completed, the starting LBA for the request, the number of sectors i.e the request size and the time at which the request was made. This is accomplished by:

```
probe kernel.function("elv_completed_request"){
    printf(" printf("SystemTAP provides us with
some convenient functions such as gettimeofday_us()
that we can easily access. To access data structures
inside the function that we just probed we use the
symbol $, thus $rq is a variable in the function.
Actually $rq is a data structure which is explained
in the next section.
```

To run the script above we do : stap myscript.stp. SystemTAP supports a number of built-in events. SystemTAP supports a number of built-in events that are named using a syntax that looks like dot-separated parameterized identifiers. We already saw the probe kernel.function which allows us to name any function in the entire kernel source. Some others are timer.ms(200), syscall.close.return to track returns from the close system call. These scripts that we write may be thought of as event handlers that are executed when events are fired. Two events probe begin and probe end provide us the convenience for putting pre and post processing code.

SystemTAP may be downloaded from <https://sourceware.org/systemtap> which contains tutorial documents, discussion forums and of course the software to download.

3.2 The *request* and *bio* data structures

We have so far understood how user requests are translated as block requests to the ide driver via the scheduler. We have also identified the functions in the kernel source that mark the start and end point of a request. Lastly we saw how SystemTAP can be used to probe functions and peek into the variables inside the functions. This section explains the relevant variables in the request data structure which are

referred to and tracked in the `idedump.stp` systemtap script that forms the heart of this project. Another important data structure is the `bio` structure which is also described.

The `_ide_do_rw_disk` function issues READ and WRITE commands to a disk using LBA to address sectors (or CHS Cylinder Head Sectors if LBA is not supported). The function is passed the request data structure that is managed by the scheduler subsystem of the kernel. The request data structure is the main object operated upon by the scheduler and is the authentic source of request. It has various variables with sector information. Some are `sector`, `nr_sectors`, `hard_sector`, `hard_nr_sector` and more. While the `sector` and `nr_sectors` indicate the next sector to submit and the number of sectors (i.e request size) the `hard_sector` and `hard_nr_sectors` indicate the next sector to complete and number of sectors left to complete. Since the systemtap script is initiated at the start of the function all of these values are same to begin with initially and hence any of the pair of variables will work. We have chosen to report information from `hard_sector` because that is also a term that is often used as a synonym for sectors generally.

The `bio` data structure forms the basic container for all block i/o within the kernel. This structure represents block I/O operations that are active as a list of segments. Where a segment is a chunk of buffer that is contiguous in memory. By representing buffers as a list of segments the `bio` structure allows the kernel to perform block I/O operation of a single buffer from multiple locations in the memory. This is called scatter-gather I/O since your buffer is scattered all over the memory in segments but you gather all of them together in the `bio` structure as a list of segments. The figure 2 illustrates some important fields in the `bio` data structure. Basically the `bi_io_vecs` point to an array of `bio_vec` structures. The `bio_vect` structures themselves stand in for individual segments. The `bio_vec` structures are of the form `ipage, offset, leni` which describes the segment in terms of the page it lies on, its offset from start of page and the `len` indicates the number of contiguous memory locations.

The figure 2 should help you visualize the mapping

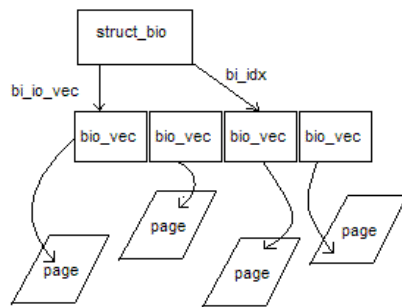


Figure 2: bio structure

from pages in memory to the bio structure. Now add to your vision that each of the *bio* structure represents one block I/O request. Remember though that each block i/o request does not mean a request for 1 block. One or more blocks may be represented as a list of *bio_vec* structures in one *bio* structure.

4 Results

4.1 The Different scenarios

The basic intuition of this project has been to examine low level statistics of disk usage and spot opportunity for improvements. In this context the various scheduling mechanisms anticipatory, cfq, noop and deadline, all seek to optimize user requests such that the users experience reduced latency and seek times. Hence several tests were run. The profiler gathered data for two types of activities i.e the postmark benchmark and kernel compile. These activities were performed over 4 different system configurations arrived at by changing the scheduler from anticipatory to cfq to noop and then to deadline. Besides these configurations there is one more configuration to be aware of, that of the postmark benchmark. Postmark was configured so that 5000 random transactions are carried out over 500 files which are bounded by maximum size to be less than 2MB.

Not much difference or opportunity was seen between these 8 configurations. Opportunity however was seen in two additional tests that were carried out. A 10 hour profiling of the normal usage of the system showed some interesting data. Another profiling of disk activity with a memory intensive AI application shows some data of interested. Hence the tables and graphs are presented for the 10 hour and AI profiles and also of compile and postmark run under cfq scheduler only.

4.2 Graphs

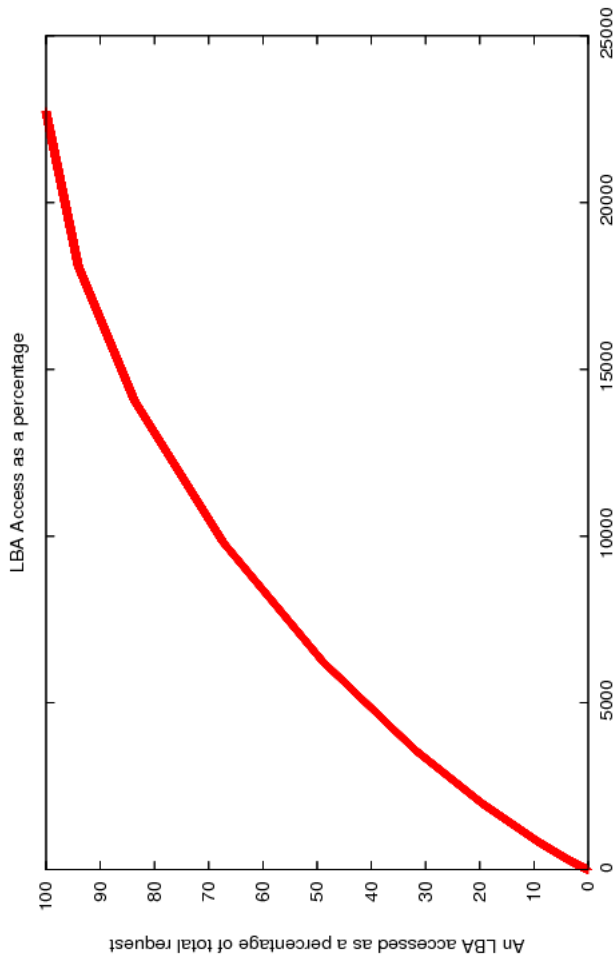


Figure 3: PostMark Benchmark

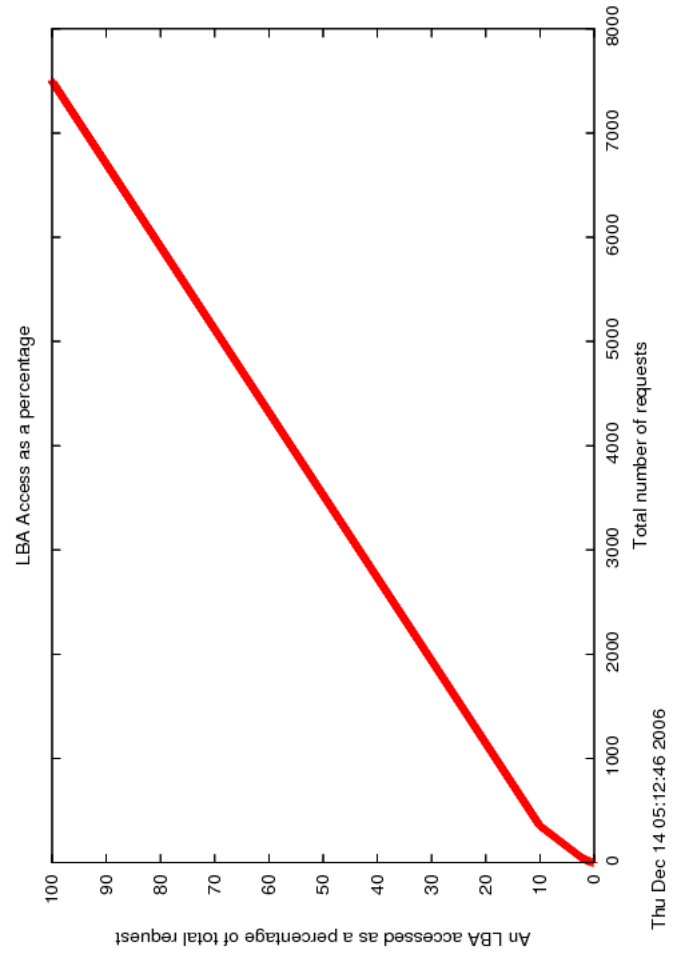


Figure 4: Compile

Thu Dec 14 05:12:46 2006

5 Conclusion

As mentioned before runs over the 8 different configurations resulting from changing over four different scheduling methods and running the postmark benchmark and kernel compile, do not result in interesting data. As a result not all graphs from these 8 test runs are presented. Instead 4 different tests for the same scheduler (cfq) is presented.

All The graphs presented show LBA's accessed as a percentage of total requests. In graph 6 it can be seen that roughly the top 50 LBA's constitute about 40 percent of the request. Similarly in 5 the same effect is seen. In the corresponding graphs 3 for postmark and 4 kernel compile with cfq scheduling the curve is almost linear indicating that there are hardly an LBA's that were repeatedly called for and missed by the cache.

Preliminary data such as this shows that for individual application there may be an opportunity to optimize access to LBA's that are called a lot of times but are missed by current scheduling methods. The postmark benchmarks graph is expected because it was configured to access random files all over the disk, it should be expected that there were no patterns. The same may not be true for kernel compile but it still shows similar characteristics as the postmark benchmark

References

- [1] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USENIX Summer*, pages 197–207, 1994.
- [2] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 15–24. ACM Press New York, NY USA, 1985.

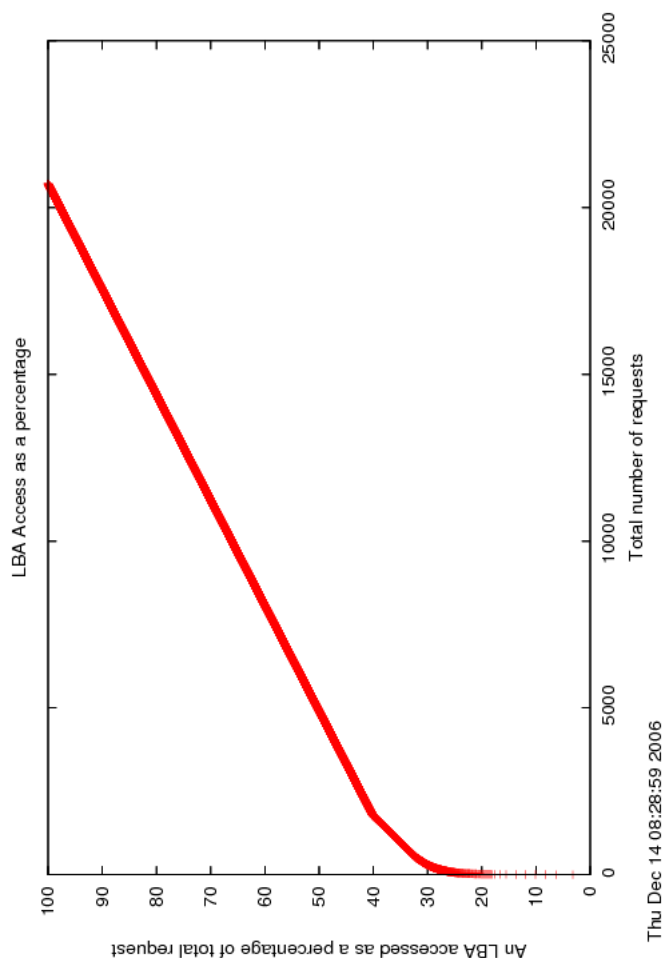


Figure 5: Disk Usage profile over 10 hours

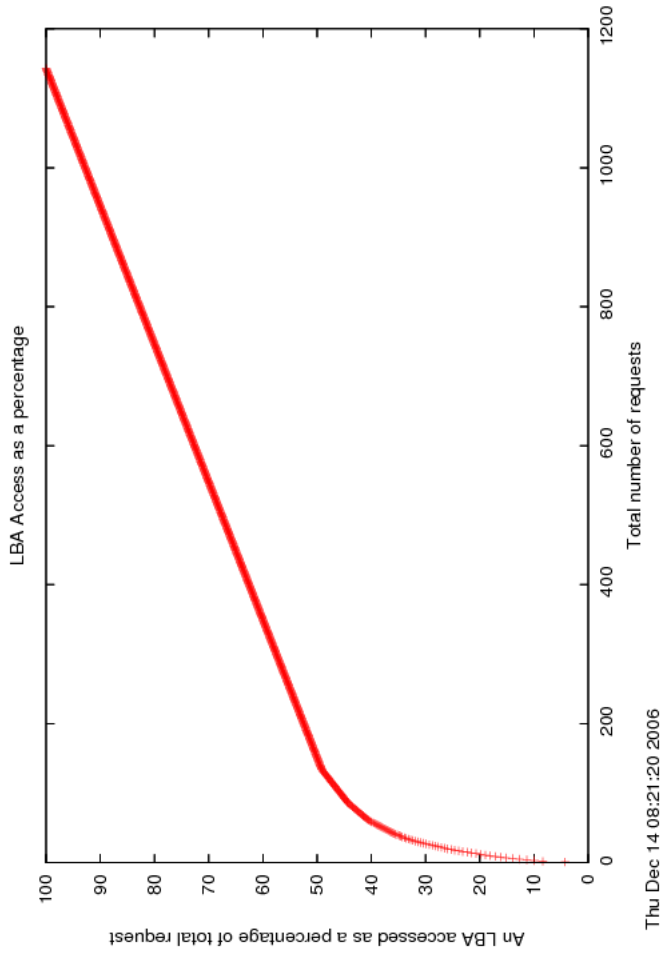


Figure 6: Disk Usage during a memory intensive AI application