Data Classification and Relaxing Storage Requirements

Brian Railing[*]
Advisor: Greg Ganger

April 30, 2004

Traditionally, there has been a single guarantee assumed for all data stored in a storage device: usually, the guarantee is that the data will remain unchanged and accessible. My work is explores applications that can continue to behave correctly even when the underlying storage does not meet this condition. Application data is classified based on the guarantees that the application requires; I will show an automated method for evaluating applications following changes to the disk's guarantees about its data.

1 Introduction

There has been much work put into improving access to the disk, with most work focused on improving the storage system access rather than the application access. In this paper, I will explore measuring the necessity of some of the guarantees for subsets of the application's data. Persistence accessibility are two areas I will focus on.

1.1 Motivation

Most applications require two functions from storage system: Stored data must only be changed by legitimate requests and requests for data at any time must be received in a timely fashion. However, some data does not always require these parameters. Unfortunately, no technique currently exists to relay this knowledge to the storage layer, in which I include the operating and file system.

My work explores separating application data into three classifications. "Normal data" relies on standard persistence and accessibility requirements. "Discardable data" is data that the application expects to meet the unchanged requirement; the storage system may have discarded the data at some earlier time. In this case, the application would respond to read requests with "discard". "Changeable data" relies on data being returned, not on changed data.

[*]School of Computer Science, Carnegie Mellon University, Pittsburgh PA (bpr@andrew.cmu.edu)

Recently, there has been work done developing a technique of storage performance evaluation called timing accurate storage emulation (TASE) [4]. TASE emulates future storage technology by using additional computer systems connected over SCSI. Storage emulation allows more detailed analysis of a proposed storage device, without the work to build a hardware prototype. In order to best meet its timing requirements, the data is stored in RAM rather than physical storage and therefore uses RAM more efficiently. The better TASE runs. This requirement limits the size of workloads that can be run on the system. Data classification came out of an effort to identify benchmark workload data that has different requirements for its storage and therefore may not need to be retained for correctness. For TASE, this would allow larger benchmarks to be run. Generally, by identifying this data, we may reduce the load on the storage system and distribute it to other parts of the system whose performance is growing faster than disks.

1.2 Determining Classification

There are two ways of determining the classification for a request or set of requests. Ideally, the application developers understand the requirements of the application's data and, using a future interface, pass this knowledge to the storage layer. Another technique is based on multiple executions of the deterministic application. While modifying various portions of the data to measure the effect on the application, the various classifications of the application's data are determined. Additionally, developing an automated technique will assist in verifying manual classifications.

1.3 Summary of Results

In real applications, a small increase in runtime can be traded for the ability to intercept and trace the syscalls made by the application. By the automated method explained in this paper, the data can be classified with high accuracy.

2 Theoretical Basis

Automated data classification poses three questions that need to be addressed. What are the application's actions? How do we create something meaningful to the application? How do we measure the effect on the application?

Intercepting all storage requests from the application can be achieved on multiple levels: user level, kernel level, and storage device level. First, we ensure that we only intercept requests from the application that are of interest. We must also choose an interception level that would better support the following steps.

The second question is how to modify the application's data while it runs so as to better determine what the data classifications are. Requests can vary greatly in size. This is nothing unusual. Having some idea of what parts of the request are important and what makes them important is a necessity.

Finally, in order to achieve anything from the steps in the automated classification, we must determine when the application's behavior changed. We have three approximations for comparing behavior: what blocks are read, what blocks are written to, and what data is written. Once we know that the application's behavior has changed, we can deduce that at least some of the data that the automated tool changed needed stricter requirements.

3 Design

3.1 Intercepting Requests

The best place for intercepting requests is at the user level. At this level, we can guarantee that we are only intercepting the requests from the application that we are testing. Furthermore, the entire system is more robust as failures, at worst terminate the current environment but not the system. The system is easily tested because changes to the tool can be tested with less turn around time. The interception tool generates a trace of the syscalls with the arguments and return values, which allows offline analysis of the application.

3.2 Modifying Requests

The question of modifying requests deals with testing the different requirements while treating the application as a black box. There are too many combinations to rely on trying all permutations, so instead, we categorize the data into three possibilities: negative, zero, and positive. As we run more tests, we can able to refine these categories to determine a better classification.

3.3 Measuring Impact

Comparing the impact that the classification technique has on the application involves a basic assumption. I assume the application will have identical behavior if it has identical data in the storage system. Future work may include adding support for applications that have variation in their performance even when there is identical data. The first comparison of performance involves what locations were requested to read. The background is that the decision of what blocks to read relies on previous reads. However, some applications may rely on randomness to determine which data blocks to read. Reading data has no impact on the data at the storage device. This does not guarantee a difference in the application's behavior.

Instead of observing the tool look at the reads, let's compare what the application sends to the disk. Should we compare where it is written or what is being written? Where the data is written does, at minimum, depend on allocation policies. What is being written will not always depend on what data the application reads. A simple exception is writing a pid or uid. In other words, there are occasions when either of these strategies, through not fault of the system, will fail.

4 Setup

The testing was run under the linux 2.4.19 kernel on a P3 700MHz laptop. There was online work suggesting a method by which the necessary interceptions in the user level of a system running linux can be performed[5]. The interception takes place by adding the tool's library to the dynamic libraries that are loaded at run time. The tool's library contains functions of the same names as the syscall wrappers in the standard library. Therefore, each call goes through the tool's function first before the standard library's function.

The tool was run on a collection of simple programs displaying certain characteristics. Each program reads from a file of random integers. Program A reads sequentially and writes the number of reads performed. A's reads are changeable. Program B reads numbers seeking between each read based on the number read until it leaves the file and writes the time it took to complete the reads. B's reads are not changeable under the strictest interpretation, but its output should not differ. Program C reads 10 numbers randomly chosen from the file and writes the average. The tool should note that C doesn't depend on the data for what is read, but its write depends on the data.

Additionally, two benchmarks in use are postmark 1.5 and ssh-build using OpenSSH (3.7p1). Postmark is a benchmark whose data can easily be recognized as changeable. Ssh-build is relatively simple to run yet has greater complexity than postmark.

5 Results

I looked at the results of running the classification tool on simple programs as well as two benchmarks. The tool currently classifies the data into changeable and normal, with counts of the number of requests that it determined were of each type. I looked at the results in three areas: correctness, effectiveness, and resources.

5.1 Correctness

Each of the programs is assumed to be fairly consistent in its requests. Only postmark lends itself to changes in its requests without impacting its behavior. There are small variations in the run time of the programs. For program B, these variations can result in incorrect classifications.

5.2 Effectiveness

The effectiveness of the tool is its support of the theory that there is changeable data. How much of an application's data is changeable? Table 1 shows the

		Automated Classification	
	Manual Classification (%)	Percent	Bytes
A	100	100	2048
B	100	99	2000
C	0	9	4
Postmark	100	100	2.99 MB
ssh-build			

Table 1: Changeable Read Data

determinations made by the tool about data read by the application and also shows in testing that the writes do not change. The small discrepancies in the classifications for programs B and C come from the small variations in runtime inherent in real systems.

5.3 Resources

More important for measuring the resource use of the the tool is evaluating its use of time, both in its classification of data, and in its impact on each run of the application. Table 2 contains the time to run the tool on each application and to generate the results seen in table 1. The tool's runtime is ideally linear with respect to runs and time per run, though there is the additional time dependence on evaluating each run.

	Runs	Time per Run (s)	Total Time
A	513	0.01	30.21 ± 0.10 s
B	506	0.01	29.97 ± 0.13 s
C	12	0.01	0.396 ± 0.011 s
Postmark	6373	2	4h 12m
ssh-build	29543	199	

Table 2: Time Requirements

Table 3 shows the increase in runtime for the applications when the interception library is included. There is a drastic difference between the benchmarks and simple programs, while the simple programs are quick the interception still results in a large increase in time. One reason for this is these simple programs are designed to exhibit certain requirements and are therefore io-bound. In addition, the overhead per operation is constant, which more dramatically escalates the time for smaller requests.

Finally it must be asked, is the tool easy to use? It is simple to run and outputs results, which summarize the classification of the application. The tool is designed towards a single run and only after major revisions to the application should it be necessary to rerun the tool.

	Interception (s)	Without Interception (s)	Percent Increase
A	0.92 ± 0.01	0.10 ± 0.01	820
B	1.04 ± 0.01	0.18 ± 0.01	477
C	0.01 ± 0.01	0.01 ± 0.01	0
Postmark	310.8 ± 1.1	331.7 ± 11.0	6.7
ssh-build	199.4 ± 0.3	196.6 ± 3.0	1.3

Table 3: Runtime with and without Interception

6 Related Work

There has been some work accounting for the difference in performance between storage devices [3], [8], which could be expanded to account for the time required to recreate data on the device itself. A couple works discuss setups where applications interface with the storage layer to decide what blocks should remain in the cache [2], [6] . A group at the University of Wisconsin studied classification within the storage device. This work was based on determining how the file system was using the blocks rather than the application's use [7]. Additionally, there has been some work on predicting the characteristics of files: like size, read / write ratio, etc, in order to improve accesses and allocation by knowing the needs of each file.

Finally, a group at Kentucky looked at categorizing data based on the level of persistence required by the application necessary. They also addressed an extension that would support the recreation of data that is lost [1].

7 Conclusions

With sufficient support from the operating system, we can present an expanded interface to the applications to further allow them to communicate details about their particular needs. This can improve response time by considering the needs of the system along with the requirements of the applications. Finally, the automated method presented by this paper, while accurate for distinguishing between "normal" and "changeable" data, can be expanded in the future to address "discardable" data and support greater variety in possible behaviors of the applications.

References

[1] T. Anderson, J. Griffioen. "An Application-Aware Data Storage Model." In *Proceedings of the 1999 USENIX Annual Technical Conference*, p. (June 1999)

[2] P. Cao, E. Felten, K. Li. "Application-Controlled File Caching Policies." In *Proceedings 1994 Summer USENIX Conf.*, p. 171–182 (June 1994)

[3] B. Forney, A. Arpaci-Dusseau, R. Arpaci-Dusseau. ”Storage Aware Caching: Revisiting Caching for Heterogeneous Storage Systems.” *Proceedings of the FAST 02 Conference on File and Storage Technologies*, p. 61–74 (January 2002)

[4] J. Griffin, J. Schindler, S. Schlosser, J. Bucy, G. Ganger. ”Timing-Accurate Storage Emulation” *Conference on File and Storage Technologies (FAST)*, p. 75 – 88 (January 2002)

[5] izik. ”hacking exists programs using ldpreload” http://sickpuppy.ath.cx/papers/libhack.txt

[6] T. Kimbel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, K. Li. ”A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching.” In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, p. 19 – 34 (October 1996)

[7] M. Sivathanu, V. Prabhakaran, F. Popovici, T. Denehy, A. Arpaci-Dusseau, R. Arpaci-Dusseau. ”Semantically-Smart Disk Systems.” In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, p. (March 2003)

[8] N.E. Young. ”On-line File Caching.” In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86. (January 1999)