I/O Workload Fingerprinting in the Genetic-Library

Jake Moilanen
IBM
moilanen@austin.ibm.com

Abstract

One great difficulty in writing an I/O scheduler is having one set of tunables which works well for every workload. If the I/O scheduler knew what kind of workload was occurring, it could modify its tunables for better performance. However, due to the I/O scheduler's depth in the kernel, it is very difficult to see this information. One method which can be used to obtain this information is to look at many small pieces of information, and then aggregate them to create a usable *fingerprint*.

This paper describes how to create an I/O workload fingerprint and its uses in both I/O schedulers, and in the genetic-library. The paper's main focus is on the application of the fingerprinting in the genetic library. By having a workload fingerprint, the genetic library can save genes which worked well for a particular workload, and reintroduce them back into the gene pool when that workload is seen again. This leads to faster convergence on an optimal tunable in an rapidly changing environment.

1 What is I/O Workload Fingerprinting?

Input/Output Workload Fingerprinting, or I/O Workload Fingerprinting, is a method of taking a number of small snapshots of individual performance metrics, classifying them, and aggregating all of them to create a `fingerprint` of the current workload. This information is used to assist I/O schedulers in making performance tuning decisions.

2 Motivation

The genetic-library [1] had a need to increase the speed which it converged on optimal tunables. When a workload changed, it took a great deal of time for the genetic-library to reconverge on the new optimal settings. To do this, the genetic-library must mutate and find good genes for the new workload. These mutations are really guesses, and guesses take time to get correct.

Thus emerged the idea of classifying workloads, and using the workload information to reintroduce known good genes to speed up convergence towards optimal genes. Reintroduction takes the guesswork out of the equation.

While the genetic-library is one user of the I/O workload fingerprinting, non-genetic-library I/O schedulers could make use of the classification. I/O schedulers can use this workload information to change their tunables, or even their scheduling algorithm.

3 How workloads are classified

These workloads are classified by how the I/O is occurring to the block device. The I/O operations have certain characteristics, such as being a read or a write, a sequential or random operation, and a size classification. Thus each I/O is broken down in three different dimensions:

Type:	Read/Write
Pattern:	Sequential/Random
Size:	Small/Large

Data for each of these dimensions is measured over a finite period, and used to determine which characteristics each dimension possesses.

To determine the `type` dimension, the number of read operations versus the number of write operations is calculated. If there are more than two times the number of read operations as write operations, then the dimension is classified as a read. Otherwise, it is classified as a write.

The `pattern` is either sequential or random. For each I/O operation, a measurement is made of the distance from the previous operation. These measurements are averaged over the finite period. If the average distance is large, then it is inferred that the disk head position is far away, and a random workload is occurring. Otherwise, if the distance is small, then the I/Os are close to each other and it is inferred that the disk operations are sequential.

The `size` dimension simply looks at the average size of each I/O operation and if the average is a page or less, then the workload is inferred to be small; otherwise it is large.

After the finite time period, these three dimensions are compiled together to form a `fingerprint` of the workload. This information is used by I/O schedulers and the genetic-library to help tune for the current workload.

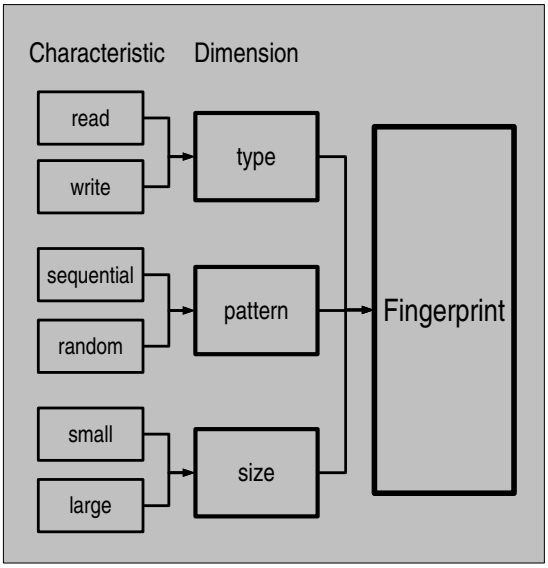

Figure 1: Fingerprint

3.1 I/O Workload Fingerprinting Terms

The term *workload* is defined as the characterization of what the system is doing during a finite period.

A quantifiable form of the workload is called a *fingerprint*.

For the purposes of this paper the term *dimension* is used in reference one aspect of the fingerprint.

The term *characteristic* is in reference to the possible outcomes a particular dimension can take.

4 How is it Implemented?

The I/O workload fingerprinting code is broken up into two pieces. The first is the helper functions which do the statistic and fingerprinting

calculations. The second piece is the user, who makes use of the fingerprint information.

The general code flow of the helper functions looks like Figure 2.

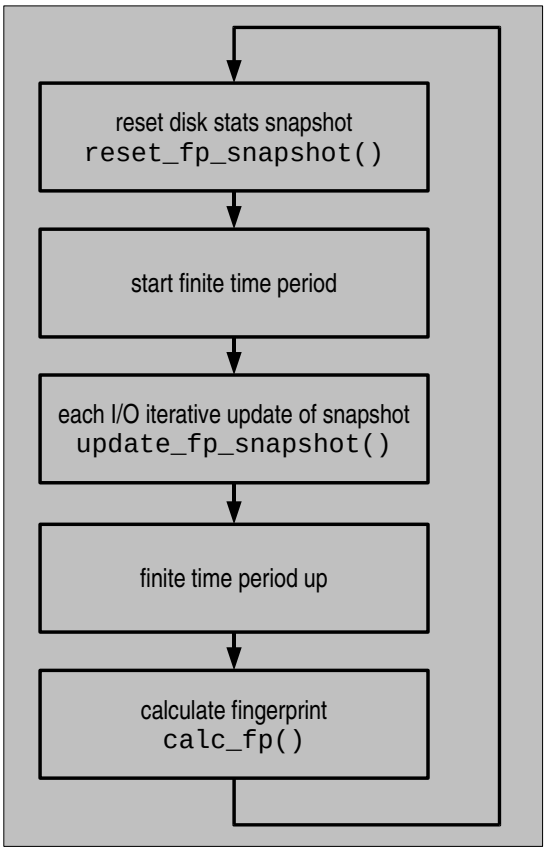

Figure 2: Codeflow

4.1 Reset snapshot

The workload is measured during a finite period, and the delta between two measurements is needed to determine the workload. Thus, at the beginning of the workload determination period the performance counters used for the workload determination are zeroed. From this point forward, any I/O operation is measured and counted towards this period's workload. The function that does this is `reset_fp_snapshot()`.

4.2 Start finite counters

The next step is to start the counters for the time period where the I/O workload is being determined. These counters are kept by the users of the fingerprinting helper functions, as there is no specific helper routines. Typically I/O is sporadic, and thus, to determine the workload a longer time period must pass to get accurate numbers. This time period needs to be at least in the order of tens of seconds.

4.3 Measure I/O metrics

Every I/O request makes a call to `update_fp_snapshot()`, which updates the snapshot of metrics with this I/O's information. The pertinent information is discovered by looking at the passed in `bio` struct. If the `bio` is a read, then the read count is incremented. Conversely, if the `bio` is a write, then the write count is incremented.

To determine the distance, the `bio->bi_sector` is used. It is inferred that this is the head position of the disk, and by taking the delta from the previous I/O's `bio->bi_sector`. This number is averaged in to the running average which has accumulated since the reset of the snapshot.

The size uses the `bio_sectors(bio)` value passed in. This value is averaged with the running average as well.

4.4 End finite period

After a predetermined amount of time, the timer pops, and the I/O workload period comes to a close. This timer handler calls into the `calc_fp()` routine to determine the fingerprint given the workload period snapshot.

4.5 Calculate the fingerprint

The `calc_fp()` call sets a fingerprint by looking at the snapshot results. The first thing determined is if the type is a read or a write. If there are more than two times as many reads as writes, then the workload type is considered to be read. The reason that this is not one-to-one is in most normal workloads there are far more reads than writes. Hence, the two times factor being used.

To determine the `pattern`, the average distance is used. If the average distance is more than `FP_CLASS_PATTERN_RAND` number of sectors, then the pattern is random. If it is under, then it is sequential. `FP_CLASS_PATTERN_RAND` is defined to be 25. This number was determined through experimentation in contrived workloads.

For the `size`, the average size is used. All buffered I/O has a minimum size of one page. Thus, if the size is greater than a page, then it is considered a large size. If it's a page, then the size is small.

Once the fingerprint is determined, this pass is complete. The next workload period is started, and the loops starts again at `reset_fp_snapshot()`.

5 Application in Genetic-Library

Figure 3 shows the code flow.

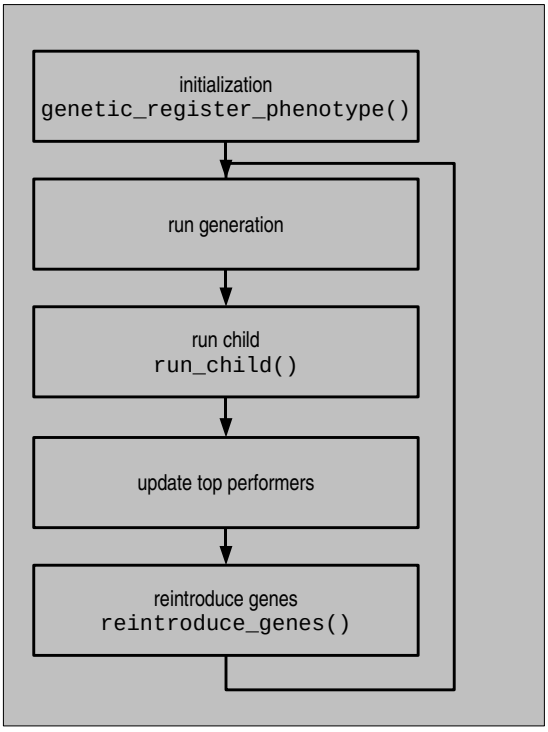

Figure 3: Genetic-Library codeflow

5.1 Initialization

During the genetic-library initialization, two three-dimensional arrays are created. The first dimension of the array is for the type, the second is for the pattern, and the last is for the size. Of the two arrays created, the first is for the top genes of each workload. The second is for the top-fitness of each workload.

There is a callback, `create_top_genes()`, which does the initialization of the genes for the particular workload. If good genes for a particular workload are known, then those are set.

5.2 Run generation

The kickoff of a new generation also kicks off the finite timers for the generation. The genetic-library uses the generation timers as the finite timers for the I/O workload determination. By using these timers, the I/O workload fingerprinting is in line with the genetic-library generations, and can tailor a new generation to the current workload.

5.3 Run child

Each child in the generation takes their fingerprint snapshot, and consolidates it with the generation's snapshot. This is done through a fingerprint helper function, `consolidate_fp_snapshot()`. This function takes one master snapshot, and updates the other child snapshots to it. This includes adding the reads and writes, incorporating the average distance, and the average size.

Once the child has updated the generation master snapshot, it resets its snapshot for the next time it is called.

5.4 Update top performers

At the end of a generation, the fingerprint is calculated, and used to determine if this generation was the best for this workload. This is done by comparing the previous top fitness for this workload. If this workload had a better fitness, then the average of this generation's genes are saved off, and its fitness is used as the top fitness for this generation.

There is also a decay factor on the top fitness for the current workload. Just in case there was a spike with less-than-optimal genes, the current workload's top fitness is reduced every pass through. This allows for self-correcting in an environment which spikes.

5.5 Reintroduce generation

When the current fingerprint changes from the last fingerprint, it indicates that the workload changed. This is the opportune time to reintroduce the genes which worked well on this workload. This is done via `reintroduce_genes()`. The first child is arbitrarily picked to get the reintroduction of genes. This is done since no matter what the child count is, there is always at least one, so the first is a safe one to put them in. This reintroduction of the genes is only done on the switch of workloads and not every generation in order to not continuously get bad genes which got set in the top gene's array because of a spike. Otherwise it could take a while for the decay to kick in and correct the genes.

6 Performance

For the genetic-library, the main purpose of I/O Workload Fingerprinting is to converge on optimal tunables quicker during a changing workload. To test how well it performed, the flexible file system benchmark [3], or FFSB, was used. The FFSB is a versatile benchmark which is able to simulate most any I/O workload.

In the performance evaluation, an OpenPower 710 system, with 2 CPUs, and 1.848 gigabytes of ram was used. The benchmarks were conducted on a SLES 9 SP3 base install with a 2.6.16 kernel. More system details can be found in Appendix A.

To determine the convergence time, four different workloads were simulated. These included a random read, a random write, a sequential read, and a sequential write. The workloads were cycled to be as malevolent as possible for the genetic-library. For instance, the benchmark started as a sequential write, and then went to the polar opposite, random read. This typically requires the genetic-library to search for all new genes.

Two runs were conducted. The first was a standard genetic-library without I/O workload fingerprinting turned on. The second run had the genetic-library plus I/O workload fingerprinting. In the second run, two passes were done.

The first pass warmed up good genes for the workload fingerprinting, the second pass was with a warm set of optimal genes.

The convergence was detected by pulling every child's genes during the run, and then plotting them. Visual inspection clearly showed the one or two dominant genes in a particular workload converging to a single value. Once those genes finally reached that value, convergence has occurred.

6.1 Results

As show in Figure 4, faster convergence did occur. As the second pass of the fingerprinting run had a drastic reduction in convergence time. Both sequential read and sequential write converged with an 89% and a 97% reduction in time, respectively. Random read and random write converged with an 61% and 19% reduction in time, respectively.

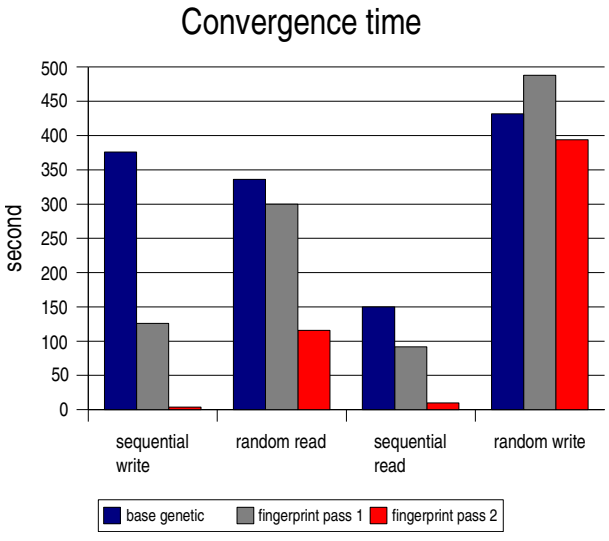

Figure 4: Convergence time

In addition to the improvement of the second pass, the first pass of the fingerprinting run did see some improvements as well. There are two factors which contributed. While the benchmarks were immediately run once login prompt was reached, there is an amount of warming of the optimal genes which occurs from bootup. This would mostly be seen in random read. The other factor is because all workload gene pools are initialized to the Anticipatory I/O scheduler defaults. On a malevolent workload change, the defaults are generally closer to the optimal genes than the current tuning.

7 Future Work

At the time of this paper, use of the I/O workload fingerprinting was reserved only for the genetic-library. Expanding it to interact directly with the Anticipatory I/O scheduler would be ideal. Currently the Anticipatory I/O Schedule is tuned to optimize sequential read operations [2]. If the workload deviates, then performance suffers. The I/O workload fingerprinting could set optimal tunables as workload changes and would greatly improve the overall performance of the Anticipatory I/O scheduler. The optimal tunables for each workload could be pulled from where the tunables converge in the genetic-library during contrived workloads.

Other future work includes setting tunables in a per-disk basis, as some systems have a RAID setup in addition to an IDE disk. The workloads between those two devices can vary greatly. However, if it was known what type of workload each was performing, then each disk could have its own set of tunables and could increase the overall performance.

Expanding this idea of workload fingerprinting to CPU workload fingerprint is an interesting idea. By taking small pieces of information and aggregating that information to an overall CPU workload, fingerprinting could be useful for the CPU scheduler. At the current time, no proposals have been made as to how to do this; it is an interesting problem that would be useful to solve.

8 Conclusion

The performance numbers clearly show a drastic improvement on the convergence time. By increasing the convergence rate, the I/O workload fingerprinting pushes the usability of the genetic-library on a desktop environment. It also greatly improves the aggregate performance of the genetic-library, as it does not waste time with less-than-optimal genes on a changing workload.

Legal Statement

Copyright 2006 IBM.

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM, the IBM logo, and POWER are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

All the benchmarking was conducted for research purposes only, under laboratory conditions. Results will not be realized in all computing environments.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates. This document is provided “AS IS,” with no express or implied warranties. Use the information in this document at your own risk.

References

[1] Moilanen, J., Williams, P., *Using genetic algorithms to autonomically tune the kernel*, 2005 Linux Symposium

[2] Pratt, S., Heger, D., *Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers*, 2004 Linux Symposium

[3] http://sourceforge.net/projects/ffsb/

Appendix A. Performance System

IBM OpenPower 710 System
2-way 1.66 Ghz Power5 Processors
1.848 GB of memory
15,000 RPM SCSI drives
SLES 9 SP3
2.6.16 Kernel

Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.