2017

Enhancing the Accuracy of Synthetic File System Benchmarks

Salam Farhat
Nova Southeastern University, salalimo@gmail.com

Enhancing the Accuracy of Synthetic File System Benchmarks

by

Salam Farhat

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor in Philosophy
in
Computer Science

College of Engineering and Computing
Nova Southeastern University

2017

We hereby certify that this dissertation, submitted by Salam Farhat, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Gregory E. Simco, Ph.D. Chairperson of Dissertation Committee	Date
Sumitra Mukherjee, Ph.D. Dissertation Committee Member	Date
Francisco J. Mitropoulos, Ph.D. Dissertation Committee Member	Date

Approved:

Yong X. Tao, Ph.D., P.E., FASME Dean, College of Engineering and Computing	Date

College of Engineering and Computing
Nova Southeastern University

2017

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Enhancing the Accuracy of Synthetic File System Benchmarks

by
Salam Farhat
May 2017

File system benchmarking plays an essential part in assessing the file system's performance. It is especially difficult to measure and study the file system's performance as it deals with several layers of hardware and software. Furthermore, different systems have different workload characteristics so while a file system may be optimized based on one given workload it might not perform optimally based on other types of workloads. Thus, it is imperative that the file system under study be examined with a workload equivalent to its production workload to ensure that it is optimized according to its usage.

The most widely used benchmarking method is synthetic benchmarking due to its ease of use and flexibility. The flexibility of synthetic benchmarks allows system designers to produce a variety of different workloads that will provide insight on how the file system will perform under slightly different conditions. The downside of synthetic workloads is that they produce generic workloads that do not have the same characteristics as production workloads. For instance, synthetic benchmarks do not take into consideration the effects of the cache that can greatly impact the performance of the underlying file system. In addition, they do not model the variation in a given workload. This can lead to file systems not optimally designed for their usage.

This work enhanced synthetic workload generation methods by taking into consideration how the file system operations are satisfied by the lower level function calls. In addition, this work modeled the variations of the workload's footprint when present. The first step in the methodology was to run a given workload and trace it by a tool called tracefs. The collected traces contained data on the file system operations and the lower level function calls that satisfied these operations.

Then the trace was divided into chunks sufficiently small enough to consider the workload characteristics of that chunk to be uniform. Then the configuration file that modeled each chunk was generated and supplied to a synthetic workload generator tool that was created by this work called FileRunner. The workload definition for each chunk allowed FileRunner to generate a synthetic workload that produced the same workload footprint as the corresponding segment in the original workload. In other words, the synthetic workload would exercise the lower level function calls in the same way as the original workload. Furthermore, FileRunner generated a synthetic workload for each specified segment in the order that they appeared in the trace that would result in a in a final workload mimicking the variation present in the original workload.

The results indicated that the methodology can create a workload with a throughput within 10% difference and with operation latencies, with the exception of the create latencies, to be within

the allowable 10% difference and in some cases within the 15% maximum allowable difference. The work was able to accurately model the I/O footprint. In some cases the difference was negligible and in the worst case it was at 2.49% difference.

Table of Contents

List of Tables

List of Figures

List of Equations

Equations

Chapter 1

Introduction

Background

Operating system benchmark suites consist of tools that aide in file system performance measurement and analysis (Agrawal, Dusseau, & Dusseau, 2008, Traeger, Zadok, Joukov, & Wright, 2008). Integral to benchmark suites are workloads that exercise the target systems in repeatable scenarios that are indicative of common application execution. One workload generation methodology involves the study of operating system access patterns by a set of applications that exhibit functions common to production software. The patterns are then incorporated into a tool that reproduces a typical application's function into representative synthetic workloads in the context of benchmark suites (Agrawal, Dusseau, & Dusseau, 2008). Thus the generated workload reproduces primary and secondary storage access with the goal of providing equivalent execution sequences through application programming interface (API) or systems calls indicative of an application's interaction with the operating system (Roselli, Lorch, & Anderson, 2000).

Current synthetic benchmarks do not generate representative workloads due to the assumption that mimicking an application's API calls is sufficient to reproduce the application's workload (Traeger, Zadok, Joukov, & Wright, 2008). This oversimplified assumption fails to account for the different execution paths that consist of lower level operating system function calls (Agrawal, Dusseau, & Dusseau, 2008, Joukov, Wong, & Zadok, 2005). An API call can have more than one execution path with a significant difference in performance. As a result two workloads with the same API calls but different execution paths will have a different performance footprint that is dependent on the function calls. An example of an API call that has multiple execution paths is the read API call. The read API call can have an execution path that consists of function calls that reads the file from primary storage, or an execution path that consists of function calls that reads the file from disk whose performance depends on whether the file is fragmented or not.

Another factor that synthetic workloads ignore is the variation within the workload. Workloads are not uniform and exhibit different workload characteristics as the workload progresses. For example, the workload's read/write ratio, average file size per request, or frequency of requests change over time. Synthetic workloads do not model this variation in their workload generation process.

The work in Agrawal, Dusseau, and Dusseau (2008) advanced file system benchmarking by taking into account the execution paths for the API calls. The application traces recorded the execution paths for each API call and the synthetic workload generation process generated API calls with the goal of having their execution paths match the execution paths of the original workload. The result of their work was a tool called codeMRI. Although the work presented the results of a few workloads that were run using the Postmark benchmark their results were not compared with other techniques to indicate whether there was any improvements gained from codeMRI. In addition, the work stated that additional refinement was needed for codeMRI to improve the accuracy of the synthetic workloads generated by it.

Another benchmarking technique consists of collecting traces of live workloads and replaying those traces on the same system (Joukov, Wong, & Zadok, 2005). Given the same system with the exact same state the traces would exercise the hardware in an equivalent manner. Trace replaying does not provide a mechanism to adjust the workload characteristics to observe how changes in the workflow can affect the overall performance (Zhang, Sivasubramaniam, Franke, & Gautam, 2004).

The inflexibility and limited availability of traces make synthetic workloads a preferable choice, but synthetic workloads suffer from being unrepresentative of real workloads (Zhang, Sivasubramaniam, Franke, & Gautam, 2004). The source of the inaccuracy problem is in the approach that existing synthetic workload generators take that neglects how the API calls are satisfied by the function calls (Traeger et al., 2008, Agrawal, Dusseau, & Dusseau, 2008).

The work developed by Agrawal, Dusseau, and Dusseau (2008) created a tool called codeMRI to address the issue of synthetic workload inaccuracy. In their work the traces of a live workload was recorded that included information about the API calls and the execution paths. The synthetic workload was then synthesized in a manner that tried to mimic the traces at the API level and the function level. Although codeMRI presented a technique for enhancing the accuracy of synthetic workloads it did not do it in a repeatable manner. The description of the methodology is not sufficient to regenerate the work and one component of codeMRI, namely the workload synthesis process, still requires some additional work and refinement as dictated by the authors. This implies that further work is required to enhance the accuracy of synthetic workloads.

This work advanced synthetic benchmarking research by improving the accuracy of synthetic workloads by taking into account the execution paths of the API calls and by modeling workload variation if it exists in a workload. The execution paths are recorded from the virtual file system (VFS) level that capture file system calls that are satisfied by primary storage and secondary storage. More specifically this work captured traces at the virtual file system level and generated a synthetic workload that more accurately matches the execution profile of the original workload.

This work was accomplished by extending the methodology presented by Tarasov, Kumar, Hildebrand, Povnzner, Kuenniehng, and Zadok (2012) that creates synthetic disk workloads. The approach in Tarasov et al. (2012) generates the configuration files for common file benchmarks such as FileBench that are based on the disk traces to produce equivalent synthetic workloads that match the trace's footprint on disk. These disk-based approaches do not consider the portion of the workload that is satisfied from primary storage since it is very difficult to relate the disk operations back to the API calls (Shan, Antypas, & Shalf, 2008). As a result this approach may produce equivalent workloads on disk, but it does not reproduce the application's workload since it does not consider the file system operations that were satisfied from primary storage. Hence, disk workloads cannot be used for file system benchmarking. However, this work adapted a similar methodology that used VFS level calls instead of disk level traces.

In the following section the three different approaches presented in this section for file system benchmarking are presented in further details emphasizing the shortfalls of each method. This is followed by a discussion on the goal of this work that proposes to extend application benchmark research by providing a methodology for improving synthetic workloads by making them more representative of live workloads.

The next section, relevance and significance, details the extent of the problem and provides some insight on what the goal would accomplish and its impact on the current state of synthetic workloads. This is followed by a brief review of the literature where the latest techniques of synthetic disk workloads are discussed outlining the shortfalls of each method to generate accurate workloads. In addition, some experimental results that illustrate the reason why current file system benchmarks do not create representative workloads are presented, and finally a description of a synthetic disk workload technique that will form the basis of the approach is also presented. The barriers and issues for research in this area are discussed in the subsequent section. The main obstacle includes properly capturing the traces and preprocessing them as input to the synthetic workload generator.

The approach is described in the following section that will contain the main steps that are needed to capture the trace, analyze and preprocess the trace, and generate the required configuration files for a tool created by this work that would generate the equivalent synthetic workload called FileRunner. In the next section the different milestones for this work are outlined and a description of how to test each milestone is presented. This is followed by the final section of this work that lists the resources that are going to be used.

Problem Statement

The workload generation approach used by current benchmarks is not sufficient at generating equivalent execution patterns across multiple operating systems because API calls are not always satisfied with the same operating system kernel function calls (Agrawal, Dusseau, & Dusseau, 2008). In this section a couple of experiments are presented that will demonstrate how the function calls can skew the workload's performance. Then a discussion about a tool called codeMRI that takes into account the different execution paths when constructing synthetic workloads is presented. The discussion of codeMRI is focused on how it attempted to incorporate the execution paths and how it still requires some improvement. Finally, another approach for file system benchmarking that is comprised of capturing and replaying traces is presented. The discussion of trace replaying will focus on its advantages and shortfalls when it comes to file system benchmarking.

A read API call can be satisfied from the file system buffer cache, from disk as a sequential read, or if the file is fragmented would be read from different locations on disk. The work in Jacobs (2009) provided measurements that validate the significant difference in the performance between disk and memory and whether the reads are sequential or random. The experimental setup consisted of a Windows server 2003 machine with 64GB RAM and a set of 8 disks setup in RAID5. The data read were chunks of 4 byte values that were accessed sequentially and randomly from both disk and memory. The results showed that sequential memory access was fastest at 358.2 million values per second while the second fastest was sequential access from disk at 53.2 million values per second followed by random memory access at 36.7 million values per second and finally the significantly slower random disk access at 316 values per second.

The vast difference in performance between primary and secondary storage is one of the main reasons current synthetic workloads do not accurately represent live workloads. The work of Traeger, Zadok, Joukov, and Wright (2008) provides evidence of the inconsistencies of the execution between real and synthetic workloads. In their experiment the Postmark benchmark was executed with the default configuration that assumes the system has a small memory footprint, thus the benchmark generated small sized files between 500 and 10,000 bytes and the benchmark took less than one tenth of a second to complete. Beyond the initial access, I/O operations were satisfied from the cache and thus did not require additional reads from disk. This illustrates the lack of low-level I/O functions that were represented in the description of the benchmark test suite thus skewing the results towards the reduced cost of memory access.

The experiment demonstrates how the workload's performance can be skewed by the execution paths. For synthetic workloads to be representative of live workloads they need to have the same performance footprint as live workloads. This means that they need to match the live workload's API calls and function calls. The only benchmark that considers the lower level function calls is the codeMRI benchmark that is presented in Agrawal, Dusseau, and Dusseau (2008). The authors of codeMRI presented the work at a high level lacking any sufficient details that makes it possible to recreate or use any portion of the codeMRI approach. While the work claims that codeMRI generated sufficiently accurate workloads it also states that the workload synthesis process requires additional work without expressing what portions of the process requires the additional work. This makes codeMRI infeasible for use as it is presented, but the goal and high level description of the methodology are of value.

Another downside for codeMRI is that methodology relies on source code analysis for the application that is being benchmarked that in some cases may not be available such as systems that use commercial of the shelf products. In addition, even though the paper presents experimental results that highlight the need to consider the execution paths to generate accurate workloads the entire work is not presented in a repeatable manner making the codeMRI difficult to use or enhance.

One other issue that codeMRI and other synthetic workloads do not address is the variation within the workload. The workload's characteristic vary across time especially when the workload is observed over an extended period of time. Synthetic workloads such as FileBench generate a uniform workload over time. The workload's characteristics are specified via the configuration file that is parsed in the beginning of the workload. Then FileBench generates requests that are based on the configuration resulting in a uniform workload. The work in Tarasov et al. (2012) addressed a similar issue when generating representative disk workloads. However, this has not been addressed in the context of file system synthetic workloads.

A different approach for benchmarking includes capturing traces and replaying them, possibly on a different system, with the goal of reproducing the exact workload on the file system (Joukov, Wong, & Zadok, 2005). Traces can be captured at different levels of a given system that include the driver level, network level, virtual file system level, and system call level with each level being suitable for a specific type of benchmark studies. For instance, the driver level traces capture disk activity and are suitable for examining disk layout strategies and disk performance. Driver level traces do not capture requests serviced from primary storage so these traces cannot represent the complete application workload and thus it cannot reproduce it (Joukov, Wong, & Zadok, 2005).

Similarly, network level traces capture all requests that are made via the network but do not capture any local requests that makes it impossible to deconstruct the application's local workload in its entirety (Joukov, Wong, & Zadok, 2005). System call level traces capture all the requests that are made by the application but do not trace how these calls are satisfied by the lower level function calls (Joukov, Wong, & Zadok, 2005). Since it is not possible to distinguish what system calls were satisfied from primary storage or those from secondary storage it would not be possible to construct an accurate workload that is representative of the original workload in terms of how it exercises the OS and hardware.

The virtual file system (VFS) traces (Joukov, Wong, & Zadok, 2005) capture all local file system operations generated by an application including information on the operations that were serviced from primary storage or secondary storage. This is similar to the codeMRI methodology discussed earlier where the function calls are essential for reproducing the workload accurately (Agrawal, Dusseau, & Dusseau, 2008).

Tracefs is a tool that captures VFS level operations and is used by Replayfs (Joukov, Wong, & Zadok, 2005) to replay those traces on any system (Aranya, Wright, & Zadok, 2004). Replayfs imports the traces captured by tracefs, and then it preprocesses those traces into the operations that will be executed on the file system. This process converts the raw traces into the format that includes the following elements: VFS operation such as read and write, the process ID, the return value of the VFS operation, and the parameters of the VFS operation. Once the preprocessing is complete Replayfs goes through the list of operations in a sequential order and runs those operations on the file system.

The problem with replay methods is that their workload generation is restricted by the limitations in the captured traces they are replaying. First, traces are not always available for several reasons such as that the data itself might be proprietary or cannot be exposed for security reasons or system builders may simply not have access to it especially if the system is new and no live data exists. When traces are available they represent a snapshot of the system at the time of the trace capture and they may not reflect the workload across other time intervals where the workload characteristics are significantly different. For example, traces may be captured at a time where the system was experiencing normal traffic load as opposed to capturing traffic during peak time (Zhang, Sivasubramaniam, Franke, & Gautam, 2004). This will allow system designers to fine tune the performance of the system based on the normal traffic load, but would not be able to fine-tune the system based on peak traffic.

Another big disadvantage is that during trace replay commands are replayed sequentially and can only be replayed sequentially (Zhang, Sivasubramaniam, Franke, & Gautam, 2004). This restriction makes trace replaying inflexible when it comes to studying an application's performance for projected workloads, because it is not possible to adjust certain parameters such as file sizes and then observe the change in performance. The flexibility in adjusting workflow parameters will allow system designers to discern the application's performance upon expected or projected changes in possible future workloads. This type of study is not possible with trace replay, but is possible with synthetic workloads (Zhang, Sivasubramaniam, Franke, & Gautam, 2004).

Unfortunately, synthetic workloads suffer from one issue that is they do not generate representative workloads. As discussed earlier this is due to the fact that they do not take into account how the API calls are satisfied via the function calls (Agrawal, Dusseau, & Dusseau, 2008). As was presented earlier, codeMRI is one exception that created representative synthetic workloads by taking into account the lower level OS calls, but this method relies on the source code to generated accurate workloads that is not always available. More importantly, no sufficient details have been provided to improve the work or even apply the work for actual use. In conclusion, the latest synthetic file system benchmarking research including codeMRI provides a need for highly accurate synthetic workloads and that further research is required to increase their accuracy.

Dissertation Goal

This work advanced file system benchmarking by providing a tool to generate synthetic workloads that are more accurate compared to synthetic workloads generated by traditional benchmark approaches. The enhanced accuracy was achieved by mimicking the original workload's cache behavior and by modeling the variation that existed in the original workload.

Mimicking the cache behavior was first accomplished by tracking the system calls that are satisfied from primary storage and those that are satisfied from secondary storage (Agrawal, Dusseau, & Dusseau, 2008, Zhang, Sivasubramaniam, Franke, & Gautam, 2004). This information was collected by a tool called tracefs that captures the VFS calls as well as the address space operations that dictates whether the VFS calls where processed from disk or cache. Next, the workload data such as VFS calls and their parameters were extracted from the trace files and used to configure FileRunner, a tool created by this work, to generate the synthetic workload. To model the workload variation the chunking method proposed in Tarasov et al. (2012) was used. This primarily splits the workload into subintervals small enough to consider the workload in each interval as uniform. A synthetic workload was then generated for each chunk and run in succession.

The work was evaluated using some workloads that were used to be traced so that equivalent synthetic workloads were generated. Synthetic workloads were then created from the traces and the throughput, latency, and I/O size, were observed for both the original workloads and the synthetic workloads. The work in Tarasov et al. (2012) states that their synthetic disk workloads had an average margin of error less than 10% with some parameters not exceeding the maximum allowable 15%.This was assumed sufficient enough to declare that the synthetic disk workloads were representative of the workload being emulated. This work followed the same assumptions.

Relevance and Significance

Benchmarks generate representative synthetic workloads that allow system designers to analyze and optimize the file system based on its usage (Joshi, Eeckhout, Bell, & John, 2008). Since performance is greatly dependent on usage the accuracy of the synthetic workloads becomes crucial so that the system is designed for optimal performance (Zhang, Sivasubramaniam, Franke, & Gautam, 2004).

Inaccurate workloads results in a system that is optimized based on that workload and may not produce optimal performance when the system is in production. In addition, adjusting workload parameters such as file size or the read-write ratio is a desirable feature available only with synthetic workloads that can provide an insight of how the system might behave as workload characteristics vary and can predict the workload performance (Zhang, Sivasubramaniam, Franke, & Gautam, 2004).

As discussed in the problem section the latest work in synthetic file system benchmarking research either produces inaccurate workloads or in the case of the work in (Agrawal, Dusseau, & Dusseau, 2008) the codeMRI methodology still requires some additional work in the workload synthesis process. This indicates that the state of the art of file system benchmarking can still benefit from synthetic workloads that are more representative of actual workloads.

The work proposed in this paper advanced synthetic benchmarks by creating more accurate synthetic workloads that was achieved by incorporating function calls in the workload generation process without the need for the source code as is the case in codeMRI and present it in a repeatable and reproducible manner. The methodology of this work borrowed some of the techniques presented in the synthetic disk workload work in Tarasov et al. (2012).

Barriers and Issues

Traditionally synthetic workload generators considered that to create an accurate synthetic workload it needed to match the live workload's API calls (Agrawal, Dusseau, & Dusseau, 2008). This helped simplify or hide the complexity of file systems, the complexity of disk, and the numerous factors that contribute to the overall performance (Traeger, Zadok, Joukov, & Wright, 2008, Tarasov, Bhanage, Zadok, & Seltzer, 2011). This basically eliminated the need to track how a given API call is satisfied by the different execution paths. While this assumption simplified the workload generation process the work in Traeger, Zadok, Joukov, and Wright (2008) and Tarasov, Bhanage, Zadok, and Seltzer (2011) discussed earlier demonstrated that this simplified approach does not produce accurate results.

The work in Agrawal, Dusseau, and Dusseau (2008) tackled this issue and produced a synthetic benchmark that takes into account the function calls. This enhanced the accuracy of synthetic workloads. However, codeMRI still requires additional work as stated by the authors and in addition it requires the source code of the system that may not be available. This demonstrates the difficulty of creating realistic synthetic workloads that is a goal this work hopes to undertake.

Assumptions, Limitations, and Delimitations

Assumptions

The main approach for synthetic file system benchmarking focuses only on generating file system calls on the underlying system. It does not take into consideration other portions of the system such as CPU and primary storage resources that can be consumed by the applications that would be generating those file system calls. Although applications and other processes can affect the performance of the file system by consuming shared resources such as the CPU and primary storage, file system benchmark studies tend to ignore these processes and focuses only on generating and executing file system commands.

Limitations

One limitation in this study would be the verification of the results across actual live workloads. Due to the unavailability of live workloads to trace this work will use other synthetic benchmarks to create a variety of workloads. One synthetic benchmark such as Postmark is designed to provide synthetic workloads that are similar to email servers. Other workloads can be generated by FileBench that can be configured to create workloads with different workload characteristics. In addition, a custom C application was built to provide an additional workload.

Delimitations

The work focused on specific environment namely the Linux kernel and used a single node to trace and run the synthetic workloads. The work was not performed on a network file system or a system with multiple nodes such as a storage area network or any other form of a distributed file system. In addition, the system used a hard disk drive and not a solid state drive.

Summary

This work presents a methodology for enhancing the accuracy of synthetic file system benchmarking. It will first present in the literature review section the current state of the art in synthetic file system benchmarks and discuss the issues with the current methodologies in the next section. The literature review will also highlight the importance of having accurate workloads when testing the file system performance. Then, the literature review will present other benchmarking methodologies and discuss their advantages and disadvantages over synthetic benchmarks, and finally it will conclude with a discussion of a synthetic disk workload generator that this work will base its approach on.

The methodology is described in the subsequent section and will contain the main steps that are needed to capture the trace, analyze and preprocess the trace, and generate the required configuration files for a tool called FileRunner that will generate the equivalent synthetic workload. Then a description on how the workloads are setup and run is delineated. Finally, this section ends with a discussion on how the results are evaluated.

Chapter 2

Brief Review of the Literature

Introduction

In this chapter a review of the literature pertaining to the problem and goals are presented. The chapter starts by summarizing the survey performed by Traeger, Zadok, Joukov, and Wright (2008) that describes the current state of benchmarking and the general problems that arise in file system benchmarking that includes the inaccuracy, inconsistency, and unrepeatable nature of the methodologies used in benchmarking.

Then the chapter delves into the literature specific to the problem this paper addressed that is the inaccuracy of synthetic file system benchmarking. The approach section of this document addresses the unrepeatability issue by describing in details the methodology that can be easily repeated. The inconsistency problem is addressed by applying the methodology across diverse workloads that further validates the approach.

This portion of the literature review consists of describing the approach that most benchmarking systems use such as Postmark and FileBench and how these approaches can lead to inaccurate workloads because they do not consider how the file system calls are being satisfied by the lower level function calls. Specifically, it consists of whether the requests are being served from primary or secondary storage. This section then continues to describe the work by Tarasov, Bhanage, Zadok, and Seltzer (2011) illustrates the need to match workloads by function calls as well.

One approach that addressed the lower level function call is called codeMRI that generates synthetic workloads that matches the live workload's API calls and function calls for increased accuracy, but suffers from being irreproducible and still requires some refinement. This work is presented next and concludes the literature review pertaining to the inaccuracy of synthetic benchmarks.

In the next subsection a discussion on another form of benchmarking called trace replay technique that generates workloads by replaying previously collected traces is presented. This method can present an accurate workload on the system based on the provided trace, but is incapable of adjusting the generated workload to provide further insight on the workload's bottlenecks and behavior under possible varying conditions. This inflexibility and unavailability of traces is the reason why synthetic benchmarks are preferred over trace replay.

In the following section a description of tracefs is presented. Tracefs is a tool that is used in the methodology of this work to collect file system traces. In the final subsection a discussion on another benchmarking technique called synthetic disk workloads (Tarasov, Kumar, Hildebrand, Povnzner, Kuenniehng, Zadok, 2012) is presented focusing on its methodology since it is extended in this work in the synthetic workload generation process.

General Problems in File System Benchmarking

In Traeger, Zadok, Joukov, and Wright (2008) the authors performed a nine year study that surveyed 415 file systems and file system benchmarks from 106 recent papers. Their study focused on assessing the benchmarking technique presented in each paper, the validity or completeness of the results, and the level of details present in the work when describing approach or the environment of which the benchmarks were done. To illustrate the extent of how misleading some benchmarks can be the authors devised an experiment that showed how application benchmarks can be ineffective in benchmarking file system. The chosen application benchmark was the compiler benchmark.

The experiment's approach consisted of modifying the performance of file system operations and then test whether the benchmark is capable of detecting any difference in performance. The modified operations were the read and write calls. Their performance was slowed down by inserting a loop in each method that performed nothing except delay the operations' execution time. In the first round of experiments the read operation was effectively delayed by a factor of 32. The OpenSSH program versions 3.5, 3.7, and 3.9 were compiled on two systems: the slowed file system and the original Ext2 file system. The results showed less than 1.5% slowdown in the modified file system which does not indicate any significant performance difference between the two systems. Similar results were obtained when other file system operations were slowed down. Since the benchmark results practically showed no substantial difference in the performance between these two systems one can deduce that the chosen benchmark is not a suitable benchmark for studying file system performance.

Experiments that Illustrate the Effect of Execution Paths on Performance

As discussed earlier synthetic workloads suffer from inaccuracies since they ignore function calls. This was illustrated in the experiment presented by Tarasov, Bhanage, Zadok, and Seltzer (2011). In that experiment the authors ran several workloads generated by the FileBench benchmark on the Ext2 file system where each workload read one file from disk repeatedly. The system RAM was 512MB and for the initial workload the file size was set to 64MB. The authors repeated the experiment varying only one parameter that is the file size. The file size was increased by 64MB for each run until the file size reached 1024MB. Each run lasted about 20 minutes, but only the last minute was reported to ensure proper system aging. There were a total number of 16 runs.

The results indicated that for all runs with file size 384MB or smaller the number of read operations was around 9,700 operations per second. For the remaining runs where the file size was 448MB or higher the performance was significantly less and averaged about 1,000 operations per second. The reason for the big drop in performance is attributed to disk I/O. For the runs where the file was small enough to fit in memory the first read was served from disk, but the subsequent reads were performed from memory. For the runs with file size 448MB or higher the files did not fit entirely in memory so for every read disk I/O was needed.

The authors performed additional experiments to identify at which point the significant drop in performance was experienced. The result was experienced in a smaller range of 6MB where file size was around 400MB. So in this case a variation of 1.5% of file size (6MB) has led to a drop in performance of about 90%. This experiment illustrates how the same API call can have a significant difference in performance even for a simple workload that produced the same number of API calls of the same file. The difference in performance is due to the different execution paths that satisfy the API calls.

In another study by Anderson et al. (2004) the authors show that the timing accuracy of the file system calls is also crucial in producing accurate workloads that can cause the response time to vary from +350% to -15%. They illustrate that mimicking file system requests such as maintaining the read/write ratio, request offset, and the order of the requests is insufficient for generating equivalent workloads. In their study it is shown that timing affects response time, queue length, and mean latency.

The CodeMRI Methodology

The codeMRI benchmark created in Agrawal, Dusseau, and Dusseau (2008) took into consideration requests that are satisfied from primary storage and secondary storage in order it provide more accurate workloads. So the resulting workload had a similar performance footprint as the original workload. The first step in this process involves tracing a live workload and recording the API calls and their corresponding function calls. The trace represents the execution profile of the live workload and the goal is to generate a synthetic workload with the same execution profile.

After the trace is completed the workload is divided into micro-workloads where each micro-workload contains the execution profile of one API call. For instance, for the read API call the micro-workload or the execution path that resulted from the read API calls are extracted from the overall trace. Then for each micro-workload a predictor set is generated that describes the execution profile (micro-profile) of the micro-workload in a concise manner. The purpose of the predictor set is to eliminate the need to maintain the entire micro-profile by using a subset of it in a manner that maintains the characteristics of the micro-profile. From the predictor set synthetic micro-workloads can then be generated with an execution profile that matches the extracted live micro-workloads. The final step consists of combining the synthetic workloads to form the final synthetic workload that is supposed to be equivalent to the original workload.

The paper does not discuss any details on how the predictor set is generated except that three metrics play a role in the generation process. The three metrics are slope, uniqueness, and stability. Slope is defined as the change in the number of function calls as one workload parameter is changed. For example if the workload parameter is file size the number of function calls is recorded as the file size is varied. The uniqueness metric signifies how closely the predictor set represents the real workload. It is measured from 0 to 1 and a uniqueness of 1 signifies that a predictor set is the only workload that is being executed during the runtime of the workload. This means that the micro-workload is fully representative of the real workload while a uniqueness of 0 means it's not related to the workload at all. Stability is defined as the variability in the slope as some workload parameter is changed or as the rate of change of the slope as a workload parameter is changed.

The authors claim that it is fairly easy to find a predictor set for a micro-workload with a uniqueness close to 1. However, for a workload consisting of several micro-workloads it is more difficult to find a predictor set with a value close to 1 since some function calls would be overlapped from several micro-workloads. In other words, it may not be clear which function call belongs to which micro-workload. In addition, the authors fail to describe how the workload parameters are varied to calculate these metrics

To overcome the overlap of function calls a linear programming algorithm is used to select predictor sets that maximize the uniqueness of each micro-workload. Linear programming is usually used in similar situations to determine the maximum or lowest outcome that can be calculated over a linear objective function. In this case, it assigns function calls to API calls in a way that would produce the greatest overall uniqueness value. Unfortunately, the work in (Agrawal, Arpaci-Dusseau, & Arpaci-Dusseau, 2008) does not provide sufficient information on how linear programming was used to derive the predictor sets.

From the predictor sets synthetic micro-workloads are generated, again no information is presented on how the synthetic micro-workloads are generated from these predictor sets nor does it present the methodology that synthesizes the synthetic micro-workloads into the final synthetic workload. However, they do mention that timing and ordering are built into the workload generation process.

Based on experiments performed by the authors the methodology presented in this paper produces accurate micro-workloads from the predictor set, and are able to construct the micro-workloads into equivalent macro-workloads. However, the authors do discuss that the micro-workload synthesis process still requires some refinement for complex workloads. In addition, the source code of the application is required for the analysis stage. Overall, the work does not provide sufficient detail to generate any portion of their methodology such as the process for creating micro-profiles or the process to generate micro-workloads from the predictor set. Even though the work presented in (Agrawal, Arpaci-Dusseau, & Arpaci-Dusseau, 2008) does not present their methodology in a reusable manner it does highlight the need for synthetic workloads to consider execution paths in their generation process.

Trace Replay Techniques

In addition to synthetic workloads there are other benchmarking approaches. One approach consists of capturing the application's traces and then replaying them (Joukov, Wong, & Zadok, 2005). In trace replay techniques that replay traces are captured at the VFS level are suitable for file system benchmarks since they capture all file system requests. Replayfs is a trace replay system that captures traces and replays them at the VFS level. Traces are captured using a stackable file system called tracefs. Each VFS operation in the trace is preprocessed by the trace compiler component so that they are playable by the trace replayer component. Each operation has a process ID, a timestamp, the parameters of the operation, and the return value. The trace replayer prefetches the processed commands sequentially, places those commands in a buffer, and then replays them onto the VFS.

The system uses a resource allocation table that contains references to the objects that are in primary storage. This allows the system to access those objects that are in memory. For example, if a file is being read from memory the location of the file needs to be known in order to be able to read it again. Tracefs stores that information and it populates into the resource allocation table during runtime.

To increase accuracy replayfs creates the same number of threads to replay requests in a similar manner and at the same time as they occurred in the live trace. This will reproduce resource contention in the replay that includes disk seek, locks, and semaphores. Threads are reused to conserve resources and a pre-spin technique is employed to ensure timing accuracy. In a pre-spin the timers are set 1ms before the event will occur so that the request is replayed at the exact moment eliminating any delays in timer delays. For further CPU and memory resource savings the data is not copied into user space from the VFS level.

In conclusion, despite the fact that tracing techniques might provide accurate workloads their inflexibility as described in the problem section and the unavailability of traces in some cases assert that another benchmarking approach is needed to fulfill the gaps and shortfalls of trace replay techniques. Synthetic workloads provide the flexibility that is needed to perform performance analysis on varied workloads (Joshi, Eeckhout, Bell, & John, 2008). Synthetic workloads are also preferred because they are less complex and have a lower setup cost (Anderson et al., 2004).

Synthetic Disk Workloads

The work in Taraseov et al. (2012) presented a synthetic disk workload that the methodology of this work is going to be based upon. In this approach disk traces are used to create a statistical model that represents the characteristics of the workflow that is used to create the configuration files for file benchmarks such as FileBench, IOzone, or Iometer. The details of this methodology and the statistical model are presented next.

The first part of the approach extracts statistical information from the disk traces for each file system dimension such as the number of reads and writes in a trace, file colocation, or file sizes. From the statistical data a function is created for each dimension called the feature function that computes the values that the dimension will have in the synthetic workload. So the feature function for the file size dimension will return the file size of the next API call. This function is called again to retrieve the file size of the following API call.

This is repeated for every dimension resulting in a multidimensional array or matrix that contains the data points for each of those dimensions. In figure 1 below the three dimensions considered are: inter-arrival distance, I/O size, and operation (read or write). So as can be seen the trace contained 52 requests of size [0 – 4KB] and inter-arrival distance less than 1KB, and 14 of those were writes and 38 were reads. Those data points are represented in a matrix that is used to generate the disk operations with the corresponding values.

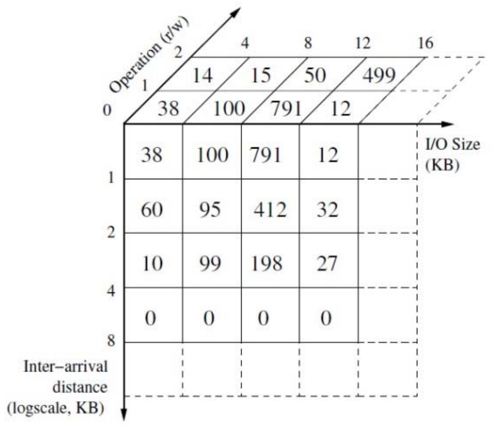

Figure 1. The n-dimensional matrix statistical model.

The authors present that the workflow characteristics change over time and that workload generators are not capable of modeling this change. For instance, a workload may exhibit a different read/write ratio across time or the I/O size may vary as the workload progresses. To address this issue the authors devised the workload chunking technique. Workload chunking is the process of dividing the workload into smaller chunks across time where the workload characteristics are assumed to be stable within that chunk. The reason behind the chunking is to be able to model the variations of the workflow.

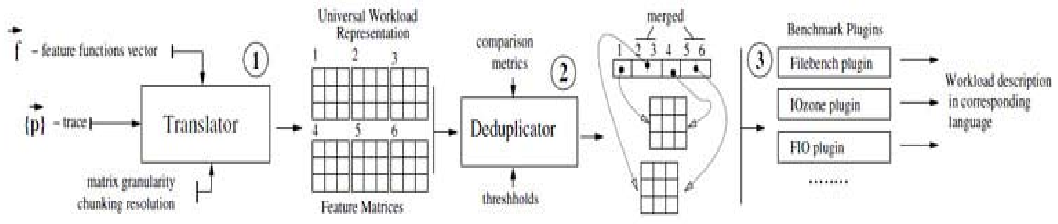
f - feature functions vector
{p} - trace
matrix granularity chunking resolution
Translator
Universal Workload Representation
Feature Matrices
comparison metrics
Deduplicator
threshholds
merged
Benchmark Plugins
Filebench plugin
IOzone plugin
FIO plugin
Workload description in corresponding language

Figure 2. Overall system design for synthesizing disk workloads from traces.

FileBench is then configured to run workloads for each chunk. These synthetic workloads are the run in succession resulting in a final synthetic workload that matches the original workload. The chunk size was determined experimentally and is discussed in further details in chapter 3.

Some chunks for a given workload may exhibit the same workload characteristics. So the same chunk data can be used for two or more intervals. This can reduce the amount of overall data needed. After all the statistical matrices are computed they are compared with every other statistical matrix. If 2 matrices are found to be similar the values in the two matrices are averaged and the 2 chunks then refer to this averaged matrix. This process is shown in figure 2.

Tracefs

Tracefs is a stackable file system for the Linux operating system that hooks into the virtual file system (VFS) and implements all the file system operations. By intercepting the file system operations tracefs is capable of logging the VFS calls and their parameters along with the timestamp and process information. To understand how tracefs works a short description of the Linux File System (LFS) is presented that is followed by a description of tracefs.

Linux File System

The (LFS) separates the implementation of the file system and the kernel via an API called the VFS API (Konishi, Amagai, & Sato, 2006). A file system can be mounted to any location in the file system hierarchy even while the kernel is running. Once the file system is mounted then any file system call made to that path is forwarded to the file system mounted at that location.

The LFS has four main objects that are *superblock*, *inodes*, *files, dentries* (directories entries), and *superblock*. Each file on disk in Linux is represented by an *inode* data structure that stores all the information about the file. When a file is loaded in memory by a process the process information including the state of the file is found in the *file* data structure. This is an in-memory only data structure. *Dentry* objects are used to link *inodes* with their parent directory. The *dentry* cache is used to quickly find an *inode* given a path. Finally, the *superblock* object contains the information about the mounted file system.

```
struct file_operations {

loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char _user *, size_t, loff_t *);
}
```

Figure 3. Portion of the File operations struct.

Each data structure has a corresponding “operations” structure that contains a list of pointer functions. It is those operations that comprise the VFS API. For example, the *file* data structure has a corresponding *file_operations* data structure that defines the operations that can be performed on the *file* object. A portion of the *file_operations* struct is presented in figure 3 below. The llseek, read, and write operations are three of the available operations for the file object. The mounted file system does not need to implement all operations for all file system objects if it does not need to support those operations. However, some operations such as the mount and unmount operations for the *superblock* object are required since the file system cannot be mounted without these operations.

How Tracefs Works

Tracefs is a stackable pass-through file system (Aranya, Wright, & Zadok, 2004). It is part of the file system translator (FiST) project (FiST: Stackable File System Language, 2014). A stackable file system such as tracefs uses two locations in the file system. Usually one is called upper and the other lower. Sometimes the lower location is referred to as hidden. Tracefs mounts itself to a configurable predefined location in the file system (the upper location), and implements the VFS API operations. So every file system call to the upper location is received by the VFS that in turn routes the call to tracefs, since it is the file system that is mounted at that location.

Once a system call is received by tracefs it logs that call and its parameters, and then it makes the same exact call on the lower location. This lower path is part of another file system that is usually the default file system or the file system under study. Then tracefs captures the return value for the call it made to the lower path and it forwards that return value to the original caller. So in essence tracefs receives a system call, logs it, then forwards it to the “actual” file system, receives the reply from the file system, and finally forwards it to the caller. Figure 4 shows the structure of tracefs relative to the kernel and user space and figure 5 shows a snippet of code for the write VFS call named *tracefs_write*. Additional comments have been added to the code to describe what each portion of the code is performing.

For example, if tracefs is mounted to /root/upper and there is a program that made a system call to create the file at this path: /root/upper/newfile.txt. The system call goes to the VFS that determines that tracefs is the mounted file system at that path and calls the VFS create function implemented by tracefs. Tracefs then logs that call and then issues a VFS create file call to the lower path: /mnt/lower/newfile.txt. The virtual file system determines that the default file system is mounted at that path and calls the create function for the default file system that creates the file at the lower path. Tracefs then receives the return value from the default file system, and then logs it and returns it to the original caller.

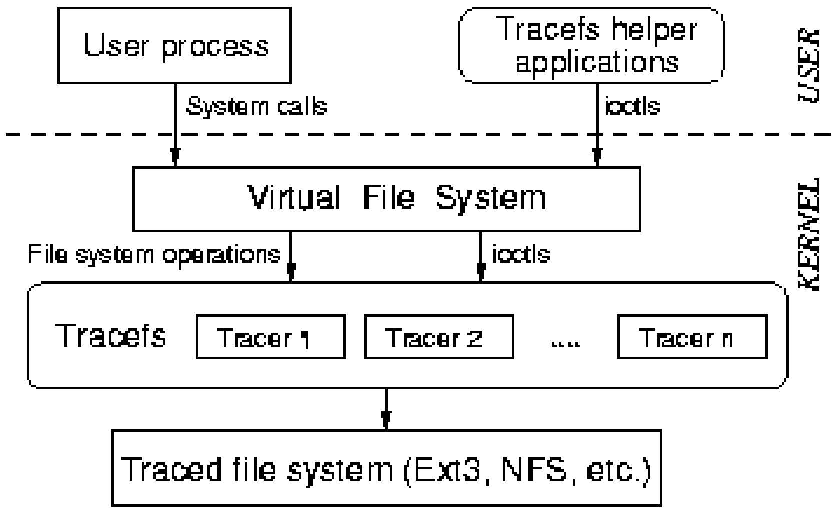

Figure 4. Tracefs structure.

```
ssize_t
tracefs_write(upper_file)
{
        //log this call and the file parameters
        trace_write_pre(…);
        //Get the lower file to perform the write operation on
        VFS_write(lower_file);
        //log the return result for this call
        trace_write_post(…);
        return;
}
```

Figure 5.General structure of a VFS method.

Similarly, if a command is issued to read that same file: /root/upper/newfile.txt, the VFS will determine that tracefs is the mounted file system so it will invoke tracefs's read VFS call.

Tracefs will log that call, issue the same command to the lower location, wait for the return value, log the return value, and then finally forward the return value to the original caller.

The files in the upper location do not physically exist on dsik. They refer to the files in the lower location. Tracefs does not actually implement the drivers needed to access disk such as the case with common file system like ext3 and ext4. Instead it captures the calls made to the upper file system and forwards them to an existing file system by issuing the equivalent call on the lower path.

Summary

The literature discussed in this section presented that it is not sufficient to only consider the system calls when generating synthetic workloads. This is because current synthetic benchmarks do not consider the effects of cache on performance (Traeger, Zadok, Joukov, & Wright, 2008, Tarasov, Bhanage, Zadok, & Seltzer, 2011). One approach that addresses the cache issue is called codeMRI. However, codeMRI has some pitfalls such as requiring the source of the application being benchmarked. In addition, the authors state that the work still requires some additional improvements.

Trace replay techniques can offer accurate replay of workloads but require traces in order to generate the workload and are highly inflexible since it is not possible to modify one or more workload parameters. This inflexibility does not permit the study of the workload's performance under slightly varying conditions.

In the next chapter a synthetic workload methodology is presented that consists of tracing an original workload using a tool called tracefs, parsing the trace using a tool created by this work that generates the configuration file for another tool created by this work called FileRunner that generates the equivalent synthetic workload. The variables in the configuration files can then be easily adjusted to generate similar synthetic workloads, but with one or more varying workload parameter.

Chapter 3

Methodology

Overview

The work described in this chapter follows a similar approach to the one outlined in Tarasov et al. (2012). The work in Tarasov et al. (2012) generated synthetic disk workloads from disk traces. As part of their process the authors created the chunking mechanism to model workload variation and was used in this work for the same purpose. The high level approach for this work is also similar to the high level approach in Tarasov et al. (2012) and consists of the steps shown in figure 6.

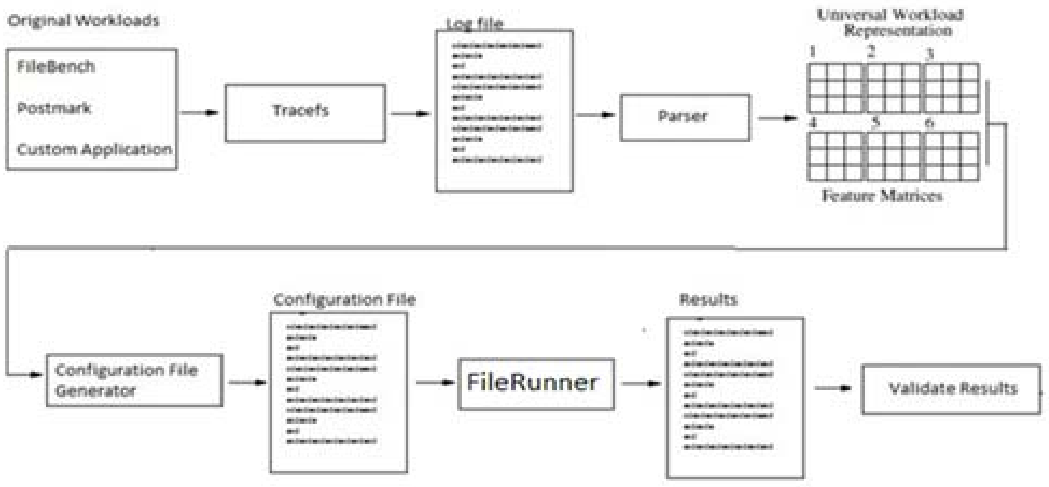

Figure 6. Overall system design.

The work presented in this chapter differs from the work in Tarasov et al. (2012) because it captures the traces at the VFS level instead of disk level. This means that there is a difference in the data being captured. Specifically, the VFS operations and their parameters were captured instead of the disk level operations and their parameters. This work also captured the operations that were satisfied from the cache and not just from disk. Another difference in this work is how the trace points were used to generate the configuration files for the workload generation tool created by this work called FileRunner.

The details of each step of the proposed approach are presented in the next subsections along with details about the individual tools that are used in this approach. The first two subsections discuss the purpose of using workload chunking across time and I/O size and the approach for choosing the proper chunk sizes. At this point in the methodology the chunk sizes are determined and used in the remainder of the process.

The next subsection discusses the next step of the approach that describes the tools that are going to be used to generate the workloads that are considered the original workloads. These are the workloads that are being traced and the ones that this methodology is trying to regenerate synthetically.

The following two subsections discuss the changes that are needed for tracefs to become compatible with the kernel version 3.2 and the changes needed for tracefs to capture the cache information. The latest version of tracefs was built for an older kernel version and is not compatible with current kernel versions. This section also includes a brief description of the VFS API and data structures that facilitates the understanding of how tracefs works and the changes that are needed to port tracefs to the Linux kernel version 3.2.

Once tracefs is setup to work on kernel version 3.2 it is loaded into the kernel and then an original workload is run. Tracefs will generate a binary log file that is then converted into its textual format by a tool that comes with tracefs. This work has created a tool that parses the textual format of the trace into a data structure array called *operations* shown in appendix A.

Each entry in the *operations* array corresponds to one VFS call in the trace and contains all the information about that call.

Once the traces have been parsed into the *operations* array, as the process outline above shows, the next step is to process the data in the array into the statistical matrix that will be used to generate the configuration files. This requires setting up feature extraction functions that will convert the data in the *operations* array into indices of the statistical matrix. As an example, a process ID of value 323 will correspond to index 1 in the process dimension of the statistical matrix. Further details on the construction of the feature extraction functions are provided in the next subsection.

The next section describes the process of generating the configuration files for the synthetic load generation tool called FileRunner from the statistical matrix followed by a section that describes how FileRunner works. FileRunner is then run to produce the synthetic workload. At this point both traces for the original and synthetic workloads are available and the next step consists of the verification process that was used to validate this work and is described in detail. This includes the list of metrics used to compare the original workload and the synthetic workload generated by this work, and how these metrics are implemented in the workload generation process. In the following section a description of the resources being used and the environment where the work was generated and validated is described.

Workload Chunking

The work in Tarasov et al. (2012) realized that some workloads are not uniform and exhibit some variation across time. Workload variation is any change in the workload's behavior that is indicated by a varying parameter of the workload. For example, the read/write ratio of the workload may change over time or the file size of the VFS calls can vary as the workload progresses. Workload variations exist especially if the traces were collected over a long period of time such as hours or days (Tarasov et al., 2012). The next subsection describes why chunking is needed to model the variation followed by a section that shows how to determine the chunk size.

Why Workload Chunking Works

To generate an accurate synthetic workload the variation of the original workload needs to be incorporated in the synthetic workload. One approach to modeling the variation is by dividing the workload into smaller intervals that are small enough that the workload in that interval can be considered uniform (Tarasov et al., 2012), and then creating equivalent workloads for each interval and running them in succession. However, if the chunk size is too small it would become similar to the trace replay technique and it loses the flexibility that comes with synthetic benchmarks and would require a larger configuration file and data set.

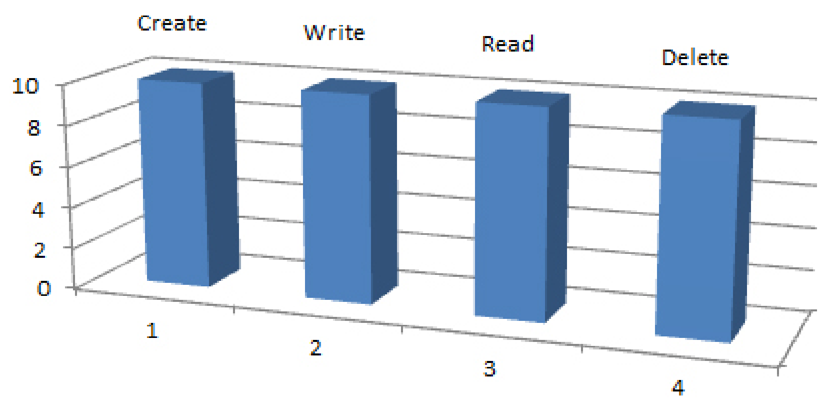

Figure 7. Sample workload.

Figure 7 shows an example of how this chunking technique can model the variation in a workload. The figure shows one workload that starts with creating 10 files, then the workload performs 10 write operations on those files, followed by ten read operations on those files, and finally the workload deletes those files. Without considering chunking the statistical matrix in this case would show that there are 10 operations for each of the four VFS calls. The benchmark may be configured to perform 10 iterations of creating a file, writing to it, then reading it and finally deleting it as opposed to having all of the create operations executed, followed by the write operations, then the read operations, and finally followed by the delete operations.

Dividing the workload into smaller intervals and then generating the synthetic workload based on these intervals would result in a more accurate workload. Figure 8 shows the same workload divided into 4 chunks. Each chunk contains the 10 operations for a given VFS call. The benchmark can now be configured to run 4 workloads in succession starting with the workload that creates 10 files, followed by the workload that will perform 10 writes, followed by the read workload, and lastly the workload that will delete the files. In this manner the result is a synthetic workload that matches the variation of the original workload.

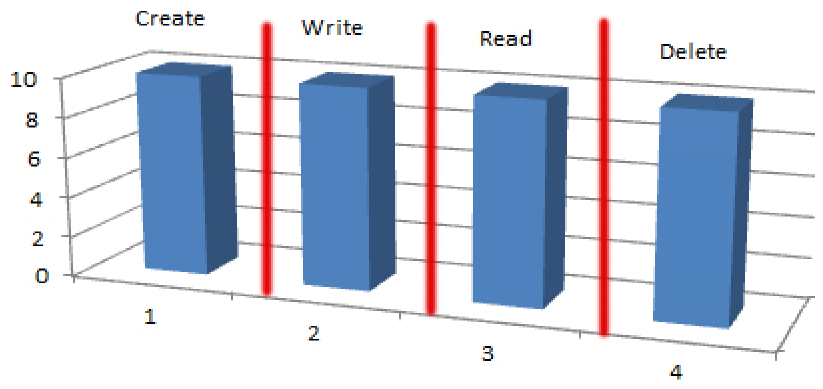

Figure 8. Workload chunking.

Determining the Proper Chunk Size

The approach used by Tarasov et al. (2012) to determine the optimal chunk size is to start by selecting a large chunk size. The chosen starting chunk size was 100 seconds that was considered large enough starting point. The trace is then divided into 100 seconds intervals. For each interval the statistical matrix is computed. Then for each statistical matrix the configuration file is generated. FileRunner is then setup to run the configuration files in sequential order to generate the synthetic workload based on the original workload characteristics. The performance metrics of the new synthetic workload is compared to the performance metrics of the original workload as outlined in the next section. If the difference in the metrics is less than 10% then the two workloads are considered equivalent and the chunk size is considered appropriate. Otherwise the chunk size is reduced and the process is repeated until an equivalent synthetic workload is produced.

I/O Size Chunking

To create a synthetic workload with a similar footprint as the original workload the synthetic workload needs to issue I/O operations with similar I/O size as the original workload. Thus, I/O chunking is the process defined in this work that enables the specification of the size of I/O operations. It is similar to the process of workload chunking by time. It enables modeling the workload's I/O size variation without the need to track the I/O size for every operation.

Determining the I/O chunk size is experimental as is the case for time chunks. Just like the time chunk the I/O chunk size is initially set at an arbitrary large I/O size of 1MB. The configuration files are generated with this setting and the synthetic workload is run. The total I/O size of the original workload and the synthetic workload are compared and if the difference is less than 10% the chunk size is deemed appropriate. Otherwise the I/O chunk size is decreased and the process is repeated with this new chunk size.

The chunk size in the parser is represented by an integer variable called *iochunksize* and specifies the size of the chunk in bytes. As the parser is going through the trace it extracts the value of the size parameter for a given operation and then divides that value by *iochunksize*. This new value would be the index in the statistical matrix for the I/O dimension. This index is used by the FileRunner when it is generating the file system operation and needs to determine the I/O size for that operation. The size is determined by the formula shown in equation 1.

$$\frac{(iochunksize * iondex) + iochunksize \ * (iondex + 1)}{2}$$

Equation 1. Formula for determining I/O size from I/O index.

The following example illustrates how reducing the chunk size produces a more accurate I/O footprint. Even though the process starts with 1MB and gradually reduces the I/O chunk size until the desired accuracy is achieved this example chooses the values 1000 and 100 as the two I/O chunk sizes to illustrate how the reduction of the I/O chunk size affects the I/O footprint. So if the size of an operation in the original workload is 1800 bytes and the value of *iochunksize* in the parser is set as 1000 bytes then the parser would calculate the I/O index to 1800 / 1000 = 1. So when FileRunner needs to generate the I/O size from the I/O index it uses the formula presented in equation 1 and that would result in a value of (1000 * 1 + 1000 * (1+1)) / 2 = 1500 bytes. Reducing the value of *iochunksize* in the parser to 100 would result in the index of 1800 / 100 = 18. Then FileRunner using the formula in equation 1 would result in an operation with an I/O of (100 * 18 + 100 * (18+1)) / 2 = 1850 bytes.

Setting up Benchmarks to Generate the Original Traces

While FileRunner was used to recreate the workloads from the given traces, other methods are needed to generate original workloads to produce those traces. It is also preferred to produce a variety of original workloads with varied workload characteristics to ensure that the methodology proposed in this work can produce equivalent workloads to these generated traces.

One approach for generating an original workload is to replay traces that are collected from live servers (Anderson et al., 2004, Tarasov et al., 2012). This work used another approach that created original workloads using synthetic benchmarks. This approach is commonly used among studying file system performance (Traeger, Zadok, Joukov, & Wright, 2008). The survey on file system benchmarking published by Traeger, Zadok, Joukov, and Wright (2008) states that 29 of the surveyed papers used Postmark to generate the synthetic workloads. Postmark creates workloads that are similar to that of mail and ecommerce servers. Therefore, Postmark was used in this work to generate an original workload using its default configuration. The default configuration generates files ranging from 512 bytes to 10,240 bytes. The average I/O of each operation is 4,096 bytes. Postmark issues the workload on the directory that it is launched in. So in the terminal a command was issued to change the directory to the directory where the tracefs logs the file system calls and Postmark is launched. Once launched the number of transactions was set as 3,500,000 and then the run command was issued. This produced a workload that ran approximately 5 minutes with 1,748,488 create and delete operations, 5,111,815 write operations, and 3,890,374 read operations.

In addition to using Postmark this work used FileBench that is also used in various works to study file system performance (Kaiser, Meister, Hartung, & Brinkmann, 2012, Ahmad, 2007). A FileBench run consists of a setup stage and a run stage. In the setup stage the initial files and file sets are created. This is followed by the run stage that creates the workload. For the purposes of this work FileBench was modified to run a script that issues a command to tracefs to start tracing file system operations. This occurred after the setup stage and before the run stage. This way the collected traces did not capture the file system operations of the setup stage.

FileBench has various workload personalities that are provided with the FileBench installation. This work used three of these workload personalities to generate 3 original workloads. The three personalities include workloads that mimic a web proxy, a file server, and finally a web server.

The web proxy workload was setup to run for 300 seconds and to target the same directory that tracefs logs the file system calls. Appendix F shows the complete workload definition that defines 5 threads to execute the following operations in sequence: delete file, create new file, write to that new file, and then read it five consecutive times.

The web server personality was setup to run for 300 seconds and to target the same directory that tracefs logs the file system calls. Appendix G shows the complete workload definition that starts by creating 1000 files with content and one empty file that is referred to as the log file. Then 5 threads are created to execute the workload. Each thread picks a file from the initial 1000 files and then performs a read on that file 10 times in a row. Finally, the thread would append a variable sized text into the log file. The write mean size is 16kb.

The file server workload was setup to run for 300 seconds and to target the same directory that tracefs logs the file system calls. Appendix H shows the complete workload definition that starts by creating 1000 files and allocating content for 80 percent of those files. Then 5 threads are created to execute the workload. Each thread first creates a new file and adds it to the file set. Then it writes to that file. Next, each thread picks another file from the file set and appends more content to it. They then perform a read on that file. Finally, the thread would delete that file.

The different Postmark and FileBench workloads will have different workload characteristics, but their workloads will be uniform as they are run. In other words, there will be no variation in the workload as it is executing. To validate the work against modeling workload variation a C application was created by this work to create an original workload with varying workload characteristics. Its workload consisted of creating a number of files, write to those files, then read those files, and finally delete those files. The application had global parameters that made it possible to configure the number of files to create, the average file size, the average I/O, and the number of I/O operations. Although the C application produced a workload that may not correspond to a live workload it provided an original workload that varied across time and that was used in the validation process to validate workload variation modeling.

The different methods described above generated a variety of original workloads that were used in the validation process. The overall evaluation section describes the context for the use of these workloads and their role in the validation of the methodology.

Setting up Tracefs to Capture Traces

Porting Tracefs

The latest version of tracefs was released in 2007 for the Linux kernel version 2.6.17.13 and is not compatible with the current kernel releases. One of the issues is that the VFS operations in current kernel versions have different signatures than the ones in the kernel version 2.6.17.13. This caused compilation issues for tracefs. Figure 9 shows a signature example of a VFS operation for kernel version 2.6.17.13 and kernel version 3.2.0-68-generic. These changes caused compilation issues and needed to be addressed. Another issue with the latest version of tracefs was that it used deprecated functions. For example, the read VFS call in tracefs used the deprecated method *generic_file_read*.

```
fsync(file_t *file, dentry_t *dentry, int datasync)
fsync(file_t *file, loff_t start, loff_t end, int datasync)
```

Figure 9. Difference between VFS method signatures between two kernel versions.

To address the compilation issue due to changes in the VFS API another tool called wrapfs was used. Wrapfs is a stackable file system just like tracefs except it has no tracing functionality. Wrapfs is maintained as part of the kernel so it is updated as the kernel changes. Similar to tracefs, wrapfs intercepts the VFS API calls, forwards them to the lower file system, retrieves the response, and forwards the return result back to the original caller.

Figure 10 shows the *fsync* VFS call for both tracefs and wrapfs. It demonstrates the two issues discussed above. The first issue is the different method signatures and the second issue is the use of a deprecated function `Fsyn_on_lower_file_old` in tracefs while wrapfs uses the new function `Fsyn_on_lower_file_new`. In addition, the figure shows that the remaining difference between the two is the tracing code that only exists in tracefs and consists of two method calls `Log_call_and_parameters` and `Log_return_value`.

```
wrapfs_fsync(file_t *file, dentry_t *dentry, int datasync)
{
        Determine_lower_file();
        Fsyn_on_lower_file_new();
}
tracefs_fsync(file_t *file, loff_t start, loff_t end, int datasync)
{
        Log_call_and_parameters(…);
        Determine_lower_file();
        Fsyn_on_lower_file_old();
        Log_return_value(…);
        return;
}
```

Figure 10. Fsync method structure for wrapfs and tracefs.

```
wrapfs_fsync(file_t *file, dentry_t *dentry, int datasync)
{
        Log_call_and_parameters(…);
        Determine_lower_file();
        Fsyn_on_lower_file();
        Log_return_value(…);
        return;
}
```

Figure 11. Final working version of fsync method with tracing functionality

To produce a working version of any given tracefs VFS call the tracing code was inserted into the corresponding wrapfs VFS call. The result of this process on the *fsync* function is shown in figure 11. This method resolved the compilation issue due to the invalid method signatures. It also resolved the issue of using deprecated methods since these methods are updated in wrapfs. The result is a working version of tracefs in current kernels.

Capturing Cached Operations

The work in Agrawal, Dusseau, and Dusseau (2008) and Tarasov, Bhanage, Zadok, and Seltzer (2011) stated that the different lower level function calls that satisfy a given system call have a different performance footprint. For instance, system calls that are satisfied by the cache perform faster than the system calls satisfied from disk. Chapter 2 discussed the experiment in Tarasov, Bhanage, Zadok, and Seltzer (2011) that illustrated the effect of cache on the workload. The result was a speedup in performance by a significant factor.

In order to generate representative workloads this work generated synthetic workloads with the same footprint as the original workload. This entailed accounting for the cache. So it was important to be able to determine from the traces the VFS calls that are serviced from cache and the calls that are serviced from disk. This information is then used in the workload generation process that is discussed in subsequent sections.

To capture the operations that are satisfied from the cache tracefs was setup to use the address space operations that are provided in the kernel. The address space operations map a physical file that is on disk to memory. Tracefs has an option that can be supplied when mounting it to specify whether to use memory mapped operations or not, but these operations are not compatible with the kernel version 3.2. Thus the address space operations in tracefs were re-implemented to be compatible with the kernel version 3.2.

Without using the address space operations when a read system call is issued the tracefs VFS read method issues a *vfs_read* call on the lower file. The underlying file system handles the call that fetches the file from cache or from disk. This information is not available for tracefs. When using memory mapped operations the read method in tracefs will replace the *vfs_read* call to the lower file system with the generic read function called *do_sync_read* that is provided by the kernel. If the file is in the cache the *do_sync_read* method returns the data from the cache, otherwise it issues a call to the VFS address space operation *readpage*.

The *readpage* method is part of tracefs and contains tracing code just like the other VFS operations in tracefs. This method determines the lower file and then issues a *vfs_read* call on it. It also maps the data read from disk that is returned from the *vfs_read* call on the lower file into a page in memory. The result is when a system call is issued on a file for the first time the *vfs_read* method calls the *do_sync_read* method that then issues a call to the *readpage* method to retrieve the file from disk. Both of these VFS calls contain tracing code so they will be present in the trace. The second time a read call is issued on the same file and the file is still in the cache the *vfs_read* method will call the *do_sync_read* method that will retrieve the call from cache and does not call the *readpage* method. So the lack of a *readpage* in the trace signifies a cache hit.

Running and Tracing the Workload

At this point the tracefs module was ready to be compiled and inserted as a module into the kernel to start capturing the traces. Tracefs takes two configuration parameters that correspond to the upper and lower directories. The upper directory is the directory that tracefs intercepts the calls, logs them, and then forwards the calls to the lower directory.

After the tracefs module has been inserted into the kernel an original workload was then run. The original workload was configured to run on the upper directory so that all of its calls are traced. As the workload is run tracefs logs the calls into a file at a preconfigured location. Once the workload terminated a command was issued to tracefs to stop tracing and finalize the log file.

Parsing the Log File and Extracting the Features from the Raw Data

The data in the log files generated by tracefs are in binary format. For performance reasons tracefs does not convert the data being logged into their textual representation. For example, an integer value of 123 does not have the same value as the text ‘123’. A tool that comes with tracefs, the log parser, parses the binary log file and produces the textual equivalent of the binary data. So the log parser converts the binary value 123 to the textual value ‘123’. Figure 12 shows the output of the log parser for a *readpage* VFS call along with the call’s return value.

The first line of every file operation starts with the word BEGIN and contains the type of the file operation, OP_READPAGE or the *readpage* call in this case. The line also dictates whether this is part of the call (PRE_OP) or the return value (POST_OP). The last line for a given call starts with the word END. Every line in between contains some information about a parameter for the call.

```
BEGIN: OP_READPAGE PRE_OP
       UTime: 1393967326, 579074
       INum: 536360
       File name: /tmp/mmap.c
       PID: 18213
END: OP_READPAGE PRE_OP
BEGIN: OP_READPAGE POST_OP
       UTime: 1393967326, 579531
       Return = 0
END: OP_READPAGE POST_OP
```

Figure 12. Trace output for read VFS call.

In figure 12 the second line contains the information about the time of the call (UTime). Subsequent lines show that the operation is being performed on the file *mmap.c*, and the full path of the file is */tmp/mmap.c*. Other relevant information includes the process ID that performed the call, and the data that is read from disk.

The data in the trace file was used to populate the statistical matrix that was then used to generate the configuration files for FileRunner. The matrix is a six dimensional array of integers that are the time chunk dimension, process dimension, VFS call dimension, I/O size dimension, directory depth dimension, and the cached dimension.

The format of the data in the traces does not correspond directly to indices in the matrix. For instance, figure 12 shows that the time of the *readpage* operation occurred on “1393967326, 579531”. This value has to be converted into an index along the time chunk dimension. This conversion is referred to as feature extraction. Similarly, the process ID for the *readpage* call would be converted to an index in the process dimension. For some dimensions all the data needs to be parsed and readily available in order to calculate the index. This is required for the cached dimension. For the other dimensions the index value can be calculated as the data is parsed. The feature functions are discussed next followed by a description of the tool that parses the trace file generated by tracefs. Finally this section ends with a discussion on the feature function for the cached dimension.

Feature Functions

The feature function for the time chunk dimension uses the number of chunks variable and the time of the first operation in an equation shown in equation 2. ‘t’ refers to the timestamp of the current operation and ‘t0’ refers to the timestamp of the first operation.

$$\frac{\mathrm{t - t0}}{\text{\# of chunks}}$$

Equation 2. Feature function equation for the time chunk dimension.

$$\frac{\text{IO size}}{iochunksize}$$

Equation 3. Feature function equation for the I/O chunk dimension.

The feature function for the I/O dimension is similar to the time dimension feature function. It divides the I/O size found in the trace by the preset I/O chunk value as shown in equation 3. The VFS call feature function uses VFS call name and searches a predetermined linked list for a matching string and then returns the corresponding number/index. Appendix B shows the operations and their corresponding index value. So for the *readpage* operation in figure 12 the feature function returns the value 46. The feature function for directory depth counts the number of '/' characters in the file path, subtracts 1, and then returns that number. So the string '/tmp/dir/file' returns the value 2.

The feature function for the process ID maintains a linked list of (process ID, index) pairs and an index counter variable. The index counter variable starts with the value 0. Whenever a process ID is encountered from the trace the feature function checks to see whether the process ID exists in the linked list. If it exists then the index value of the linked list entry is returned. Otherwise, a new entry is added to the linked list that consists of the process ID and the current index counter. The value of the new index is returned. Afterwards, the index counter is incremented by one. So the first process ID gets assigned the index 0 and the following process ID gets assigned the index 1 and so on.

The feature function for the cached dimension requires all the data to be parsed. The next section describes the process of parsing the trace file followed by the description of the feature function for the cached dimension.

Parsing the Log File

This work created a tool that parses the file generated by the log parser and for each operation found in the trace an entry is added to the *operations* array. The details of the *operation* data structure is shown in appendix A. The tool reads the file line by line and when it encounters a line that starts with the word BEGIN it will create a temporary *operation* variable. That line will also contain the VFS call. Figure 12 shows a line starting with the word BEGIN and the operation is the *readpage* method.

The tool then parses the following lines and for every line the tool will first determine the type of data that line contains by string matching the beginning of the line. Then it will extract the value in that line into the corresponding entry in the temporary *operation* variable. For example, the first line after the BEGIN shown in figure 12 contains the string UTime. So the tool will extract the value “1393967326, 579531” into the *timestamps* variable of the temporary *operation* variable.

The tool will parse the following lines until it encounters a line with the word END. Once END is encountered it will add the temporary *operation* variable into the *operations* array. The parsing continues until the end of the trace file is reached. The result of the tool is an array containing *operation*, hereon called the *operations* array. Each entry in the operations array corresponds to an entry in the log file that is one VFS call.

Determining the Cached Operations

In the log parsing step above the cached value for the *read* operations is always set to one (i.e. cached). The proper value is determined after parsing is complete when all the operations are parsed. This is because a *read* call is satisfied from cache only if no *readpage* call for the same file exists. If a *readpage* call for the same file exists then the *read* operation was satisfied from disk and the cached value for the *read* operation needs to be set to 0.

This process for determining the cached value maintains a linked list consisting of entries that are comprised of the index of the *read* in the *operations* array and the file’s *inode* number. The *inode* number uniquely identifies a file.

So after the trace file has been parsed into the *operations* array a loop is performed on that array. For every iteration it will check to see if the operation is a *read* call or a *readpage* call. If it is a *read* call it will first check to see if the *inode* number exists in the linked list. If it does not it adds a new entry with the *inode* number and the current index of the *read* operation.

When the loop encounters a *readpage* call it will search the linked list for the last entry that contains the *inode* number. Once a matching entry is found it will retrieve the index number for that entry. The index number corresponds to the entry in the *operations* array for the *read* call. The algorithm will set the value of the cached variable for the *read* operation to 0.

Calculating Additional Workload Data Characteristics

In addition to collecting data on individual file system calls the parser collects data on the frequency of repeat write and read calls to the same file. Some workloads such as web server workloads perform read calls on the same set of files. They also perform the write calls on one log file. Having this information and providing it to FileRunner will allow FileRunner to execute a workload similar to the original workload by maintaining the same repeat frequency as the original workload. For original workloads that do not perform repetitive read or write operations on the same file they will have a write and read frequency of 1.

```
void PostProcess(operation * ops) {
    for (int index = 1; index < nops; index++) {
        if (ops[index].op == 34)
            totalwrite++;
        else if (ops[index].op == 35)   totalread++;
        else   continue;
        int i = index - 1;
        operation * op = &ops[index];

        while (i-- >= 0) {
            if (ops[i].inum == op->inum && ops[i].op == op->op)
                if (op->op == 34) writeexisting++;
                if (op->op == 35) readexisting++;

      if (ops[i].inum == op->inum && (ops[i].op == 4))
            break;

      }
}
write_frequency = (totalwrite) / (totalwrite - writeexisting);
read_frequency = (totalread) / (totalread - readexisting);
}
```

Figure 13. Determining the write and read repeat frequencies.

After the parser has parsed all the file system operation information it loops through these operations to determine if any of the write operations have occurred on a file that a previous write operation has happened. The algorithm for this process is presented in figure 13.

The algorithm consists of looping through all of the workload operations and checking if the operation is a write operation. If it is not it goes to the next operation. If it is then it enters another loop traversing the list of previous write operations and it checks to see if the operation is a write operation and if the *inum* number matches the *inum* number of the current operation. If it is then it marks the current write operation as a repeated write operation since it found a write operation on the same file. The algorithm keeps track of the total number of writes in the trace as well as the number of repeated writes. Then it determines the repeat write frequency of the workload by dividing the total number of write calls by the difference of the total number of write calls and the repeated write calls. The algorithm follows the same procedure for determining the read frequency. Both the write and frequency values are stored in global variables that are later outputted into the configuration file for FileRunner.

Populating the Statistical Model

At this point the parsing is complete and the indices for each *operation* in the *operations* array are computed. A 6 dimensional integer array that is referred to as the statistical matrix is used to represent the workload characteristics. The array dimensions are the time chunk dimension, the process dimension, the operation dimension, the I/O dimension, the directory depth dimension, and the cached dimension.

Some of the dimensions have a fixed size while others vary depending on the trace data or chunking size. The number of VFS calls is always fixed and has the size of 52 as there are 52 VFS operations that are shown in appendix B. So the operation dimension has a size of 52. The size of the cache dimension is 2 for the 2 possible values: cached, not cached. The size of the process dimension is the number of unique process IDs found in the trace. So this will vary based on the number of active processes found in the trace. The size of the time chunk dimension and the I/O size dimension are predetermined and are discussed in the workload chunking sections. The size of the directory depth dimension is the maximum directory depth size for a file or directory found in the trace. Using the known sizes for the statistical matrix the 6 dimensional array is created and initialized by setting the value in every cell to zero.

After the matrix is initialized a loop is performed on the *operations* array. At this point the feature functions have been applied to the raw data and converted to indices along the 6 dimensions. For example, if the file path was "/tmp/tmp2/file.txt" then the entry in the *operations array* would contain the value 2 indicating the subdirectory level of the file. If the operation was a *read* operation then based on Appendix B the value of the operation index would be 35. So for every operation found in the trace the value of the cell in the matrix that corresponds to the indices of the operation is incremented by 1.

An example of the process of converting the raw data into indices and incrementing the statistical matrix is shown in figure 14. The figure assumes only three workload characteristics that are being tracked as opposed to the six that this work is tracking. The three characteristics are the operations (*read/write*), I/O size, and directory depth. The *read* operation is given the index of 0 and the *write* operation is given the index of 1. The figure shows the raw trace of one *read* operation on the file "/m/m.c" with an I/O size of 4096. The operation index is 0 since it is a *read* operation, and if the *iochunksize* is set as 2048 then the I/O index would be 4096 / 2048 = 2. The directory depth index is 1 since it is being performed on the first subdirectory. Once the indices are computed the statistical matrix is initialized with all cells having a value of 0. Then for each trace point (only one is shown in this case) the value in the cell that corresponds to the indices (0, 2, 1) is incremented by one.

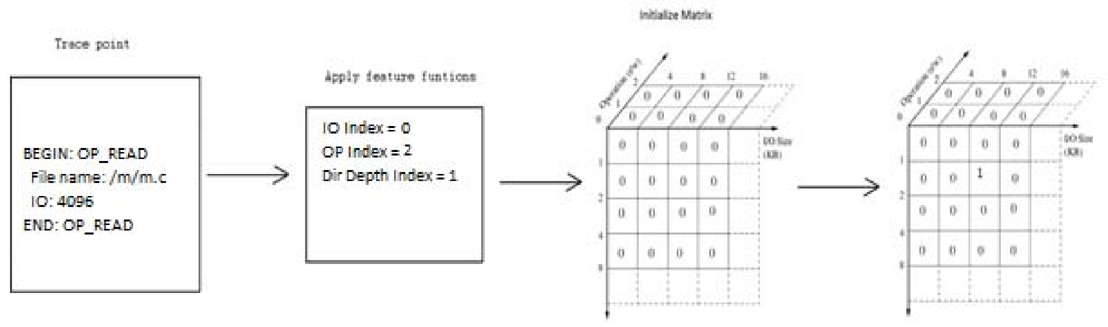

Figure 14. The parsing process.

Generating the Configuration File

Configuration File Structure

After parsing the trace file and generating the statistical matrix the parser generates the configuration file for FileRunner. The statistical matrix includes all the information about all of the file system operations present in the original workload and that information is written to the configuration file. A description of the structure of the configuration file is presented next followed by a section that describes the process of generating the configuration file from the statistical matrix.

The configuration file consists of two sections: header and body. The header spans the first 4 lines of the configuration file and contains global workload values that includes the read and write frequencies, I/O chunk size, and the max directory depth value. These values are calculated during the parsing process and are stored in the parser's global variables that is described in previous sections. A sample header for a configuration file is shown in figure 15 that shows that the I/O chunk size is 512 bytes, a max directory depth of 2, a repeat write frequency of 2.2, and a repeat read frequency of 1.1.

```
IOChunkSize = 512
DirDepthIndex = 1
RepeatWrite = 2.2
RepeatRead = 1.1
```

Figure 15. Sample header for a configuration file.

The body of the configuration file is comprised of one or more time chunks with each time chunk containing the operations that FileRunner will perform in that chunk. Each chunk definition starts with a line indicating the time chunk index. The format of this line is "TIndex = n" where n is a number indicating the time chunk index. This is followed by zero or more lines where each line defines an operation, its characteristics, and the number of iterations to perform that operation. The operation definition line consists of 6 comma separated values that correspond to a dimension in the statistical matrix. The first value indicates the process ID index, the second value corresponds to the operation index, followed by the directory depth value, then the I/O index, then whether the operation is cached or not, and finally by the number of iterations to execute the operation. Figure 16 shows a sample configuration time chunk.

```
TIndex = 0
0,19,0,0,0,10
```

Figure 16. Sample time chunk.

The first line "TIndex = 0" indicates the start of a time chunk definition. The TIndex value is 0 indicating that it is the first time chunk in the configuration file. The time chunk contains one operation definition that consist of 6 comma separated values and is "0,19,0,0,0,10". The first value in that line indicates the process ID index that is 0. The second value is the operation index and it is 19. The value 19 indicates that it is a create file operation based on Appendix B. The third value indicates the directory depth. A value of 0 indicates that it is to be performed on the parent directory. The fourth value is the I/O index and this value will be 0 for all create operations. The cached index is also 0 and is not used in the case of the create operation. The last value represents the number of times to perform the create operation and that value is 10.

Generating the Configuration File

The configuration generation process consists of two parts. The first part generates the header and simply outputs the values of the global variables that are *iochunksize* and the write and read frequency variables. Also the header outputs the size of the directory depth dimension. The second part consists of looping through the elements of the statistical matrix that is a 6 dimensional array, and adding a line for each cell value that is greater than or equal to 1. The line would consist of the loop indexes and the value of the cell. Figure 17 shows the pseudo code of the algorithm.

```
Void GenerateConfigFile () {

          //Generate the header
          Write   ("IOChunkSize = ",       iochunksize);
          Write   ("DirDepthIndex = ",     mdim.dirdepth);
          Write   ("RepeatWrite = ",       writefrequency);
          Write   ("RepeatRead = ",        readfrequency);
          //Generate Body
          Loop through time dimension(tindex)
            Write "TIndex = ", tindex);
            Loop through processes dimension (pindex)
              Loop through I/O dimension (ioindex)
                Loop through operations dimension (opindex)
                  Loop through dir depth dimension (ddindex)
                    Loop through cached dimension(cindex)
                      value = get_matrix_cell_value()
                            if (value != 0)
                  Write (pindex, opindex, ddindex, ioindex, cindex, value)
```

Figure 17. Configuration file generation process.

FileRunner

FileRunner is the load generation tool created by this work that takes as input a configuration file that contains all the data that characterizes an original workload and produces an equivalent synthetic workload based on that input data. A FileRunner run consists of three stages that consists of parsing the configuration file, data structure initialization and file system setup, and finally the run stage that creates the synthetic file system workload. In the next section a description of the parsing stage is presented. This is followed by a section that describes the data structures that are required to manage the files created by FileRunner along with how these data structures are updated. Next, a description of the setup stage that creates the initial state of the file system. Finally, a description of the run stage is presented.

Parsing Stage

The first step in this stage is to parse the first four lines in the configuration file that constitutes the header and saves the I/O chunk size, directory depth, and the read and write repeat frequencies into the following global variables: *iochunksize*, *maxdirdepth*, *repeatread*, and *repeatwrite*. The parser then goes through the list of file operation definitions and saves each line into the *operations* array where each element is of type struct called *operation* shown in figure 18.

```
typedef struct {
    long tindex;
    long pindex;
    long opindex;
    long cindex;
    long ioindex;
    long ddindex;
    long iters;
} opinfo;
```

Figure 18. Operation struct.

As FileRunner is parsing the configuration file it notes the different values for the I/O size index and saves these unique values into an I/O array. For example, the workload described in figure 19 has the following unique I/O values: 0, 100, 300, and 400. An array of integers called the *iosizesarray* containing these values is created and the total number of unique values is also saved in a separate global variable called *uniqueIOcount* that has a value of 4.

```
IOChunkSize = 2048
DirDepthIndex = 3
RepeatWrite = 1
RepeatRead = 1
TIndex = 0
0,19,0,0,0,10
0,34,0,300,0,10
0,35,0,400,0,10
0,22,0,0,10
TIndex = 1
0,19,0,0,0,10
0,34,0,300,0,10
0,35,0,400,0,7
0,22,0,0,10
TIndex = 2
0,19,0,0,0,10
0,34,0,100,0,6
0,22,0,0,10
```

Figure 19. Sample workload to illustrate the IO array.

File Indexers

When a file is created by FileRunner it can be written to or deleted at a later stage in the workload. Furthermore, when FileRunner encounters a delete operation it needs to be able to choose a file that is on disk and delete it, or if a file is deleted from disk it needs to know that it can no longer perform any file system operation such as reading from it or writing to it. Additionally, when a file undergoes a write operation it can be later used to be read from. So FileRunner needs to keep track of the size of the files to be able to issue read calls to files with appropriate size so that it does not issue a read operation with a size larger than the size of the file. FileRunner needs to also track the files' location relative to the main directory so it can issue file system operations on files with the desired directory level. Finally, if the read or write frequencies in the header are greater than one then FileRunner needs to be able to issue repeat reads and writes on files to ensure it generates a workload with the same read and write frequencies.

So throughout the setup stage and during the workload generation stage FileRunner needs to keep track of the files that are created, or available to be written to or deleted, and the files that are available for reading along with the size of each file. To do that FileRunner uses one data structure for storing references to all files called the file set and 3 other data structures called file indexers that contains references to the files in the file set. The first file indexer contains the references to the files that can be written to called write indexer, the second one contains references to the files that can be read called read indexer, and the last one contains references to the files that can be deleted called delete indexer.

The file set is a one dimensional array of the file struct shown in figure 20. The data structure consists of two variables. The first is a number called fd that refers to the index of the file in the file set. The fd value of the first file created by FileRunner is zero and each subsequent file created would have an fd value equal to the fd value of the previously created file plus one. So the second file created would have an fd value of 1 and the third an fd value of 2. This is accomplished using a file counter that counts the number of files already present in the file set. It is incremented every time a new file is created and that value is assigned to the fd variable. The second variable is a character array that stores the full path of the file.

```
typedef struct {
      long fd;
      char fullpath[100];
   } file;
```

Figure 20. File struct.

The write and delete indexers have the same structure and are composed of two parts. The first part is the main array where each element of the array is an array of integers. In other words, it is an array of integer arrays. The size of the main array is the value of the *maxdirdepth* variable found in the header of the configuration file. Based on figure 19 the size of the main array is 3. Initially, each of the three subarrays are of zero size. The second part of the write and delete indexers is called the position array and is an array of integers. The size of the position arrays is the value of *maxdirdepth*. Whenever a file is needed for a write or delete operation the position arrays are used to reference the indexers and retrieve the fd value for a file in the file set. The usage of the position arrays to access the fd values in the write and delete indexers is detailed later in this section.

The read indexer is also composed of two parts. The first part is a two dimensional array of integer arrays. The size of the first dimension is the value of the *maxdirdepth* variable found in the header of the configuration file. Based on figure 19 the size is 3. The size of the second dimension corresponds to the value of *uniqueIOcount* that is determined in the parsing stage. Based on figure 19 the value of *uniqueIOcount* is 4 since there are 4 unique I/O indexes found in the workload that are: 0, 100, 300, and 400. Each cell in this 2 dimensional array contains an array of integers. Initially, those arrays contain no elements. The second part of the read indexers is a 2 dimensional integer array called the position array. The size of the first dimension is the value of the *maxdirdepth* variable. The size of the second dimension corresponds to the value of *uniqueIOcount*. The usage of the position array to access the fd values in the read indexer is detailed later in this section.

```
IOChunkSize = 2048
DirDepthIndex = 3
RepeatWrite = 1
RepeatRead = 1
TIndex = 0
0,19,0,0,0,2
0,19,1,0,0,2
0,19,2,0,0,2
TIndex = 1
0,19,0,0,0,1
TIndex = 2
0,34,0,300,0,1
0,34,0,400,0,1
0,34,1,300,0,1
0,34,0,400,0,1
```

Figure 21. Sample create workload.

Figure 21 contains a sample workload and is used to illustrate how FileRunner utilizes the file indexers to track files. A detailed description of the process of how these file indexers are updated is described in the next section. There are three default rules for updating the indexers.

The first rule states that every time a file is created a new entry is added to the file set and then to the write indexer. The second rule states that every time a file is written it is added to the read indexer. Finally, the third rule states that whenever a file is read it is added to the delete indexer.

The first time chunk in figure 21 issues 2 create operations on the main directory, 2 create operations at subdirectory level 1, and 2 create operations on subdirectory level 2. The second time chunk issues 1 create operation on the main directory. The resulting file set and write indexer data structures are shown in figure 21. Basically, for each create operation a new entry will be added to the file set. There are a total of 7 create operations that are reflected in the file set as shown in the figure. Since the first two create operations having fd values of 0 and 1 are created on the main directory, or directory level of 0, these values are added to the first array, or the array at index 0, of the write indexer. The second 2 create operations are performed on the subdirectory of the main directory, or subdirectory level of 1, these two new files with fd values of 2 and 3 are added to the second array of the write indexer, or the array at index of 1. Next, when the third operation definition is executed two additional files will be added to the write indexer's third array or index of 2. These two files have fd values of 4 and 5. Then, the second time chunk performs one create operation at the main directory so a new file with an fd value of 6 is created and added to the first array of the write indexer as shown in figure 22.

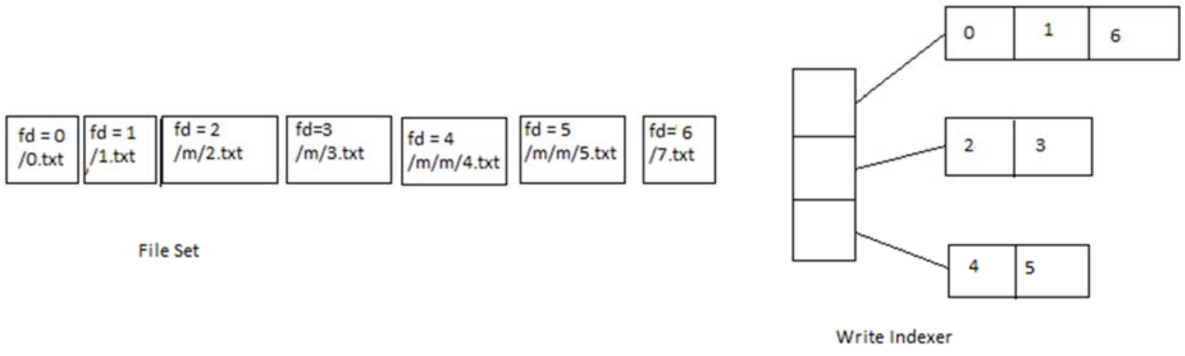

Figure 22. File set and write indexer.

As indicated earlier the parsing stage will create an array that stores the unique values of the I/O values called the *iosizesarray* and *uniqueIOcount* will contain the size of that array. Based on figure 21 the *iosizesarray* contains three values that are 0, 300, and 400 and the value of *uniqueIOcount* will be 3. Also, based on the workload in figure 21 the first dimension of the read indexer will be *maxdirdepth* or 3 and the value of the second dimension will be the value of *uniqueIOcount* or 3.

The third time chunk shows four write operations definitions. The first write definition issues a write on directory level 0 with an I/O size of 300 times the *iochunksize*. Whenever a write operation needs to access a file it checks the write indexer to get an index of the file in the file set. In this case the write needs to be performed on the main directory so it accesses the fd value from the first array of the write indexer. Since this is the first write operation on that directory it will grab the first fd value in the first array. Based on figure 22 that fd value is 0. Once the write operation is completed it will add that fd value to the read indexer. Since the write size has an index of 300 the write operation will determine the index of the value of 300 in the *iosizesarray*. Based on figure 21 that value is 1. So the fd value of 0 is added to the 2 dimensional array of the read indexer at the index [0,1]. This is reflected in figure 23.

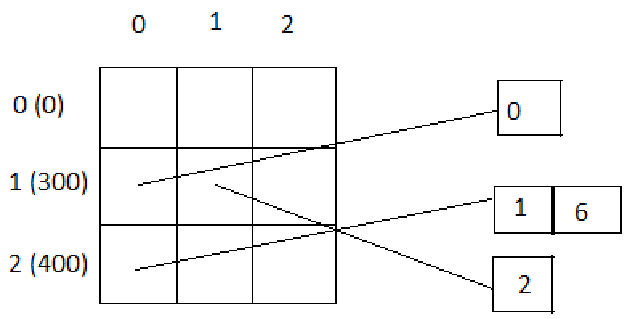

Figure 23. Read indexer.

The second operation definition specifies a write on the main directory and of I/O size 400 times the *iochunksize*. Since this is the second operation on that directory the write operation will access the second fd value in the first array of the write indexer. The second fd value is 1. The index of 400 in the *iosizesarray* is 2. So the fd value of 1 is added to the read indexer at the following index: [0,2].

The third operation definition specifies a write on the subdirectory level 1 and of I/O size 300 times the *iochunksize*. Since this is the first operation on that subdirectory the write operation will access the first file in the second array. This file would have the fd value of 2. The index of 300 in the *iosizesarray* is 1. So the fd value of 2 is added to the read indexer at the following index: [1, 1]. Similarly, the fourth operation will add the fd value of 6 to the read indexer at the array located at index [0, 2]. The read indexer after the write updates is shown in figure 23.

The mechanism of tracking the files that were last accessed in the write indexer involves the second part of the write indexer data structure called the position array. As presented earlier this data structure is an array of integers and has a size of 3 since the value of *maxdirdepth* is 3. Initially, all the three values in this array are zero. So when the first write operation in the third time chunk of figure 21 needed a file that is located in the main directory it checked the value of the position array at index 0. That value was 0 since all values are initially set to 0. So it grabbed the fd value in the first array of the write indexer. That fd value in the write indexer is 0 as shown in figure 22. So the file set was accessed using the index 0 that referenced the physical file with the path “/0.txt”. Then after the write operation the position array at index 0 is increment by 1.

The second operation needed a file on the main directory. So the value of the position array at index 0 is retrieved and that value is 1. So it grabbed the second fd value in the first array of the write indexer. That value is 1 as shown in figure 22. Then it incremented the position array at index 0 by 1. That value is now 2.

The third write operation required a file at the first subdirectory level, or subdirectory level 1. The position array was accessed using the second element, or index 1. That value is 0. The second array in the write indexer array was accessed at index 0. As shown in the figure that fd value is 2. So the file set was accessed at index 2 and the file at that index has the path "/m/2.txt".

When the fourth operation needed a file on the main directory it will check the value of the position array at index 0 and that value is 2. So it grabbed the third fd value in the first array of the write indexer. That fd value is 6 as shown in figure 22.

Updating the File Indexers

The default method of updating the file indexers is as follows. Once a file is created it is added to the write indexer signifying that a file is created and ready to be written to. Once a file is written to it is added to the read indexer signifying that it is ready to be read. Once it is read it is placed into the delete indexer until it is finally deleted. There are however exceptions to these default rules. To illustrate the default method and the exceptions for updating the file indexers a sample workload is presented in figure 24.

```
IOChunkSize = 2048
DirDepthIndex = 3
RepeatWrite = 1
RepeatRead = 1
TIndex = 0
0,19,0,0,0,10
0,34,0,1,0,10
0,35,0,1,0,10
0,22,0,0,10
TIndex = 1
0,19,0,0,0,10
0,34,0,1,0,10
0,35,0,1,0,7
0,22,0,0,10
TIndex = 2
0,19,0,0,0,10
0,34,0,1,0,6
0,22,0,0,10
```

Figure 24. Sample workload to illustrate the file indexers updates.

In the first time chunk 10 files will be created as specified in the first line and these 10 files will be added to the write indexer. The second line specifies 10 write operations that FileRunner will execute on those same 10 files and then adds them to the read indexer. The third line performs the 10 reads and adds these files to the delete indexer that will finally be deleted in the last line of the first time chunk. This illustrates the default option for updating the file indexers.

The second time chunk creates 10 files and adds them to the write indexer. The second line will write 10 files, but only 7 of those files will be placed into the read indexer the remaining 3 will be placed in the delete indexer. This is because the write operation performs a check to see if the number of read operations is less than the number of delete operations. If this is the case then it places the needed number of files in the read indexer and the remaining ones in the delete indexer. In this example the write operations will place 7 files into the read indexer and 3 files into delete indexer. Later in the run stage the read operations will place the 7 files it reads into the delete indexer making the total number of files available for deletion 10.

The create operation performs also performs a check to see if the number of delete operations are greater than the number of write operations. If they are then it places the needed number of operations in the write indexer and the remaining ones in the delete indexer. The third time chunk shows 6 write operations and 10 delete operations. In this case, the create operation will place 6 files into the write indexer and the remaining 4 into the delete indexer. Once the write operation completes there will be 10 available files for the 10 delete operations.

There are also 2 other exceptions that override the default assignment of the file indexers and it is when the read and write frequencies are greater than one. To illustrate how the file indexers are updated when the frequencies are greater than 1 the sample workload in figure 25 is used. In that workload the read and write frequencies are both set to 2. The value 2 for the write frequency indicates the half the write operations need to be performed on a file where a write operation has already been performed. In the figure the second line of time chunk 1 indicates that there are 10 write operations. What would happen here in the case of write frequency of 2 is that 10/2 = 5 of those files are placed in the read indexer, the other 5 would be placed back in the write indexer so that they would be written to again. Similarly, the read operations would place 5 files in the delete indexer, and the other 5 back into the read indexer so they can also be read again.

```
IOChunkSize = 2048
DirDepthIndex = 3
RepeatWrite = 2
RepeatRead = 2
TIndex = 0
0,19,0,0,0,10
0,34,0,1,0,10
0,35,0,1,0,10
0,22,0,0,10
TIndex = 1
0,19,0,0,0,10
0,34,0,1,0,10
0,35,0,1,0,7
0,22,0,0,10
TIndex = 2
0,19,0,0,0,10
0,34,0,1,0,6
0,22,0,0,10
```

Figure 25. Sample workload with repeat frequencies more than 1.

Overview of Creating the Initial State of the File System

The sample input configuration for FileRunner presented in figure 26 shows that in the first time chunk 10 files are created. In the second time chunk 15 files are written to. The write frequency of value 1 indicates that each file is written to exactly once. This means that the file system contained at least 5 files before the run started because the workload so far has only created 10 files but has written to 15 different unique files. So in order to recreate the workload synthetically the setup portion of FileRunner needs to create those 5 files before starting the workload.

```
IOChunkSize = 2048
DirDepthIndex = 3
RepeatWrite = 1
RepeatRead = 1
TIndex = 0
0,19,0,0,0,10
TIndex = 1
0,34,0,1,0,15
TIndex = 2
0,35,0,1,0,15
0,35,0,2,0,15
TIndex = 3
0,22,0,0,15
0,22,1,0,30
```

Figure 26. Sample configuration file.

The third time chunk contains two operation definitions. The first indicates that there are 15 reads with I/O index of 1 and the second indicates that there are 15 reads of I/O index of 2. To be able to perform these read operations the files with the appropriate size need to be available on disk. So there needs to be 15 files with an I/O size of 1*2048 =2048 bytes, and another fifteen files with the size of 2*2048 = 4096 bytes. The second time chunk, have specified to perform 15 writes with an I/O index of 1. This means that there are 15 files of size 2048 bytes available on disk. So the first read operation definition will be able to issue 15 reads using those files. However, the workload so far has not written 15 files with an I/O size of 4096 bytes. This means that these files were already available before the workload has started. So the setup process would have to create those 15 files with the appropriate size before the workload starts.

The fourth time chunk contains 2 operation definitions. The first specifies 15 delete operations are performed on files at directory depth of 0, or the main directory. The second 15 delete operations are performed on a directory depth of 1, or in a subdirectory of the main directory. For FileRunner to be able to issue the delete operations these file have to be available on disk. In the previous time chunk there are operation definitions that created fifteen files in the main directory. So the first delete operation definition can issue the delete operations on those files, but since no files have been created by the workload in the subdirectory the setup portion of the FileRunner needs to create those 15 files in the appropriate location before starting the workload.

So the setup stage determines the files that are needed by the run stage that are not created by the workload itself and then it creates those files. For delete and write operations the file needs to be on disk and in the appropriate directory level. For the read operations the files need to be on disk, in the appropriate directory level, and with a size equal to or greater than the size of the read request.

Process of Creating the Initial State of the File System

The process of determining the files that need to be created before the run stage starts uses 3 data structures. The first tracks the files that need to be created and added to the write indexer and is an array of integers having the size of *maxdirdepth*. The value in each element of the array specifies the number of files to create at the corresponding directory level. Figure 27 shows an example of this array having three elements. The first element indicates that three empty files need to be created at the root directory and a value of four in the second element indicates that four empty files need to be created in a subdirectory of the root directory. The third value 0 signifies that no files need to be created in the subdirectory level 2. This variable is called *to-create-to-write*.

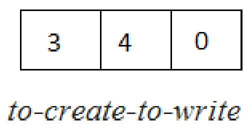

Figure 27. The *to-create-to-write* array.

The second variable tracks the files that need to be created and added to the read indexer and is a two dimensional array. The first dimension is the directory level index and the second dimension is the I/O index. The value in each cell of the two dimensional array specifies the number of files that need to be created at the corresponding directory level and with the corresponding size. This variable is called *to-create-to-read*.

The third variable tracks the files that need to be created and added to the delete indexer and is a one dimensional array. It has the same structure as the *to-create-to-write*. The dimension corresponds to the directory level index and each element contains the number of files that need to be created at that directory level and added to the delete indexer. This variable is called *to-create-to-delete*.

Another set of temporary variables is needed as the algorithm parses the operation definitions and uses these values to update the *to-create* variables. They are 4 variables corresponding to the create, write, read, and delete operations. They contain the number of operations for the current time chunk and are referred to as the *current-ops* variables.

The first variable is called *current-creates* and is a one dimensional array where the dimension is the directory level index and the values in each element of the array specifies the number of create operations at the directory level. The second variable is called *current-writes* and is a two dimensional array where the first dimension is the directory level index and the second dimension is the I/O index. The value in each cell of the two dimensional array specifies the number of write operations. The third variable is called *current-reads* and is a two dimensional array where the first dimension is the directory level index and the second dimension is the I/O index. The value in each cell of the two dimensional array specifies the number of read operations. The last variable is called *current-deletes* and is a one dimensional array where the dimension is the directory level index and the values in each element of the array specifies the number of delete operations at the directory level. A high level description of the algorithm is presented next followed by an example that illustrates the process. Then the implementation details of the algorithm are presented.

The process starts by going through the list of operations in each time chunk and storing the number of operations in the *current-ops* variables. Then the algorithm loops through the directory levels and checks to see if the number of create operations on each directory level is less than the number of delete operations on that directory level. If it is then it needs to create files to satisfy the delete requests. The setup stage performs the same checks for the write operations.

The algorithm then goes through two nested loops. The first loop goes through the directory level indexes and the second loop goes through the I/O indexes and it checks to see if there are read operations on files that do not have a corresponding write operations. In other words it checks to see if the number of write operations at the directory level with a given size is less than the number of read operations at the same directory with the same size. If it is then it needs to create files on that directory level with that size to satisfy the read operations.

This algorithm is illustrated using the sample workload provided in figure 28. The setup algorithm goes through the operations one time chunk at a time starting with the first time chunk. In the example in figure 28 there are 2 operation definitions in the first time chunk. The first creates 10 files and the second deletes 5 files. Both are on directory level 0. So the first portion of the algorithm updates the two current variables: *current-creates* and *current-deletes*. These two current variables are 1 dimensional arrays containing three elements because the directory depth value is 3. Upon going through the operations of the first time chunk the *current-creates* variable's first element gets set to 10 since there are 10 create operations on the directory level index of 0. Similarly, the first element of the *current-deletes* array gets updated to the value of 5.

```
DirectoryDepth = 3
TIndex = 0
0,19,0,0,0,10
0,22,0,0,5
TIndex = 1
0,22,0,0,4
TIndex = 1
0,22,0,0,5
```

Figure 28. Sample workload to illustrate the file setup process.

```
for (j = 0; j < dirdepth; j++) {
       //deletes
        if (currentdeletes[j] > currentcreates[j]) {
              currentdeletes[j] -= currentcreates[j];
              currentcreates[j] = 0;
              to-create-to-delete[j] += currentdeletes[j];

        } else {
             currentcreates[j] -= currentdeletes[j];
              currentdeletes[j] = 0;
        }

}
```

Figure 29. Algorithm segment for the *to-create-to-delete*.

FileRunner then goes into a loop that iterates through the directory level index. A segment of the algorithm showing this loop is presented in figure 29. In the first iteration where the index j is zero the *current-deletes* value is 5 while the *current-creates* has a value 10. The conditional is false so the *current-creates* at index 0 becomes 10 – 5 = 5, and the first element of *current-deletes* becomes 0. The algorithm continues to loop, but no values changes since the remaining values for the remaining arrays are zero.

The algorithm goes into the second time chunk operation definitions and in that time chunk there is only one operation definition that is a delete operation on directory level 0 with 4 iterations. The algorithm updates the first element of the *current-deletes* to 4. Then the algorithm goes to the second stage where the conditional evaluates to false again and the *current-deletes* first elements is set to 0 after the *current-creates* is updated and that becomes 5 – 4 = 1.

The algorithm now goes to the third iteration and updates the value of the *current-deletes* to 5 and it goes to the second stage. The conditional at that point evaluates to true which sets the value of the *current-deletes* at 5 - 1 = 4 and the *current-creates* to 0. Then it updates the *to-create-to-delete* variable at index 0 to 4. Finally, it sets the value of the current deletes at index 0 to 0.

In summary, the workload specifies in the first time chunk that 10 files are created and then 5 of those files are deleted. This leaves 5 files available in the file system. The second time chunk deletes 4 of those files leaving one file. The third time chunk needs to delete 5 files, but there is only 1 file left. So the setup needs to create 4 files in order for the third time chunk be able to perform 5 delete operations.

```
//writes
      docreate[j][k]->numwrite =
            docreate[j][k]->numwrite / cdata->writerepeat - 1;
             numwrites[j][k] += docreate[j][k]->numwrite;
             if (docreate[j][k]->numwrite > numcreates[j]) {
                 docreate[j][k]->numwrite -= numcreates[j];
                 numcreates[j] = 0;
                 docreate[j][k]->tocreateempty +=
                        docreate[j][k]->numwrite - numcreates[j];
            } else {
                   numcreates[j] -= docreate[j][k]->numwrite;
            }
```
Figure 30. Algorithm segment for the *to-create-to-write*.

The algorithm performs the exact same steps to update the *to-create-to-write* data structure. The only difference here is that the number of writes is divided by the write frequency. This is because when the write frequency is greater than 1 the run stage will put back into the write indexer some files to be written to again. For example, if the value of the write frequency is 2 and there are 5 create operations and 10 write operations, then there is no need to add any values to the *to-create-to-write* data structure because during the run stage FileRunner will add 10 / 2 = 5 files to the write indexer and there are already 5 files added due to the 5 create operations. This segment of the algorithm is shown in figure 30.

The algorithm does a similar check to update the *to-create-to-read* data structure. Basically, the algorithm, checks to see if for a given read operation there is one write operation matching the directory level index and the I/O index. If not it would add to the number of files required to the *to-create-to-read* data structure. The number of read operations is also divided by the read frequency. The segment of the algorithm that performs this check is shown in figure 31.

```
 docreate[j][k]->numread = docreate[j][k]->numread / cdata->readrepeat;
if (docreate[j][k]->numread > numwrites[j][k]) {
      docreate[j][k]->numread -= numwrites[j][k];
      numwrites[j][k] = 0;
      docreate[j][k]->tocreatewithcontent += docreate[j][k]->numread;
} else {
      numwrites[j][k] -= docreate[j][k]->numread;
}
```

Figure 31. Algorithm segment for the *to-create-to-read*.

After FileRunner has looped through all of the operation definitions for all time chunks it enters the last stage of the setup process. In this stage the FileRunner loops through all of the *to-create* variables and creates the files on disk and updates the file indexers accordingly. This concludes the setup stage and FileRunner enters the run stage that is described in detail in the next section.

FileRunner Run

After the setup process is completed FileRunner issues a command to tracefs to start the tracing. To maintain workload variation FileRunner executes the operations for each time chunk in sequential order and for each time chunk FileRunner will first execute the create operations, then the write operations, followed by the read operations, and finally running the delete operations. This is done by looping through the *operations* array and executing all the create operations for the first time chunk. Then it resets the loop index and goes through the *operations* array and executes all the write operations for the first time chunk. Then it resets the loop index and goes through the *operations* array and executes all the read operations for the first time chunk, and it does the same thing to execute the delete operations. Once all the delete operations for the first time chunk are executed the time chunk index is incremented. Then the process is repeated to execute the operations for the second time chunk in the same manner. The segment of this algorithm is presented in figure 32. *Tindex* is the time chunk index and *curropindex* is the current operation index.

```
for (i = 0; i < nops; i++) {
        if (tindex != ops[i]->tindex && curropindex == 22) {
            currtindex = i;
            curropindex = 19;
            tindex = ops[i]->tindex;
        } else if (tindex != ops[i]->tindex && curropindex == 19) {
            curropindex = 34;
            i = currtindex;
        } else if (tindex != ops[i]->tindex && curropindex == 34) {
            curropindex = 35;
            i = currtindex;
        } else if (tindex != ops[i]->tindex && curropindex == 35) {
            curropindex = 22;
            i = currtindex;
        }
}
```

Figure 32. FileRunner algorithm for ordering the execution of operations.

Each operation in the operations array has the required information to enable FileRunner to execute the appropriate file system operation. For instance, if the operation is a create operation it will call the create method passing it the directory level index and the number of iterations. The create operation first determines the base path based on the directory level index it is passed and the globally defined base path. If the directory level index value is 2 then the create method concatenates the base directory with two strings “000/”. If the base directory is “root” then the path becomes “/root/000/000/”.

The create operation then goes into a loop and in each iteration it will create the full path. It does this by incrementing the file counter and then concatenating it to the base path and then finally adding the string “.txt” to the end of the path. So if the current file index is 11 the final path would be “/root/000/000/11.txt”. The create method will then create the file on disk, and add the file either to the write indexer or the delete indexer based on the criteria discussed in the previous section.

The write method takes 3 parameters that are the number of iterations, the directory level index, and I/O size as values. First, the write method creates a string of the size that is specified in the I/O size parameter. Then it goes into a loop and in each iteration it picks an index from the write indexer based on the provided directory level index, uses that index to get the file in the file set. Using the path defined in the file it writes to that file that string it created earlier. Finally, it adds the file index into either the read indexer, write indexer, or delete indexer based on the criteria specified in previous sections.

The read operation takes in 3 parameters that are the number of iterations, the directory level, and I/O size as values. It goes into a loop and picks a file index from the read indexer based on the directory level index and the I/O size index. Then it uses the index to get the file in the file set. Using the path defined in the file it performs a read to that file. Lastly, it adds the index to the delete indexer.

Finally, the delete operation takes in two parameters. The first is the directory level and the second is the number of iterations. It enters into a loop and in each iteration it grabs an index of a file from the delete indexer and then uses that index to get the file from the file set. It then uses the path defined in the file to issue a delete operation.

Evaluating the Results

So far the methodology has presented how to trace an original workload and use the collected traces to generate configuration files for FileRunner in order to generate a synthetic workload that is equivalent to the original workload. In this section a description of the process that was used to validate this work is presented starting with a description of how the results are presented. This is followed by a description of how the measurements were captured and how the performance metrics were calculated. Finally, the overall evaluation process is described.

Presenting the Results

This approach was validated using the same metrics that were used by Tarasov et al. (2012). These metrics are throughput, latency, and I/O. In addition, this work validated that the cache contribution for the two workloads are similar. In their work the authors calculated the root means square (RMS) distance and the maximum distance for all the parameters and are presented as shown in figure 33. This work used the same method to validate all observed parameters. The authors also used graphs to show how closely the synthetic workload matched the original workload. One of the graphs in their work is shown in figure 34.

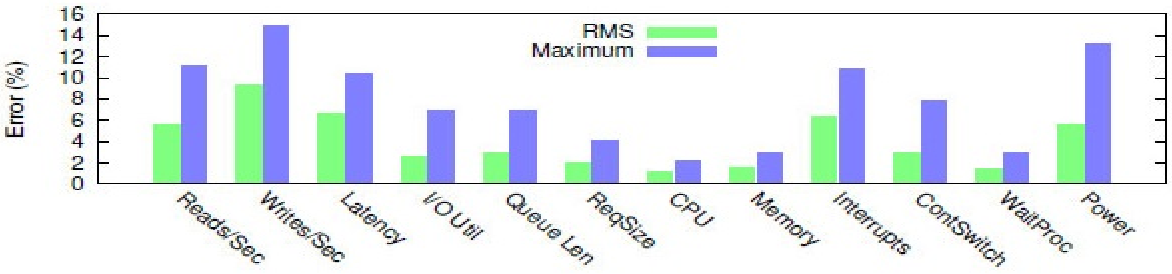

Figure 33. Maximum emulation error and RMS distance for observed parameters.

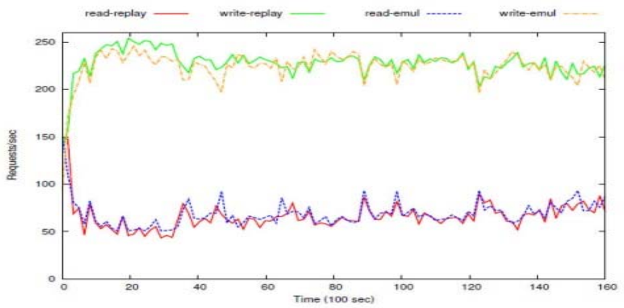

Figure 34. Read and writes per second for the original and synthetic workloads.

Calculating the Workload's Performance Parameters

The throughput and latency data were extracted from the traces since the operations and their timestamps are logged in the traces. Throughput is calculated by dividing the number of operations by the total time. The difference between the first and last timestamp for the operation in the trace determines the duration of the entire workload. The number of operations divided by the duration provides the overall throughput for the workload. The number of operations can be determined by summing all the cells in the statistical matrix. The latency for an individual operation can be calculated by subtracting the timestamp of the original call from return value. The two workloads are also validated in how they both make use of the cache. The cache usage information is already available in the statistical matrix. Specifically, this is found in the cached dimension of the statistical matrix.

Overall Evaluation Process

The work was validated on a single custom built machine. Specifically the machine had an Intel Core i5-4590 CPU, 8GB of ram, and a 1 TB hard drive. The operating system used was Linux with kernel version 3.2. The traces were run and collected on the same machine. The synthetic workloads were also run and traced on the same machine. The validation process starts by describing how the original workload is run and traced followed by how the synthetic workload is run and traced.

The remainder of the validation process is split into two parts. The first part consisted of three experiments that used the C application created by this work. Each experiment validated a specific aspect of the methodology. The first experiment validated modeling workload variation. The second experiment verified that the work can produce a synthetic workload with an I/O footprint that matches the I/O footprint of the original workload. The third experiment validated that the work can produce a synthetic workload with the same cache behavior as the one produced by the original workload. The three experiments only validated one aspect of the work the remaining aspects where not validated.

The second stage consisted of running larger workloads that tested the methodology as a whole in addition to testing the scalability of the work. This included extracting performance metrics of the original workload such as operation latencies and throughput and comparing it to the performance metrics of the synthetic workload.

As presented previously the work was validated by a variety of workloads to ensure that this work is capable of producing equivalent workloads for a diverse set of workloads. The list of workloads that this work used to validate the work is discussed in a previous section and includes one workload produced by Postmark, three workloads produced by FileBench, and a workload produced by a custom application.

Postmark was run using the default values and generated workloads similar to email servers. The three FileBench workloads that were used were sample workloads included in the FileBench installation and they include a workload that emulates a web proxy server, a workload that emulates a file server, and a workload that emulates a web server. Finally, a custom application was implemented that created a workload that varied over time and was used to verify that the methodology can model workload variation. The other workloads were not validated against workload variation since they do not exhibit any variation as they are run.

The evaluation process for validating the synthetic workload for one original workload is presented in figure 35. The initial steps consist of running the original workload, capturing the traces, and then generating the configuration files are discussed earlier as part of the methodology. The next step was to run the synthetic workload and trace it.

The synthetic workload was traced for two reasons. The first is that the trace work consumes some system resources slightly affecting the original workload. So in order to run the synthetic workload in a similar environment the trace software need to be running as well. The second reason is that the traces contain information about the operations and their timestamps and were used to calculate the throughout and latency that are needed in the validation process. In addition, the trace contained information on cache usage that was also used in validating the work.

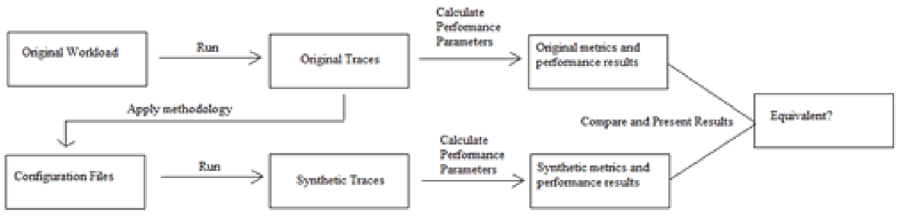
Original Workload
Run
Original Traces
Calculate Performance Parameters
Original metrics and performance results
Apply methodology
Compare and Present Results
Equivalent?
Configuration Files
Run
Synthetic Traces
Calculate Performance Parameters
Synthetic metrics and performance results

Figure 35. The evaluation process.

The next step in the evaluation process consisted of calculating and presenting the performance parameters for the two workloads as discussed in the previous subsections. This is followed by comparing the two results and determining whether the two workloads are considered equivalent. This work considered two workloads to be equivalent if the difference between the observed metrics is less than 10% with a maximum allowable difference of 15%. This is the same approach used in (Tarasov et al., 2012). This work also validated workload variation. This was done by plotting the file access data for the original and synthetic workloads as shown in figure 34.

Resources Used

The work was limited to a single custom built machine. Specifically the machine had an Intel Core i5-4590 CPU, 8GB of ram, and a 1 TB hard drive. The traces were run and collected on the same machine. The synthetic workloads were run and traced on the same machine. Between every run the system was restored to the same state as the original trace was run. This eliminated any file system aging effect on the performance. In addition, the only operating system considered is the Linux OS. The work can be easily extended to other operating systems such as Windows using file system call listeners.

Summary

This chapter presented the methodology of this work that outlined the process of generating equivalent synthetic workloads from original workloads. The original workloads used in this work consisted of workloads generated by FileBench, Postmark, and a custom application. The original workloads were traced by a tool called tracefs and the log file generated by tracefs was passed to a parser that extracted the workload characteristics into the statistical model. This work configured tracefs to log the address space operations that provided information about cached operations. This information was then used when generating the configuration files that in turn configured FileRunner to issue file system calls that were satisfied from cache or disk.

The statistical model consisted of sub-matrices where each sub-matrix represented the workload characteristic for a specific time interval or time chunk. Then for each time chunk the configuration files for FileRunner were created. FileRunner then ran the configuration files to produce the equivalent synthetic workload. The synthetic workload generated by FileRunner was traced and validated against the traces from the live workload.

Chapter 4

Results

Overview

The previous chapter presented the methodology for generating equivalent synthetic workloads. This chapter presents the validation approach and the data obtained from running various workloads aimed at verifying the methodology described in chapter 3. The validation process was divided into two stages. The first stage consisted of running various workloads aimed at validating specific aspects of the methodology that include modeling workload variation, matching I/O chunk size, and cache behavior. This basically validated that each component of the methodology is functioning and produced the desired results. The second stage consisted of running larger workloads that tested the methodology as a whole in addition to testing the scalability of the work. This included extracting performance metrics of the original workload such as latency and throughput and comparing it to the performance metrics of the generated synthetic workload.

The next section describes the process for running an original workload and creating its synthetic equivalent. This process was used throughout the validation process to validate various aspects of the methodology. This section used a C application that consisted of one outer loop that contained 4 inner loops. The first inner loop would create files, the second inner loop would write to those files, the third inner loop would read those files, and finally the fourth inner loop would delete those files.

The code of the C application is presented in appendix C. There are two parameters that control the number of iterations of the main outer loop and the number of operations to perform in each inner loop and they are *numberOfChunks* and *numberOfOps* respectively. The *numberOfOps* controls the number of operations performed in each chunk. The *numberOfChunks* controls the total number of loops.

The following section uses the process of running an original workload and creating its synthetic equivalent that is introduced in the previous section to present the validation process pertaining to modeling workload variation. It illustrates how the workload of the C application varied. Then it presents the validation process that included tracing the original workload and dividing the trace into chunks to produce a synthetic workload that varies the same way as the original workload.

In the next section the validation process for verifying that the synthetic workload produced the same I/O footprint as the original workload is presented. It first details the changes made to the C program that produced a workload with a varied I/O size footprint. Then it describes the process of determining the I/O chunk size and how it was verified.

The following section presents the findings in regards to modeling cache behavior by first describing the changes made to the C program and then validating that tracefs and the custom parser can identify whether the operations are satisfied from cache or from disk. Then it describes the process of generating the synthetic workload with the same cache behavior as the original workload.

In the experiments just mentioned only one specific aspect of the methodology was validated. So the experiment that validated the cache behavior was not validated against any other aspect. The following section discusses the results of the synthetic workload's performance relative to the original workload based on all of the metrics mentioned above. In addition, larger workloads were used to validate the scalability of the work. The workloads used include the custom C application, Postmark, and three personalities from FileBench. The C application is the only application that produced a varied original workload. The remaining 4 exhibit a uniform workload over time. So only the C application was validated against workload chunking.

Running a workload and Obtaining Results

Tracefs comes with two scripts. The first script performs two actions that includes loading the tracefs module into the kernel and starting the tracing. The second script stops the trace, unloads the tracefs module from the kernel, and then runs the tool to parse the binary trace and converts it into textual format. The second script was modified with an action being added that runs the custom parser that was created by this work. The custom parser parses the log created by tracefs and generates the configuration file for FileRunner. This configuration file was used by FileRunner to create a workload that is referred to as the synthetic workload. The custom parser also performs other tasks that include calculating throughput, latency, and counting the number of operations performed by the workload including identifying whether the read operations were performed from cache or disk.

The first step in creating the synthetic workload was to setup a workload that was traced with tracefs. This is referred to as the original workload. In this case the original workload was created by a custom C application presented in Appendix C. The *numberOfChunks* parameter of the C application was set to 10 and the *numberOfOps* parameter was set at 5,000. This means that for each chunk there were 5,000 create operations, followed by 5,000 write operations, then 5000 read operations, and finally 5,000 delete operations. This occurred 10 (*numberOfChunks)* times. This made the total number of each type of operation at 5,000 multiplied by 10 or 50,000.

Now that the original workload was setup the next step was to run the first tracefs script that loaded the module into the kernel and started the tracing. Then the C application was run and once it was completed the second tracefs script was run. This stopped the tracing and removed the module from the kernel, then ran the tracefs parser, and finally the custom parser was run. At this stage the configuration file was generated and the data of the original workload was also recorded. The data included the workload's latency, throughput, the total number of operations including cached operations, and the total I/O footprint of the original workload.

Next, the first tracefs script was run again to start capturing traces. FileRunner was run using the configuration file generated earlier. The workload generated by FileRunner is referred to as the synthetic workload. Once FileRunner completed its run the second tracefs script that stops the tracing was run. The workload characteristics for the synthetic workload was recorded and compared to the original workload. Figure 36 shows the number of operations for both runs and that the synthetic workload generated the same operation count as the original workload.

	Create	Write	Read	Delete
Original Workload	50000	50000	50000	50000
Synthetic Workload	50000	50000	50000	50000

Figure 36. Comparing workload's operation footprint.

Results of Modeling Workload Variation

Current synthetic benchmarks do not model workload variation. Workload chunking as described in the methodology was used to enable the modeling of workload variation. The evaluation process section presented that a proper chunk size is required to accurately model it. It then described a custom C application that was used to validate that the generated synthetic workload varies in the same way as the original workload.

The first step in the verification process was to run the C program described in appendix C and obtain a trace of that run. This was followed by the chunk size determination process that determines the proper time chunk size and is presented in the next section. The section after that describes the process of generating the synthetic workload from the configuration file that is created using the determined chunk size.

The parameters of the C application were set as 10 for the *numberOfChunks* and 5,000 for the number of *numberOfOps*. So the workload pattern of the C application consisted of 5,000 create calls, followed by 5,000 write calls, followed by 10,000 read calls, and finally followed by 5,000 delete calls. This was repeated 10 times since the value of *numberOfChunks* was set as 10. A theoretical graph of one chunk of the workload is presented in figure 37. Although the time is not shown on the graph, since the graph is theoretical, the graph shows that the create and delete operations take less time to complete than the read and write operations. This is to be expected as the read and write operation usually have a higher I/O footprint than the create and delete operations. The complete workload would have ten of these chunks in succession.

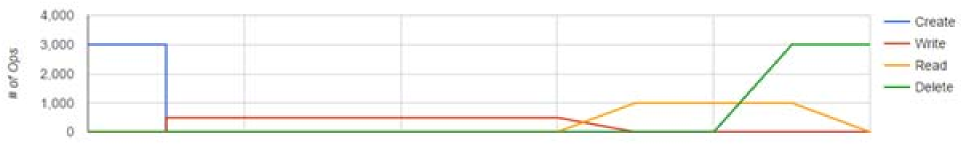

Figure 37. Expected flow of the C workload.

Data for the Determining Proper Number of Chunks

The chunking process divides the workload into smaller chunks. To determine the proper chunk size the custom parser is run with *workloadChunks* parameter varied from smaller to larger until the access pattern of the C program was properly captured. The first time the custom parser was run the value of the *workloadChunks* parameter was set as 10. As can be seen in figure 38 the data at this chunk size does not correctly capture the C application access pattern. The custom parser was run repeatedly with *workloadChunks* set as 10, 20, 50, 100, 200, and 500 and the results are presented in figures 38 through 43. The data in the figures illustrate that as the chunk size was reduced the access pattern of the C application became more apparent**.** Figures 38 and 39 show a lot of overlap between operations calls for each time chunks. This does not correctly capture the access pattern of the custom C application described earlier.

Figures 40 and 41 correspond to the workload divided into 50 and 100 chunks respectively. The workload's access pattern in these two figures starts to show. However, the figures still show a significant overlap between the different types of operations. Figures 42 and 43 capture the access pattern of the original workload therefore illustrating that using the chunking procedure of increasing the number of chunks will capture the workload variation.

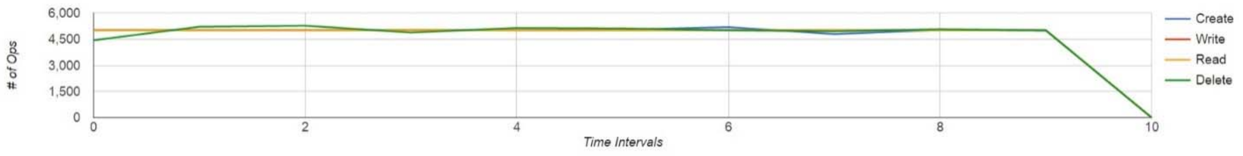

Figure 38. Original workload divided into 10 chunks.

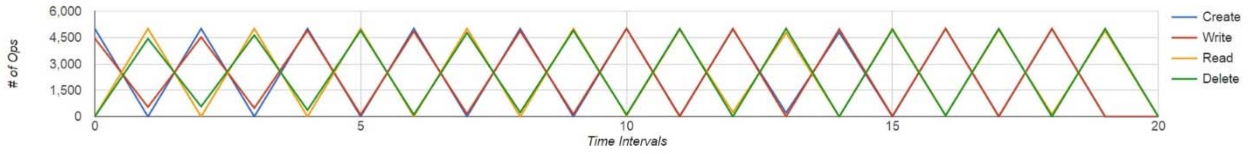

Figure 39. Original workload divided into 20 chunks.

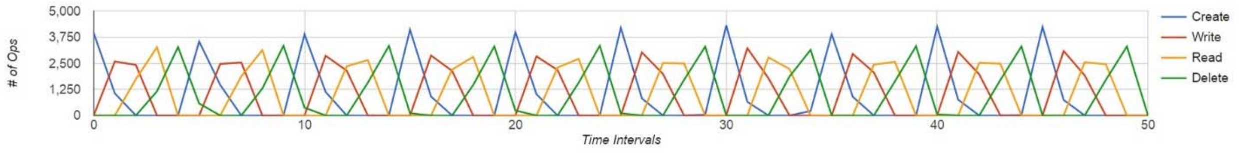

Figure 40. Original workload divided into 50 chunks.

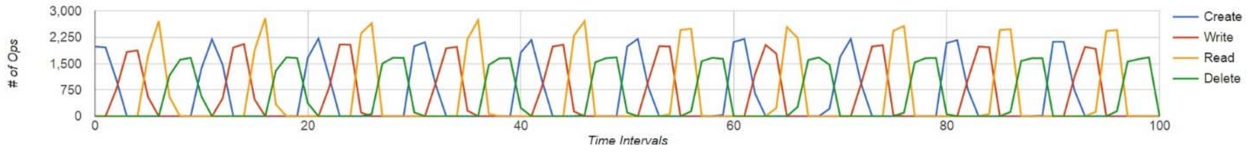

Figure 41 Original workload divided into 100 chunks.

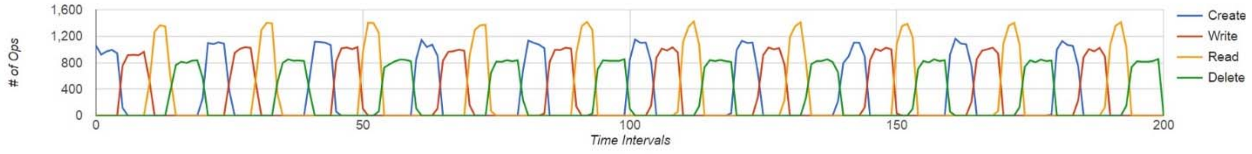

Figure 42. Original workload divided into 200 chunks.

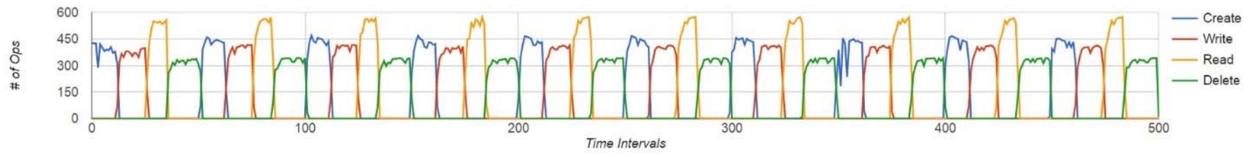

Figure 43. Original workload divided into 500 chunks.

Data for the FileRunner Run

The values of the *numberOfChunks* variable that were used in figures 42 and 45 were 200 and 500. Since both values for the number of *numberOfChunks* captured the access pattern for the original workload either value can be chosen to generate the configuration file for FileRunner. In the case for the value of 200 the workload will have fewer time chunks so it would be easier to modify and extend.

FileRunner was then was run and traced by tracefs. The parser then parsed the trace files generated by FileRunner and the workload's footprint is shown in figure 44. From figure 44 it can be seen that the footprint of the FileRunner workload very closely matches the footprint of the original workload shown in figure 42. Similar data was obtained using various versions of the C program including varying the inner loop values and the workload parameter. Furthermore, FileRunner generated the same number of calls as the original workload while maintaining the original workload's variation.

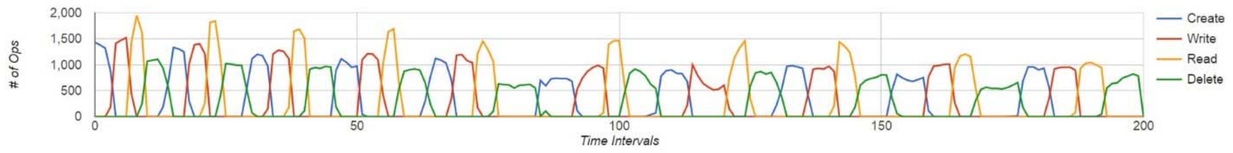

Figure 44. FileRunner run at 200 chunks.

Correctly Generating Proper I/O Size

To properly validate that the work produced a synthetic workload with a similar I/O size footprint as the original workload this work needed to be validated against a workload that produced operations with varying I/O size. This illustrates that the work can handle a range of varied I/O size operations and that the synthetic workload created by this work produced an equivalent I/O footprint as the original workload.

The C program was modified to produce an original workload with varying I/O size for the read and write operations. The change in the program that can be seen in appendix D consisted of creating several literal strings of the following sizes in bytes: 200,000, 100,000, 50,000, 10,000, and 1000.

The inner loop of the read and write operations were changed to select in a circular fashion one of the literal strings to read and write to disk. So in the first iteration the string of size 200,000 was selected. Then in the second iteration the string of size 100,000 was selected. This is followed by selecting the strings of size 50,000, 10,000, and 1,000. The next iteration looped back and selected the string of size 200,000.

Finally, two variables were added and were used to count the total amount of data written and read by the C program. The parser was changed to include the same two variables that counted the total number of bytes read and the total number of bytes written that are present in the trace file. The original workload was run and traced and the values of the total read and write I/O variables in the C program matched the values of the total read and write I/O variables extracted from the trace. It can be concluded that custom parser can correctly extract from the trace the I/O size footprint of the original workload.

Next, several I/O chunk sizes were used and FileRunner was run for each chunk size. The size of the I/O chunks in bytes were: 100, 500, 1,000, 5,000, and 10,000. Table 1 shows the total size of I/O operations for the original workload and the totals for the synthetic workloads at the various chunk sizes. For this specific workload the chunk size of 100 produced a 0.06%

difference in the total I/O size, and size chunk of 10,000 bytes produced a difference of 6.2% in the I/O size for the overall workload.

Table 1. Total I/O Size Based on Chunk Size.

	I/O Chunk size(bytes)	*Read I/O (bytes)*	*Write I/O (bytes)*
Original Workload		3610000000	3610000000
Synthetic Workload	*100*	3612450000	3610651255
Synthetic Workload	*500*	3620670005	3622450000
Synthetic Workload	*1000*	3633190006	3634950000
Synthetic Workload	*5000*	3723300015	3724950000
Synthetic Workload	10000	3848400010	3849950000

Correctly Capturing and Modeling Cache Behavior

At this point in the verification process the synthetic workload matches both the access pattern of the original workload and the I/O size of the file operations. To verify the workload mimics the original workload in cache behavior the first step would be to correctly capture the behavior to model it. The C program was modified slightly and served as a controlled experiment that was used to verify that the parser correctly identifies the operations that are served from the cache and the ones that are served from disk. The change in the program, shown in appendix D, consisted of adding an additional read operation that performed a read of a file that has just been read. This meant that the second read on the same file would cause the read to be satisfied form cache and not from disk. The frequency of the second read was controlled by the conditional statement using the modulus function that controlled the number of the second reads. So if the value of the integer in the conditional was 5 then an additional 20 percent of the reads were made and all of them were satisfied from the cache. Using the same parameter values for the C program (*numberOfChunks*, 10) and (*numberOfOps*, 3000) the number of cached calls should be 20 percent of 30,000 or 6000.The parser correctly identified the number of cached calls. The integer was changed to various values and the parser was also able to determine the correct number of cached calls.

Now that the correct number of cached calls were identified by the parser the second thing that was verified was that FileRunner can produce the same cache behavior. The number of cached calls in the synthetic workload produced by FileRunner was 5999 that is almost identical to the number of cached calls in the original workload. The difference was off by one since the first call had to be from a file that is from disk. The subsequent calls were made were after the file was cached.

Validating Overall Performance

This section presents the results of running longer workloads and validating the synthetic workloads' overall performance with the corresponding original workloads. Performance metrics that were used were latency, throughout, I/O size, and number of cached operations. There were five workloads that were used and they included Postmark, three workloads from FileBench, and a custom C program. The details of the workloads and the results are presented next.

Validating the Work with the C Application

The C application version that was used in this test is the one found in Appendix E. It was the original version without the additional modifications that were used to validate the cached reads or the I/O footprint. The *numberOfChunks* parameter of the C application was set to 10 and the *numberOfOps* parameter was set at 60,000. The size of each read and write I/O operation was set at 4,000,000 bytes. The number of operations produced by the C application along with the one produced synthetically by FileRunner are shown in figure 49. The parser was run three times with the *numberOfChunks* set at 10, 50, and 200. Figures 45 through 47 show the plot of operations per unit chunk for the original workload. The configuration file where the *numberOfChunks* was set to 200 was used to run FileRunner. Figure 48 shows the result of running FileRunner. Figure 49 shows the operation count of the two workloads and that the original workload and the synthetic workload had identical operation count for all operation types.

The I/O footprint of the two workloads was almost identical with less than 1% difference (figure 50). The throughput of the synthetic workload was 4.78% less than that of the original workload. The latencies showed that the difference was within 15% except for the create latency that was at 35%. The data is presented in figure 51.

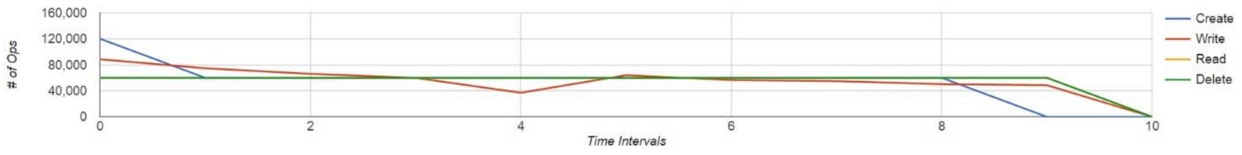

Figure 45. C Application with *numberOfChunks* at 10.

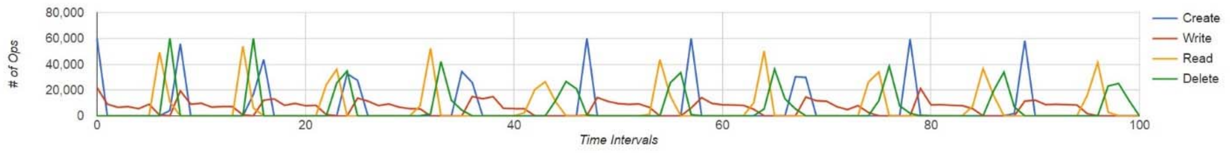

Figure 46. C Application with *numberOfChunks* at 100.

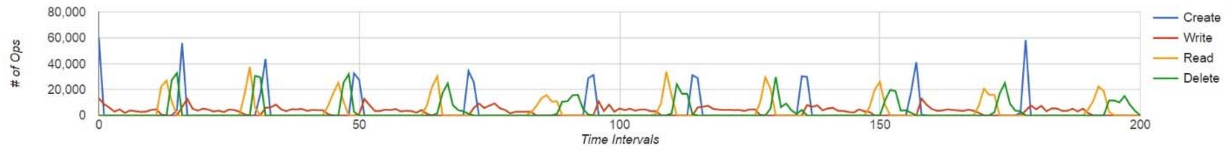

Figure 47. C Application with *numberOfChunks* at 200.

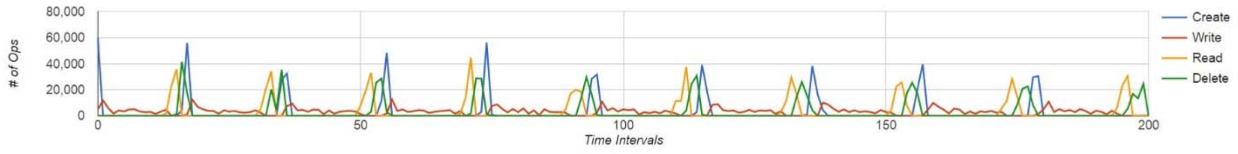

Figure 48. FileRunner with *numberOfChunks* at 200.

	Create	Write	Read	Delete	Read Cached
Original Workload	600000	600000	600000	600000	0
Synthetic Workload	600000	600000	600000	600000	0
Difference	0	0	0	0	0

Figure 49. C application and FileRunner operation counts.

	I/O Read	I/O Write
Original Workload	43320000000	43320000000
Synthetic Workload	43277450085	43349400000
% Difference	0.098222334	-0.06786704

Figure 50. C application and FileRunner I/O footprints.

	Throughput	Avg Create	Avg Write	Avg Read	Avg Delete
Original Workload	278	19	271.17	47.61	10.4
Synthetic Workload	265	12.3	279.16	40.81	9.45
% Difference	4.676258993	35.26315789	-2.946491131	14.28271372	9.134615385

Figure 51. C application and FileRunner throughput and average latencies.

Validating the Work with Postmark

The number of transactions that was specified for the Postmark run was 3,500,000. This produced 1,748,488 create and delete operations, 5,111,815 write operations, and 3,890,374 read operations with 1,943,957 of those satisfied from the cache. The number of operations produced by FileRunner is shown in figure 52.

The total runtime for Postmark was approximately 285 seconds. The throughput of the FileRunner run was 272 seconds or a difference of 4.67% compared to the Postmark run. The throughput and the average latencies of both runs are shown in figure 53 along with the percentage difference between the two. Finally, the I/O footprints of both runs are shown in figure 54. The synthetic workload produced 1.25% more I/O while producing 1.02% less in write I/O.

	Create	Write	Read	Delete	Read Cached
Original Workload	1748488	5111815	3890374	1748488	1943957
Synthetic Workload	1748488	4868386	3745490	1748488	1943956
Difference	0	4.762085482	3.724166	0	1

Figure 52. Postmark and FileRunner operation counts.

	Throughput	Avg Create	Avg Write	Avg Read	Avg Delete
Original Workload	285	13.35	9.65	6.58	10.89
Synthetic Workload	272	18.19	11.76	6.59	11.41
% Difference	4.561403509	-36.25468165	-21.86528497	-0.15197568	-4.775022957

Figure 53. Postmark and FileRunner throughput and average latencies.

	I/O Read	I/O Write
Original Workload	11949996752	11946641543
Synthetic Workload	11798023490	12068678152
% Difference	1.271743124	-1.02151394

Figure 54. Postmark and FileRunner I/O footprints.

Validating the work with Web Proxy Personality of FileBench

The web proxy workload is one of the workload personalities that comes with the FileBench installation. Appendix F shows the complete workload definition that defines 5 threads to execute the following operations in sequence: delete file, create new file, write to that new file, and then read it five consecutive times. This work ran and traced the workload for 300 seconds. Then it generated the synthetic workload using FileRunner. Figure 55 shows the number of operations performed in each workload as well as the number of the read operations that were serviced from cache. It can be seen that the number of operations in each workload were identical to each other. The number of cached operations was only off by 1.

The workloads I/O footprint in bytes is shown in figure 56. The difference in the I/O read was about 2.5% and was less than 0.005% for the I/O write footprint that is within the 10 percent error range. Finally, the workload throughput and average latencies for each operation are shown in figure 57.

	Create	Write	Read	Delete	Read Cached
Original Workload	1168547	1168477	11685103	1168547	10516551
Synthetic Workload	1168547	1168477	11685103	1168547	10516550
Difference	0	0	0	0	1

Figure 55. Web proxy and FileRunner operation counts.

	I/O Read	I/O Write
Original Workload	47835380033	9567685252
Synthetic Workload	46640999325	9567231651
% Difference	2.496856317	0.00474097

Figure 56. Web proxy and FileRunner I/O footprints.

	Throughput	Avg Create	Avg Write	Avg Read	Avg Delete
Original Workload	300	12	19	6	9
Synthetic Workload	286	17	20	5	10
% Difference	4.666666667	41.66667	5.26316	16.66667	11.11111

Figure 57. Web proxy and FileRunner throughput and average latencies.

Validating the work with Web Server Personality of FileBench

The web server workload is one of the workload personalities that comes with the FileBench installation. Appendix G shows the complete workload definition that starts by creating 1000 files with content and one empty file that is referred to as the log file. Then 5 threads are created to execute the workload. Each thread picks a file from the initial 1000 files and then performs a read on that file 10 times in a row. Finally, the thread would append a variable sized text into the log file. The text mean size was 16kb.

This work ran and traced the web server workload for 300 seconds. Then it generated the synthetic workload using FileRunner. Figure 58 shows the number of operations performed in each workload as well as the number of the read operations that were serviced from cache. It can be seen that the number of operations in each workload were identical to each other. The number of cached operations was only off by 1.

	Create	Write	Read	Delete	Read Cached
Original Workload	0	606213	12124996	0	12123995
Synthetic Workload	0	606213	12124994	0	12123994
Difference		0	2		1

Figure 58. Web server and FileRunner operation counts.

The workloads I/O footprint in bytes is shown in figure 59. The difference in the I/O read was about 2.5% and was less than 0.005% for the I/O write footprint which is within the 10 percent error range. Finally, the workload throughput and average latencies for each operation are shown in figure 60.

	I/O Read	I/O Write
Original Workload	93828623638	4966680615
Synthetic Workload	92283520427	4966345979
% Difference	1.646729059	0.006737619

Figure 59. Web server and FileRunner I/O footprints.

	Throughput	Avg Create	Avg Write	Avg Read	Avg Delete
Original Workload	300	0	229.53	109.89	0
Synthetic Workload	338	0	242.27	120.18	0
% Difference	-12.6666667	NA	-5.550472705	0.914378432	NA

Figure 60. Web server and FileRunner throughput and average latencies.

Validating the work with File Server Personality of FileBench

The file server workload is one of the workload personalities that comes with the FileBench installation. Appendix H shows the complete workload definition that starts by creating 1000 files and allocating content for 80 percent of those files. Then 5 threads are created to execute the workload. Each thread first creates a new file and adds it to the file set. Then it writes to that file. Next, each thread picks another file from the file set and appends more content to it. Then a read is performed on that file. Finally, the thread would delete that file.

This work ran and traced the file server workload for 300 seconds. Then it generated the synthetic workload using FileRunner. Figure 61 shows the number of operations performed in each workload as well as the number of the read operations that were serviced from cache. It can be seen that the number of operations in the synthetic workload produced by FileRunner matches the number of operations in the original workload. The number of cached operations is only off by 1.

	Create	Write	Read	Delete	Read Cached
Original Workload	591518	1182992	1183034	591517	1179051
Synthetic Workload	591518	1182992	1183034	591517	1179050
Difference	0	0	0	0	1

Figure 61. File server and FileRunner operation counts.

The workloads I/O footprint in bytes is shown in figure 62. The difference in the I/O read was about 0.5% and about 1.5% for the I/O write footprint that is within the 10 percent error range. Finally, the workload throughput and average latencies for each operation are shown in figure 63.

	I/O Read	I/O Write
Original Workload	78474384557	82379409473
Synthetic Workload	78051549289	83588975345
% Difference	0.538819476	-1.46828665

Figure 62. File server and FileRunner I/O footprints.

	Throughput	Avg Create	Avg Write	Avg Read	Avg Delete
Original Workload	299	19.84	271.17	21.31	21.44
Synthetic Workload	304	20.4	279.16	17.63	17.8
% Difference	-1.6722408	-2.822580645	-2.946491131	17.26888785	16.97761194

Figure 63. File server and FileRunner throughput and average latencies.

Summary of Results

This chapter presented two sets of experiments that were run to validate this work. The first set validated individual components of the methodology that included modeling workload variation, modeling cache behavior, and mimicking I/O footprint. It used a custom C application to do so. The results of each experiment verified the proper functioning of each individual component.

The second stage consisted of running larger workloads, each up to 5 minutes, and compared the performance characteristics of the original workload with that of the workload generated by the methodology. The results indicated that the methodology can create a workload with a throughput within 10% difference and with operation latencies, with the exception of the create latencies, to be within 10% difference and in some cases within the 15% maximum allowable difference. The work was able to accurately model the I/O footprint in some cases the difference was negligible and in the worst case it was at 2.49% difference.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusion

This work advanced file system benchmarking by providing more accurate synthetic workloads that will enable the fine tuning of file systems more accurately. Specifically, the achieved goals of the work can be re-stated as follows:

- Model workload variation if it exists in the original workload
- Incorporate cache behavior to achieve higher accuracy
- Verify the synthetic workload is similar to the original workload by comparing the following metrics
    - Workload throughput
    - Operation Latencies
    - I/O size footprint

Chapter 3 described the approach of this work that consisted of running and tracing an original workload, and then generate an equivalent workload based on the trace using a tool created by this work called FileRunner. The validation approach presented in chapter 4 consisted of two stages. The first stage validated specific aspects of the workload that included modeling variation, cache behavior, and modeling I/O footprint. The second stage validated all aspects of the workload using 5 different original workloads that spanned minutes.

The first aspect that was validated in the first stage is workload variation. A custom C application was used for this validation process. Based on the results presented in figures 38 through 47 and in another experiment where the results are presented in figures 45 through 48 it can be seen that the work is capable of modeling workload variation given that the original workload was segmented into a sufficient number of chunks. The validation process presented that in order to determine the proper number of chunks to segment the original workload is to start with a small number of chunks and then increase that number until the access pattern of the application becomes apparent. Figure 38 shows the workload being split into 10 chunks and when the number of chunks was changed to 20 the graph in figure 39 showed a different access pattern. In figure 40 the workload was divided into 50 chunks and that also shows a different access pattern than the previous two figures. At 100 chunks the workload access pattern starts to show more clearly, but with a significant overlap between the operations. At 200 chunks the overlap between the operations decreased significantly. At 500 chunks, shown in figure 42, the workload did not differ significantly from the one divided into 200 chunks as shown in figure 42.

Once the proper number of chunks was determined the work was then able to emulate the same access pattern using FileRunner in an accurate manner as shown in figure 42. This process was repeated using the same C application but with a workload that spanned 5 minutes as shown in figure 47. Again this work was able to regenerate the same access pattern as shown in figure 49.

Another aspect that was validated in the first stage was the modeling of cache behavior. The synthetic workload generated almost the same number of cached calls as the original workload for all runs present in chapter 4. The difference as can be seen in figures 53, and 56 was only 1 operation. The same result was achieved in the second stage using the 5 workloads. So the methodology can be assumed to produce the correct number of read calls that are serviced from the cache.

The last aspect that was verified in the first stage was the I/O footprint produced by the synthetic workload. It was shown the methodology was correct in assuming that it can start with a large I/O size for the read and write operations and if the total footprint of the synthetic workload differed significantly compared to the original workload then the I/O size can be made smaller and the synthetic workload rerun until the I/O footprint difference was acceptable. The I/O chunking technique was applied to 5 minute workloads in stage 2 and showed that it can consistently generate workloads with an I/O footprint less than 5% as shown in figures 50, 54, 54, 59, and 60.

The second stage validated all aspects of the work and that the work is scalable beyond the smaller workloads that were used in the first stage. Five workloads were used in this stage and each ran for 5 minutes. The five workloads were Postmark, a custom C application, and 3 workload personalities that come with FileBench. The workload characteristics of the original workloads were compared with the workload characteristics of the synthetic workloads and based on the results presented in chapter 4 they are closely matched. The throughput, I/O footprint, and the create, read, write, and delete latencies are all within 10% difference with some characteristics under the 15% maximum allowable difference.

For three workloads, Postmark, the custom C application, and the web proxy workload from FileBench, the create latency was higher than the 15% allowable maximum difference and that is because the setup stage of FileRunner needs to create some files that are used in the workload. Having a larger file count in a directory slows down the file system operations on that directory.

With the exception of the create latency in 3 of the five workloads having a difference of more than 15% it can be seen that this work can accurately generate a synthetic workload based on an original workload. The synthetic workload will have the same variation as the original workload with the same cache and I/O footprints.

Implications

The process of optimizing file systems relies heavily on being able to optimize the file system against its expected usage. Thus, file system benchmarking plays an essential role in file system optimization by providing workloads that are as close as possible to the expected live workloads. As the problem statement presented along with qualifying research in the literature review section current benchmarks do not model cache behavior or workload variation. The methodology in this work has provided a tool called FileRunner that accomplishes these two tasks based on a trace. However, the tool can be easily configured to create a workload even if a trace is not present. In other words, this work presented a file benchmark that can rely on input from a trace or explicitly defined manually by the user. Furthermore, the work can be extended to generate the configuration files for other existing benchmarks such as FileBench without the need to use FileRunner.

Additionally, another novel feature produced by this work is the ability for FileRunner to produce different workloads by multiple processes at different points in time. For example, some servers might have an ongoing workload at all times while some other backup process performs certain tasks at specific times of day. FileRunner can generate a similar workload using one configuration file without the need to run two instances of FileRunner at different times to produce the same workload.

In short, this work advanced the current state of the art in file system benchmarking by being able to generate more accurate workloads, allow the modeling of workload variation, and incorporate cache behavior in workload design.

Recommendations

As discussed earlier this work can be used to create the configuration files for other benchmarks such as FileBench. Furthermore, this work can be extended to include tracing the CPU usage of the original workload and replicating it in the synthetic workload (Tarasov et al., 2012). In addition, this can be tested for scale and if needed be extended to work for longer periods of time (Traeger, Zadok, Joukov, & Wright, 2008). Additionally, this work was not tested against a variety of file systems. The current work only used the ext3 file system. Additional work can be done for testing it on different file systems other than ext3 file system (Shegal, Tarasov, & Zadok, 2010).

Summary

Operating system benchmark suites consist of tools that aide in file system performance measurement and analysis (Agrawal, Dusseau, & Dusseau, 2008, Traeger, Zadok, Joukov, & Wright, 2008). Integral to benchmark suites are workloads that exercise the target systems in repeatable scenarios that are indicative of common application execution. One workload generation methodology involves the study of operating system access patterns by a set of applications that exhibit functions common to production software. The patterns are then incorporated into a tool that reproduces a typical application's function into representative synthetic workloads in the context of benchmark suites (Agrawal, Dusseau, & Dusseau, 2008). Thus the generated workload reproduces primary and secondary storage access with the goal of providing equivalent execution sequences through application programming interface (API) or systems calls indicative of an application's interaction with the operating system (Roselli, Lorch, & Anderson, 2000).

Current synthetic benchmarks do not generate representative workloads due to the assumption that mimicking an application’s API calls is sufficient to reproduce the application’s workload (Traeger, Zadok, Joukov, & Wright, 2008). This oversimplified assumption fails to account for the different execution paths that consist of lower level operating system function calls (Agrawal, Dusseau, & Dusseau, 2008, Joukov, Wong, & Zadok, 2005). An API call can have more than one execution path with a significant difference in performance. As a result two workloads with the same API calls but different execution paths will have a different performance footprint that is dependent on the function calls. An example of an API call that has multiple execution paths is the read API call. The read API call can have an execution path that consists of function calls that reads the file from primary storage, or an execution path that consists of function calls that reads the file from disk whose performance depends on whether the file is fragmented or not.

Another factor that synthetic benchmarks ignore is the variation within the workload. Some workloads are not uniform and exhibit different workload characteristics as the workload progresses. For example, the workload’s as read/write ratio, average file size per request, or frequency of requests change over time. Synthetic benchmarks do not model this variation in their workload generation process.

This work proposed a methodology that advanced synthetic benchmarking research by addressing the two issues mentioned above that consisted of improving the accuracy of synthetic workloads by taking into account the execution paths of the API calls and by modeling workload variation if it exists in a workload.

The overall methodology is shown in figure 64. The first step consists of running a workload and then tracing it using a tool called tracefs. While FileRunner was used to recreate the workloads from the given traces, other benchmarks were used to generate original workloads. Postmark, a custom C application, and three workload personalities from FileBench were used as the original workloads. Once the trace of an original workload is collected it is then divided into smaller intervals that are small enough that the workload in that interval can be considered uniform (Tarasov et al., 2012). This chunking technique allows the modeling of variation as FileRunner would create an equivalent synthetic workload for each chunk.

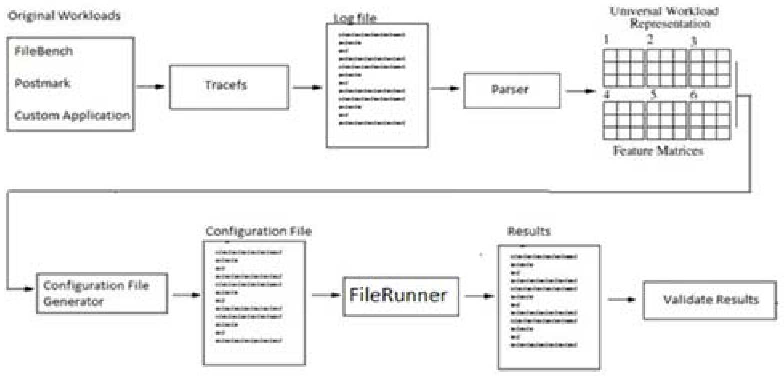

Figure 64. Overall system design.

A FileRunner run consists of three stages that consists of parsing the configuration file, data structure initialization and file system setup, and finally creating the file system workload. The parsing stage first parses the header and saves the I/O chunk size, directory depth, and the read and write repeat frequencies into the appropriate data structures. Then it goes through the list of file operation definitions and saves each line into a separate array element where each element is of type struct called operation.

In the second stage FileRunner performs the setup operations that include initializing the data structures that are needed for the run itself and for creating the initial state of the file system. This creates the files that are needed in the workload. For example, if the workload consists of deleting 5 files then the setup stage needs to create these 5 files.

Once the setup is completed FileRunner enters the third and final stage and that starts the run. At this point there is an array of operations available from when FileRunner parsed the file and possibly some files in the file set that were created in the setup stage. FileRunner then goes through each time chunk and it first executes all the create operations, if any, for that time chunk. Then it executes all the write operations, if any, for that time chunk, followed by the read operations, if any, in that time chunk, and finally the delete operations, if any, for that time chunk. Then it advances to the next time chunk and follows the same order of operation execution until there are no operations to execute.

The workload created by FileRunner is also traced and the workload characteristics are compared with the workload characteristics of the original workloads. This is done for 5 workloads discussed earlier. The results indicated that the methodology can create a workload with a throughput within 10% difference and with operation latencies, with the exception of the create latencies, to be within the allowable 10% difference and in some cases within the 15% maximum allowable difference. The work was able to accurately model the I/O footprint in some cases the difference was negligible and in the worst case it was at 2.49% difference.

In conclusion, this work advanced the current state of the art in file system benchmarking by being able to generate more accurate workloads, allow the modeling of workload variation, and incorporate cache behavior in workload design.

Appendix A

Operations data structure

```
typedef struct  {

    unsigned long long timestamp;
    int op;
    char * ops;
    char * fn;

    unsigned long long fptr;
    unsigned long long fflags;
    char * fname;

    unsigned long long flags;

    unsigned long long inum;

    int ret;
    pid_t pid;

    unsigned long long iptr;
    unsigned long long dptr;

    int offset;
    int cached;
} operation;
```

Appendix B

Operations index

```
#define OP_BASE         0
/* super block ops */
#define OP_READ_INODE               (OP_BASE + 1)                           // 1
#define OP_WRITE_INODE              (OP_READ_INODE + 1)         // 2
#define OP_PUT_INODE                            (OP_WRITE_INODE + 1)        // 3
#define OP_DELETE_INODE             (OP_PUT_INODE + 1)          // 4
#define OP_PUT_SUPER                (OP_DELETE_INODE + 1)       // 5
#define OP_STATFS                   (OP_PUT_SUPER + 1)          // 6
#define OP_REMOUNT_FS               (OP_STATFS + 1)             // 7
#define OP_CLEAR_INODE              (OP_REMOUNT_FS + 1)         // 8
#define OP_UMOUNT_BEGIN             (OP_CLEAR_INODE + 1)        // 9
#define OP_WRITE_SUPER              (OP_UMOUNT_BEGIN + 1)                   // 10
/* dentry ops */
#define OP_D_REVALIDATE             (OP_WRITE_SUPER + 1)        // 11
#define OP_D_HASH                   (OP_D_REVALIDATE + 1)       // 12
#define OP_D_COMPARE                (OP_D_HASH + 1)             // 13
#define OP_D_DELETE                             (OP_D_COMPARE + 1)          // 14
#define OP_D_RELEASE                            (OP_D_DELETE + 1)           // 15
#define OP_D_IPUT                   (OP_D_RELEASE + 1)          // 16
#define OP_DGET                     (OP_D_IPUT + 1)             // 17
#define OP_DPUT                     (OP_DGET + 1)                           // 18
/* inode ops */
#define OP_CREATE                   (OP_DPUT + 1)                           // 19
#define OP_LOOKUP                   (OP_CREATE + 1)             // 20
#define OP_LINK                     (OP_LOOKUP + 1)             // 21
#define OP_UNLINK                   (OP_LINK + 1)                           // 22
#define OP_SYMLINK                  (OP_UNLINK + 1)             // 23
#define OP_MKDIR                    (OP_SYMLINK + 1)            // 24
#define OP_RMDIR                    (OP_MKDIR + 1)              // 25
#define OP_MKNOD                                (OP_RMDIR + 1)              // 26
#define OP_RENAME                               (OP_MKNOD + 1)              // 27
#define OP_READLINK                             (OP_RENAME + 1)             // 28
#define OP_FOLLOW_LINK              (OP_READLINK + 1)           // 29
#define OP_INODE_REVALIDATE         (OP_FOLLOW_LINK + 1)        // 30
#define OP_SETATTR                  (OP_INODE_REVALIDATE + 1)               // 31
#define OP_PERMISSION               (OP_SETATTR + 1)            // 32
/* file ops */
#define OP_LLSEEK                   (OP_PERMISSION + 1)         // 33
#define OP_READ                     (OP_LLSEEK + 1)             // 34
#define OP_WRITE                    (OP_READ + 1)                           // 35
#define OP_READDIR                  (OP_WRITE + 1)              // 36
#define OP_POLL                     (OP_READDIR + 1)            // 37
#define OP_FLUSH                    (OP_POLL + 1)                           // 38
#define OP_MMAP                     (OP_FLUSH + 1)              // 39
#define OP_LOCK                     (OP_MMAP + 1)               // 40
#define OP_FSYNC                    (OP_LOCK + 1)                           // 41
#define OP_FASYNC                   (OP_FSYNC + 1)              // 42
#define OP_IOCTL                    (OP_FASYNC + 1)             // 43
#define OP_OPEN                     (OP_IOCTL + 1)                          // 44
#define OP_RELEASE                  (OP_OPEN + 1)                           // 45
/* mmap ops */
#define OP_READPAGE                             (OP_RELEASE + 1)            // 46
#define OP_READPAGES                (OP_READPAGE + 1)           // 47
#define OP_WRITEPAGE                (OP_READPAGES + 1)          // 48
```

```
#define OP_WRITEPAGES            (OP_WRITEPAGE + 1)          // 49
#define OP_PREPARE_WRITE         (OP_WRITEPAGES + 1)         // 50
#define OP_COMMIT_WRITE          (OP_PREPARE_WRITE + 1)      // 51
```

Appendix C

C Program to Model Variation

```
int main(int argc, char** argv) {

        int numberOfOps = 10000;
        int numberOfChunks = 10;
        int i = 0, j = 0;

        for (i = 0; i < chunk; i++) {

    //creates
    for (j = 0; j < loop; j++) {
                sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
                int fd = open64(filepath, createflag, 0666);
                close(fd);
      }
    }

    //writes
    for (j = 0; j < loop; j++) {
      sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
      int fd = open64(filepath, openflag, 0666);
      write(fd, content, contentsize);
      close(fd);
    }

        //reads
    for (j = 0; j < loop; j++) {
                sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
                fopen(filepath, "ab+")))
                fread(buff, contentsize, 1, (FILE*) fp);
                fclose(fp);
        }

    //delete
    for (j = 0; j < loop; j++) {
      sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
      remove(filepath);
    }
}
```

Appendix D

C Program with Varied I/O size

```
int main(int argc, char** argv) {

        int numberOfOps = 10000;
        int numberOfChunks = 10;
        int i = 0, j = 0;

        for (i = 0; i < chunk; i++) {

    //creates
    for (j = 0; j < loop; j++) {
                sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
                int fd = open64(filepath, createflag, 0666);
                close(fd);
      }
    }

    //writes
    for (j = 0; j < loop; j++) {
      sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
      int fd = open64(filepath, openflag, 0666);
         if (j % 5 == 0)
                contentsize = 200000;
          else if (j % 5 == 1)
                contentsize = 100000;
          else if (j % 5 == 2)
                contentsize = 50000;
          else if (j % 5 == 3)
                contentsize = 10000;
          else if (j % 5 == 4)
                contentsize = 1000;
      write(fd, content, contentsize);
      close(fd);
    }

         //reads
    for (j = 0; j < loop; j++) {
         sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
         fopen(filepath, "ab+")))
         if (j % 5 == 0)
                contentsize = 200000;
         else if (j % 5 == 1)
                contentsize = 100000;
         else if (j % 5 == 2)
                contentsize = 50000;
         else if (j % 5 == 3)
                contentsize = 10000;
         else if (j % 5 == 4)
                contentsize = 1000;
         fread(buff, contentsize, 1, (FILE*) fp);
         fclose(fp);
         }

    //delete
    for (j = 0; j < loop; j++) {
```

```
                sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
                remove(filepath);
            }
        }


void initialize_content(long n) {
    int i = 0, j = 0, size = 0;

    for (i = 0; i < n; i++) {


        if (i == 0)
            size = 200000;
        else if (i == 1)
            size = 100000;
        else if (i == 2)
            size = 50000;
        else if (i == 3)
            size = 10000;
        else if (i == 4)
            size = 1000;

        newcontent[i] = malloc(size);

        memset(newcontent[i], 0, size);
        for (j = 0; j < size - 1; j++) {
            newcontent[i][j] = 'a';
        }
        newcontent[i][j] = 'a';

    }
}
```

Appendix E

C Program with Cache Modeling Logic

```
int main(int argc, char** argv) {

        int numberOfOps = 10000;
        int numberOfChunks = 10;
        int i = 0, j = 0;

        for (i = 0; i < chunk; i++) {

    //creates
    for (j = 0; j < loop; j++) {
                sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
                int fd = open64(filepath, createflag, 0666);
                close(fd);
      }
    }

    //writes
    for (j = 0; j < loop; j++) {
      sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
      int fd = open64(filepath, openflag, 0666);
      write(fd, content, contentsize);
      close(fd);
    }

        //reads
    for (j = 0; j < loop; j++) {
                sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
                fopen(filepath, "ab+")))
                fread(buff, contentsize, 1, (FILE*) fp);
                fclose(fp);

         if (j % 5 == 0) {
                lseek64(filepath, 0, SEEK_SET);
                fd = open64(filepath, openflag, 0666);
                read(fd, content, contentsize);
                close(fd);
              }
           }
        }

    //delete
    for (j = 0; j < loop; j++) {
      sprintf(filepath, "%s/%d-%d.txt", dir, i, j);
      remove(filepath);
    }
}
```

Appendix F

Web Proxy workload by FileBench

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

set $dir=/tmp
set $nfiles=10000
set $meandirwidth=1000000
set $meanfilesize=16k
set $nthreads=5
set $meaniosize=16k
set $iosize=1m

define fileset
name=bigfileset,path=$dir,size=$meanfilesize,entries=$nfiles,dirwidth=$meandirwidth,prealloc=8
0

define process name=proxycache,instances=1
{
  thread name=proxycache,memsize=10m,instances=$nthreads
  {
    flowop deletefile name=deletefile1,filesetname=bigfileset
    flowop createfile name=createfile1,filesetname=bigfileset,fd=1
    flowop appendfilerand name=appendfilerand1,iosize=$meaniosize,fd=1
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile2,fd=1,iosize=$iosize
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize
    flowop closefile name=closefile4,fd=1
    flowop openfile name=openfile5,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile5,fd=1,iosize=$iosize
    flowop closefile name=closefile5,fd=1
    flowop openfile name=openfile6,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile6,fd=1,iosize=$iosize
    flowop closefile name=closefile6,fd=1
    flowop opslimit name=limit
  }
}

echo  "Web proxy-server Version 3.0 personality successfully loaded"
usage "Usage: set \$dir=<dir>"
usage "       set \$meanfilesize=<size>    defaults to $meanfilesize"
usage "       set \$nfiles=<value>     defaults to $nfiles"
usage "       set \$nthreads=<value>   defaults to $nthreads"
usage "       set \$meaniosize=<value> defaults to $meaniosize"
usage "       set \$iosize=<size>  defaults to $iosize"
usage "       set \$meandirwidth=<size> defaults to $meandirwidth"
usage "       run runtime (e.g. run 60)"
```

Appendix G

Web Server workload by FileBench

```
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#


set $dir=/tmp
set $nfiles=1000
set $meandirwidth=20
set $filesize=cvar(type=cvar-gamma,parameters=mean:16384;gamma:1.5)
set $nthreads=5
set $iosize=1m
set $meanappendsize=16k


define fileset
name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,dirwidth=$meandirwidth,prealloc=100,readonly
define fileset
```

```
name=logfiles,path=$dir,size=$filesize,entries=1,dirwidth=$meandirwidth,prealloc
define process name=filereader,instances=1
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop openfile name=openfile1,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile1,fd=1,iosize=$iosize
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile2,fd=1,iosize=$iosize
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize
    flowop closefile name=closefile4,fd=1
    flowop openfile name=openfile5,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile5,fd=1,iosize=$iosize
    flowop closefile name=closefile5,fd=1
    flowop openfile name=openfile6,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile6,fd=1,iosize=$iosize
    flowop closefile name=closefile6,fd=1
    flowop openfile name=openfile7,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile7,fd=1,iosize=$iosize
    flowop closefile name=closefile7,fd=1
    flowop openfile name=openfile8,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile8,fd=1,iosize=$iosize
    flowop closefile name=closefile8,fd=1
    flowop openfile name=openfile9,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile9,fd=1,iosize=$iosize
    flowop closefile name=closefile9,fd=1
    flowop openfile name=openfile10,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile10,fd=1,iosize=$iosize
    flowop closefile name=closefile10,fd=1
    flowop appendfilerand
name=appendlog,filesetname=logfiles,iosize=$meanappendsize,fd=2
  }
}
echo  "Web-server Version 3.1 personality successfully loaded"
run 60
```

Appendix H

File Server workload by FileBench

```
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#


set $dir=/tmp
set $nfiles=10000
set $meandirwidth=20
set $meanfilesize=128k
set $nthreads=5
set $meaniosize=1m
set $meanappendsize=16k
set $runtime=5


define fileset
name=bigfileset,path=$dir,size=$meanfilesize,entries=$nfiles,dirwidth=$meandirwidth,pre
alloc=80
```

```
define process name=filereader,instances=1
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop createfile name=createfile1,filesetname=bigfileset,fd=1
    flowop writewholefile name=wrtfile1,srcfd=1,fd=1,iosize=$meaniosize
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile1,filesetname=bigfileset,fd=1
    flowop appendfilerand name=appendfilerand1,iosize=$meanappendsize,fd=1
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile1,fd=1,iosize=$meaniosize
    flowop closefile name=closefile3,fd=1
    flowop deletefile name=deletefile1,filesetname=bigfileset
    flowop statfile name=statfile1,filesetname=bigfileset
  }
}


set $nfiles=1000


echo "  File-server Version 3.0 personality"
echo "    \$dir=$dir"
echo "    \$nfiles=$nfiles"
echo "    \$meandirwidth=$meandirwidth"
echo "    \$meanfilesize=$meanfilesize"
echo "    \$nthreads=$nthreads"
echo "    \$meaniosize=$meaniosize"
echo "    \$meanappendsize=$meanappendsize"
echo "    \$runtime=$runtime"


run 300
```

References

Agrawal, N., Dusseau, A.C., & Dusseau, R.H. (2009). Generating Realistic Impressions for File-System Benchmarking. *Proccedings of the 7th conference on File and storage technologies,* 125-138.

Agrawal, N., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2008). Towards realistic file-system benchmarks with CodeMRI, *ACM SIGMETRICS Performance Evaluation Review, 36(2),* 52-57.

Ahmad, I. (2007). Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server. *International Symposium on Workload Characterization*

Anderson, D. (2009). Fstress: A Flexible Network File Service Benchmark. *Proceedings of the 7th conference on File and storage technologies.*

Anderson, E., Kallahalla, M., Uysal, M., & Swaminathan, R. (2004). Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies.*

Aranya, A., Wright, C.P., & Zadok, E. (2004).Tracefs: A File System to Trace Them All. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 129-145

Chen, P. M. & Patterson, D. A. (1994). A new approach to I/O performance evaluation: self-scaling I/O benchmarks, predicted I/O performance, *ACM Transactions on Computer Systems (TOCS), 12*(4), 308-339.

Congalton, R. G., & Green K. (2008). *Assessing the Accuracy of Remotely Sensed Data: Principles and Practices (pp. 110)*. Boca Raton, FL: CRC Press, Taylor and Francis Group

Eeckhout, L., Sundareswara, R., Yiz, J., Lilja, D.J. & Schrater, P. (2005). Accurate statistical approaches for generating representative workload compositions. *Proceedings of the 2005 IEEE International Workload Characterization Symposium,* 56-66.

FiST: Stackable File System Language and Templates. Retrieved June 2, 2014 from http://www.filesystems.org/

Ganger, G., Worthington, B., & Patt, Y. (2008). The DiskSim Simulation Environment Version 4.0 Reference Manual. Retrieve from http://www.pdl.cmu.edu/DiskSim/index.shtml.

Ghemawat, S., Gobioff, H., & Leung, S.T. (2003). The Google file system. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 29-43.

Gomez, M. E. & Santonja, V. (2000). A new approach in the modeling and generation of synthetic disk workload. *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems,* 199-206.

Griffin, J.L., Schindler, J., Schlosser, S.W., Bucy, J.C., & Ganger, G.R. (2002). Timing-accurate Storage Emulation. *Proceedings of the 1st USENIX Conference on File and Storage Technologies.*

Hui, L. (2009). Realistic workload modeling and its performance impacts in large-scale eScience grids. *IEEE Transactions on Parallel and Distributed Systems, 21*(4), 480-493.

Iamnitchi, A., Ripeanu, M., Neto, E.S., & Foster, I.(2011). The Small World of File Sharing. *IEEE Transactions on Parallel and Distributed Systems, 22*(7), 1120-1134.

Jacobs, A. (2009). The pathologies of big data. *Communications of the ACM, 52*(8), 36-44.

Joshi, A., Eeckhout, E., Bell, R.H., & John, L.K. (2008). Distilling the Essence of Proprietary Workloads into Miniature Benchmarks. *ACM Transactions on Architecture and Code Optimization,* 5(2).

Joukov, N., Wong, T., & Zadok, E. (2005). Accurate and efficient replaying of file system traces. *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, 37–350*

Kaiser, J, Meister, D., Hartung, T. & Brinkmann A. (2012). ESB: Ext2 Split Block Device. *Proceedings of the 18th International Conference on Parallel and Distributed Systems*

Marakatos, E.P., Pnevmatikatos, D.N., Flouris, M.D., & Katevenis, M.G.H. (2002), Web-conscious storage management for web proxies, *IEEE/ACM Transactions on Networking, (10)*6, 735-748.

Park, A., Becker, J.C., & Lipton, R.J. (1990). IOStone: A Synthetic File System Benchmark. *ACM SIGARCH Computer Architecture News 18*(2), 45-52.

Roselli, D., Lorch, J.R. & Anderson, T.E.(2000). A Comparison of File System Workloads. *Proceedings of the annual conference on USENIX Annual Technical.*

Konishi, R., Amagai, Y., & Sato, k. (2006). The Linux Implementation of a Log-structured File System. *ACM SIGOPS Operating Systems Review, 40(3),* 102-107

Schneider, S., Mueller, U., & Tiegelbekkers, D. (2005). A Reactive workload generation framework for simulation – based performance engineering of system interconnects. *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems,* 484-490.

Shan, H., Antypas, L., & Shalf, J. (2008). Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark. *Proceedings of the*

2008 ACM/IEEE conference on Supercomputing, 1-12.

Shegal, P., Tarasov, V. & Zadok, E. (2010). Optimizing Energy and Performance for Server-Class File System Workloads. *ACM Transactions on Storage (TOS)*, 6(3).

Tarasov, V., Bhanage, S., Zadok, E., & Seltzer, M. (2011). Benchmarking File System Benchmarking: It *IS* Rocket Science. *13th USENIX Workshop in Hot Topics in Operating Systems (HotOS XIII).*

Tarasov, V., Kumar, S., Hildebrand, D., Povnzner, A., Kuenning, G., & Zadok, E. (2012). Extracting Flexible, Replayable Models from Large Block Traces. FAST 12.

Traeger, A., Zadok, E., Joukov, N., & Wright, C.P.(2008). A Nine Year Study of File System and Storage Benchmarking. *ACM transactions on storage,* 4(2).

Williams, J. N. (1977). The construction and use of a general purpose synthetic program for an interactive benchmark on demand paged systems, *Proceedings of the 1977 annual conference,* 459-465.

Zhang, J., Sivasubramaniam, A., Franke, H., & Gautam, N. (2004). Synthesizing Representative I/O Workloads for TPC-H. *Proceedings of the 10th International Symposium on High Performance Computer Architecture,* 142-152.

Ziyad Al Balty, Z.A., (2011). Linux 2 6 37. Retrieved December 02, 2012, from http://kernelnewbies.org/Linux_2_6_37

Zhu, N., Chen, J., & Chiueh, T. C. (2005). TBBT: scalable and accurate trace replay for file server evaluation. *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies,* 323–336.