FILE SYSTEMS

Filebench
A Flexible Framework for File System Benchmarking

VASILY TARASOV, EREZ ZADOK, AND SPENCER SHEPLER

Vasily Tarasov is a Researcher at IBM Almaden Research Center. He started to use and contribute to Filebench extensively while working on his PhD at Stony Brook University. Vasily's interests include system performance analysis, design and implementation of distributed systems, and efficient I/O stacks for ultra-fast storage devices. vtarasov@us.ibm.com

Erez Zadok received a PhD in computer science from Columbia University in 2001. He directs the File systems and Storage Lab (FSL) at the Computer Science Department at Stony Brook University, where he joined as faculty in 2001. His current research interests include file systems and storage, operating systems, energy efficiency, performance and benchmarking, security, and networking. He received the SUNY Chancellor's Award for Excellence in Teaching, the US National Science Foundation (NSF) CAREER Award, two NetApp Faculty awards, and two IBM Faculty awards. ezk@cs.stonybrook.edu

Spencer Shepler is a Private Cloud Architect at Microsoft. Prior to Microsoft, Spencer worked at a failed startup and before that at Sun Microsystems. While at Sun, Spencer worked on Filebench along with many others and at his full-time job of bringing NFSv4 to market. sshepler@microsoft.com

File system benchmarks constitute a crucial part of a storage evaluator's toolbox. Due to the wide variety of modern workloads and ever-growing list of storage stack features, modern benchmarks have become fairly complex tools. This article describes Filebench, one of the most popular modern file system benchmark tool suites. Using several practical examples, we demonstrate Filebench's versatility, expressiveness, and ease of use. It is our hope that this article will encourage people to use Filebench to describe their real-life workloads as well as publicly contribute new workloads.

Filebench is a highly flexible framework for file system and storage benchmarking. The project started around 2002 inside Sun Microsystems and was open-sourced around 2005. It is now hosted at sourceforge.net [2] and maintained by the community, centered around the File systems and Storage Lab (FSL) at Stony Brook University. According to Google Scholar, Filebench was used in over 500 publications and remains one of the most popular file system benchmarks both in academia and industry. The popularity of the Filebench framework comes mainly from the fact that it is shipped with several predefined macro workloads, e.g., Web-server, Mail-server, and File-server. This allows users to easily benchmark their file systems against several sufficiently different workloads with a single tool.

The intrinsic power of Filebench originates, however, not from the included workloads but rather from its expressive Workload Model Language (WML), which allows users to encode a wide variety of workloads. We therefore find ourselves disappointed that most users do not go beyond the predefined workloads and consequently do not utilize the full power of Filebench. The goal of this article is to educate the community on Filebench's WML and demonstrate both its long-standing and recently added features. In addition, we describe best practices for using Filebench to avoid common beginners' mistakes.

Basic Functionality

Many existing storage benchmarks (e.g., fio, mdtest, and SPECsfs) hard code the workloads they generate quite rigidly. A user can specify some basic workload parameters (e.g., I/O size, number of threads, read/write ratio) but cannot really control the execution flow in detail. Expressing a workload with a general-purpose programming language (e.g., C/C++ or Python) is another extreme that offers the utmost flexibility but is time-consuming. The Filebench framework provides a much needed middle ground: high flexibility combined with the ease of describing a workload.

In Filebench, users define workloads using a Workload Model Language (WML). There are four main entities in WML: *fileset*, *process*, *thread*, and *flowop*. Every defined entity must have a user-assigned name that is mainly used to print per-process and per-thread statistics. A fileset is a named collection of files. To define a fileset, a user specifies its name, path, number of files, and a few other optional attributes. Listing 1 shows two filesets with 1,000 files of 128 KB size that will be located in the `/tmp` directory.

```
define fileset name="test1",path="/tmp",
                              entries=1000,filesize=128k
define fileset name="test2",path="/tmp",
              entries=1000,filesize=128k,prealloc=80
```

Listing 1: Examples of fileset definitions

A Filebench run proceeds in two stages: fileset preallocation and an actual workload execution. By default, Filebench does not create any files in the file system and only allocates enough internal file entries to accommodate all defined files. To actually create files, one should specify the percentage of files to precreate with the `prealloc` attribute. Listing 1 shows how Filebench precreates 800 files in the fileset `test2`—80% of 1,000.

The reason for Filebench not to precreate all (or any) files is that certain workloads include file creates. When a workload-defined file create operation should be executed, Filebench picks a non-existent file entry in a fileset and creates the file. The total number of simultaneously existing files in a fileset can never exceed the fileset size at any point during a Filebench run. If a workload tries to create a file but there are no more non-existent file entries, then an internal `Out-of-Resources` event is triggered, which can be interpreted either as an end of the run or an error, depending on the user's objective. Consider a WML snippet in Listing 2 that can be used to measure peak file create rate. At first, the fileset is empty and the workload starts to create files in a loop. When the workload tries to create the 10,001st file, Filebench graciously exits and reports the measurements. This happens because quit mode is set to `firstdone`; more information on this and other quit modes is described later in this article. Note that delete operations reduce the number of existing files and can balance out the file create operations.

```
set mode quit firstdone

define fileset name="fcrset",path="/tmp",
              entries=10000,filesize=16k

define process name="filecreate",instances=1 {
 thread name="filecreatethread",instances=2 {
  flowop createfile name="crfile",filesetname="fcrset"
  flowop closefile name="clfile"
 }
}

run
```

Listing 2: WML snippet to measure file create performance

WML processes represent real UNIX processes that are created by Filebench during the run. Every process consists of one or more threads representing actual POSIX threads. The attribute `instances=`*N* instructs Filebench to replicate the corresponding processes and threads *N* times. Listing 2 defines one process named `filecreate` with two identical threads. WML allows users to define any number of identical or different processes containing any number of identical or different threads. Listing 3 demonstrates a more complex workload description with five processes in total. Three processes contain one reader thread and two writer threads; two other processes contain four identical threads that create and delete files. All processes and threads run simultaneously.

```
define process name="testprocA",instances=3 {
  thread name="reader",instances=1 {
    flowop openfile name="readop",filesetname="testset"
    flowop readwholefile name="readop",iosize=4k
    flowop closefile name="closeop1"
  }
  thread name="writer",instances=2 {
    flowop openfile name="readop",filesetname="testset"
    flowop writewholefile name="writeop",iosize=4k
    flowop closefile name="closeop2"
  }
}

define process name="testprocB",instances=2 {
  thread name="crdelthread",instances=4 {
    flowop createfile name="createop",filesetname="testset"
    flowop closefile name="closeop3"
    flowop deletefile name="deleteop",filesetname="testset"
  }
}
```

Listing 3: Example of defining multiple different processes and threads

Every thread executes a loop of flowops. Flowop is a representation of a file system operation and is translated to a system call by Filebench: e.g., the `createfile` flowop creates a file and the `write` flowop writes to a file. Table 1 lists the most common WML flowops, which cover the majority of operations that real applications execute against a file system. When Filebench reaches the last flowop defined in a thread, it jumps to the beginning of the thread definition and executes flowops repeatedly until a quit condition is met (e.g., requested runtime elapsed).

Flowop	Description
`openfile`	Opens a file. One can specify a virtual file descriptor to use in the following flowops.
`closefile`	Closes a file referenced by a virtual file descriptor
`createfile`	Creates a file. One can specify a virtual file descriptor to use in the following flowops.
`deletefile`	Deletes a file
`read`	Reads from a file
`readwholefile`	Reads whole file even if it requires multiple system calls
`write`	Writes to a file
`writewholefile`	Writes whole file even if it requires multiple system calls
`appendfile`	Appends to the end of a file
`statfile`	Invokes stat() on a file
`fsync`	Calls fsync() on a file

Table 1: List of most frequently used flowops. In addition, Filebench supports a number of directory, asynchronous I/O, synchronization, operation limiting, and CPU consuming and idling operations.

Filebench uses *Virtual File Descriptors* (VFDs) to refer to files in flowops. VFDs are not actual file descriptors returned by `open()`; instead, users assign VFDs explicitly in `openfile` and `createfile` flowops. Later, these VFDs can be used in flowops that require a file to operate on. Listing 4 provides an example where the attribute `fd` is used to specify two different VFDs. First, the thread opens one file, assigning it VFD 1 and creates another file with VFD 2. Then the thread reads from one file and writes to another, keeping both files open and referring to them by their VFDs. Finally, both files are closed. This represents a simple copy workload in WML. Note that VFDs are per-thread entities in Filebench: a VFD in one thread does not impact an identically numbered VFD in another thread.

VFDs specified in `openfile` and `createfile` must not be opened prior to the flowops execution. Therefore, in most of the cases it is necessary to explicitly close VFDs with a `closefile` flowop. Other flowops that require a VFD (e.g., `read`) will open a file automatically if the corresponding VFD is not opened yet. If the `fd` attribute is not specified in a flowop then Filebench assumes that it is equal to zero. This is a useful convention for a large class of workloads that keep only one file open at a time (see Listings 2 and 3). Describing such workloads in WML does not require specifying the `fd` attribute, which streamlines the workload description further.

```
set mode quite firstdone

define fileset name="testfset",path="/tmp",
        entries=10000,filesize=4k,prealloc=50

define process name="filecopy",instances=2 {
  thread name="filecopythread",instances=2 {
    flowop openfile name=opfile",
            filesetname="testfset",fd=1
    flowop createfile name="crfile",
            filesetname="testfset",fd=2
    flowop readwholefile name="rdfile",
            filesetname="testfset",fd=1
    flowop writewholefile name="wrfile",
            filesetname="testfset",fd=2
    flowop closefile name="clfile1",
            filesetname="testfset",fd=2
    flowop closefile name="clfile2",
            filesetname="testfset",fd=1
  }
}
```

Listing 4: Simple file copying expressed in WML

When opening a file, Filebench first needs to pick a file from a fileset. By default this is done by iterating over all file entries in a fileset. To change this behavior one can use the `index` attribute that allows one to refer to a specific file in a fileset using a unique index. In most real cases, instead of using a constant number for the index, one should use custom variables described in the following section.

Filebench supports a number of attributes to describe access patterns. First, one can specify an I/O size with the `iosize` attribute. Second, one can pick between sequential (default) and random accesses. Sequential patterns usually make sense only if a file is kept open between the flowop executions so that the operating system can maintain the current position in a file. When the end of a file is reached, sequential flowops start accessing the file from the beginning. Third, for random workloads, one can specify the working set size in a file using the `wss` attribute. Finally, direct and synchronous I/Os are supported as well.

The very last line of a WML file usually contains a `run` or `psrun` command. These commands tell Filebench to allocate the defined filesets, spawn the required number of UNIX processes and threads, and, finally, start a cycled flowops execution. Both commands take the duration of the run in seconds as an argument; the `psrun` command in addition takes a period with which to print performance numbers.

To generate a workload described, e.g., in a `workload.f` WML file (`.f` is a traditional extension used by Filebench), one executes the `filebench -f workload.f` command. A non-abortive run terminates under two conditions. First, the run can be time-based; this is the default mode and if the `run` command does not have any arguments, then the workload will run for one minute only. Second, a Filebench run might finish if one or all threads completed their job. To specify Filebench's quit mode, a `set mode quit` command can be used. In Listing 2, we change the quit mode to `firstdone`, which means that whenever one of the threads runs out of resources (e.g., there are no more non-existent files to create), Filebench stops. Another scenario is when a thread explicitly declares that it completed its job using the `finishoncount` or `finishonbytes` flowops. These flowops allow one to terminate a thread after a specific number of operations completed (e.g., writes or reads) or a specific number of bytes were read or written by a thread.

In the end of the run, Filebench prints a number of different metrics. The most important one is operations per second. This is the total number of executed flowop instances (in all processes and threads) divided by the runtime. For flowops that read and write data, Filebench also prints the throughput in bytes per second. Finally, one can measure the average and distribution of latencies of individual flowops. In addition, Filebench can maintain and print statistics per process, per thread, or per flowop.

Long-time Filebench framework users might be surprised that we described Filebench's run as a non-interactive experience. In fact, before version 1.5, Filebench supported interactive runs: a console in which one could type workloads and execute various commands. However, one of the big changes in v1.5 is the elimination of interactive mode. The majority of experienced users did not use non-interactive runs. Beginners, on the other hand, made a lot of systematic mistakes in interactive mode (e.g., did not drop caches or remove existing filesets between runs). In v1.5, therefore, we made a strategic decision not to support interactive mode. This further helped reduce the total amount of code to maintain.

Advanced Features
In this section, we highlight some advanced Filebench features. They were either less known before or were just recently added in version 1.5. Listing 5 demonstrates most of these features.

Variables
Filebench supports two types of variables: regular and custom. Variable names, irrespective of their type, are prefixed with a dollar sign. With a few exceptions, variables can be used instead of constants in any process, thread, or flowop attribute. Regular variables hold constant values, are defined with the `set` keyword, and are mainly used for convenience. It is considered a good style to define all parameters of the workload (e.g., I/O sizes or file numbers) in the beginning of a WML file and then use variables in the actual workload definition; it also facilitates easier changes to the workload. Listing 5 demonstrates how the `$iosize` regular variable is used to set I/O size.

```
set $iosize=4k
set $findex=cvar(type=cvar-normal,min=0,max=999,
          parameters=mean:500;sigma:100)
set $off=cvar(type=cvar-triangular,min=0,max=28k,
          parameters=lower:0;upper:28k;mode:16k)

enable lathist

define fileset name="test",path="/tmp",entries=1000,
          filesize=32k,prealloc=100

eventgen rate=100

define process name="testproc1" {
  thread name="reader",memsize=10m {
    flowop read name="rdfile",filesetname="test",
          indexed=$findex,offset=$off,iosize=$iosize
    flowop closefile name="clsfile1"
    flowop block name="blk"
  }
  thread name="writer",memsize=20m {
    flowop write name="wrfile",
          filesetname="test",iosize=$iosize
    flowop closefile name="clsfile2"
    flowop opslimit name="limit"
  }
  thread name="noio",memsize=40m {
    flowop hog name="eatcpu",value=1000
    flowop delay name="idle",value=1
    flowop wakeup name="wk",target="blk"
  }
}
```

Listing 5: Demonstration of some advanced Filebench features

The use of custom variables (`cvar`) powerfully enables any Filebench attribute to follow some statistical distribution. Distributions are implemented through dynamically loadable libraries with a simple and well-defined interface that allows users to add new distributions easily. When Filebench starts, it looks for the libraries in a certain directory and loads all supported distributions. We ported the Mtwist package [4] to the custom variables subsystem; this immediately made Filebench support eight distributions, and this number is growing.

In Listing 5 the `indexed` attribute of the `rdfile` flowop follows the distribution described by the `$findex` custom variable. The `$findex` variable uses a normal distribution with values bounded to the 0–999 range. The minimum and maximum bounds are in sync with the number of files in the fileset here—1,000. Distribution-specific parameters—mean and standard deviation (sigma) in case of a normal distribution—are specified with the `parameters` keyword. As we mentioned in the Basic Functionality section, Filebench by default picks files from a fileset in a rotating manner and the indexed attribute can pick specific files. Assigning findex makes Filebench access some files more frequently than others using a normal distribution from a custom variable. This simulates a real-world scenario in which some files are more popular than others.

Earlier Filebench versions actively used the so-called random variables, which are essentially similar to custom variables. But we found random variables limiting because the number of supported distributions was small, and adding more distributions required significant knowledge of Filebench's code base. In version 1.5 we replaced random variables with custom variables (random variables are still supported for backward compatibility but will be phased out in the future).

Synchronization Primitives

When a workload is multithreaded, it sometimes makes sense to emulate the process by which requests from one thread depend on requests from other threads. For this, Filebench provides the block, wakeup, semblock, and sempost flowops. They allow Filebench to block certain threads until other threads complete the required steps. Listing 5 shows how a reader thread blocks in every loop until the noio thread wakes it up.

The ability to quickly define multiple processes and synchronization between them was one of the main requirements during Filebench framework conception. The task for Sun Microsystems engineers at the time was to improve file system performance for a big commercial database. Setting up TPC-C [7], database, and all of the required hardware was expensive and time-consuming for an uninvolved file system engineer. The key for simulating database load on a file system was how log writes cause generic table updates to block. With this use case in mind, Filebench's WML was designed, and a corresponding oltp.f workload personality was created and then validated against the real database. Having the Filebench framework and a workload description in WML gave engineers the time to focus just on the file system tuning task.

CPU and Memory Consumption

Filebench provides a hog flowop that consumes CPU cycles and memory bandwidth. Conversely, the flowop delay simulates idle time between requests. Also, when defining a thread, one must specify its memory usage with the memsize attribute. Every thread consumes this amount of memory and performs reads and writes from it. In Listing 5 the noio thread burns CPU by copying memory 1,000 times and then sleeps for one second per loop.

Speed Limiting

In many cases one wants to evaluate system behavior under moderate or low loads (which are quite common in real systems) instead of measuring peak performance. Filebench supports this with the flowops iopslimit (limits the rate of data operations only) and bwlimit (limits the bandwidth). In Listing 5, the reader thread issues only 100 reads per second (or fewer if the system cannot fulfill this rate). The command eventgen sets the rate, which is global for all processes and threads.

Complex Access Patterns

Originally Filebench supported only simple access patterns: uniformly random and sequential. We added the offset attribute which, in combination with custom variables, allows one to emulate any distribution of accesses within a file. For example, for virtualized workloads with big VMDK files, we observed that some offsets are more popular than others [6]. In Listing 5, the writer thread accesses file's offsets following a triangular distribution.

Latency Distribution

Measuring only the average latency often does not provide enough information to understand a system's behavior in detail. We added latency distribution profiling with the enable lathist command to Filebench [3].

Composite Flowops

In WML one can define a flowop that is a combination of other flowops. This is especially useful in cases when one wants to execute certain group of flowops more frequently than other flowops. The attribute iters can be used to repeat regular or composite flowops. In addition, Filebench's internal design allows users to easily implement new flowops in C. We do not provide examples of composite or user-defined flowops in this article but offer documentation online [2].

File System Importing

Another upcoming feature in Filebench v1.5 is importing existing file system trees. Older versions of Filebench could only work with trees that it generated itself. This new feature allows one to generate a file system with a third-party tool (e.g., Impressions [1]), or use a real file system image and run a Filebench workload against this file system.

Data Generation

Earlier, Filebench versions generated all zeros or some arbitrary content for writes. In v1.5, we are introducing the notion of a *datasource*, which can be attached to any flowop. Different datasources can generate different types of data: one controlled by some entropy, duplicates distribution, file types, etc. This new feature is especially important for benchmarking modern storage systems that integrate sophisticated data reduction techniques (e.g., deduplication, compression).

Predefined Workloads

It is important to understand that Filebench is merely a framework, and only its combination with a workload description defines a specific benchmark. The framework comes with a set of predefined useful workloads that are especially popular among users. We are often asked about the details of those workloads. In this section, we describe the three most frequently used Filebench workloads: Web-server, File-server, and Mail-server.

What does a simple real Web-server do from the perspective of a file system? For every HTTP request, it opens one or more HTML files, reads them completely, and returns their content to the client. At times it also flushes client-access records to a log file. Filebench's Web-server workload description was created with exactly these assumptions. Every thread opens a file, reads it in one call, then closes the file. Every 10th read, Filebench's Web-server appends a small amount of data to a log file. File sizes follow a gamma distribution, with an average file size of 16 KB. By default, the Web-server workload is configured with 100 threads and only 1,000 files. As described later in the section, it is almost always necessary to increase the number of files to a more appropriate number.

Filebench's File-server workload was also designed by envisioning a workload that a simple but real File-server produces on a file system. Fifty processes represent 50 users. Every user creates and writes to a file; opens an existing file and appends to it; then opens another file and reads it completely. Finally, the user also deletes a file and invokes a stat operation on a file. Such operation mix represents the most common operations that one expects from a real File-server. There are 10,000 files of 128 KB size defined in this workload by default.

The Mail-server workload (called `varmail.f`) represents a workload experienced by a `/var/mail` directory in a traditional UNIX system that uses Maildir format (one message per file). When a user receives an email, a file is created, written, and fsynced. When the user reads an email, another file is opened, read completely, marked as read, and fsynced. Sometimes, users also reread previously read emails. Average email size is defined as 16 KB, and only 16 threads are operating by default.

In addition to the workloads described above, Filebench comes with OLTP, Video-server, Web-proxy, and NFS-server macro-workloads and over 40 micro-workloads. It is important to recognize that workloads observed in specific environments can be significantly different from what is defined in the included WML files. This is an intrinsic problem of any benchmark. The aforementioned workloads are merely an attempt to define workloads that are logically close to reality and provide common ground for evaluating different storage systems. We encourage the community to analyze their specific workloads, define them in Filebench's WML [5], validate the resulting synthetic workloads against the original workloads, and contribute WML descriptions to Filebench.

Best Practices

In this section, we share several important considerations when using the Filebench framework. These considerations originated from many conversations that we had with Filebench users over the past seven years.

File system behavior depends heavily on the data-set size. Using Filebench terminology, performance results depend on the number and size of files in defined filesets. It is almost always necessary to adjust fileset size in accordance with the system's cache size. For example, the default data-set size for Filebench's Web-server workload is set to only 16 MB (1,000 files of 16 KB size). Such a data set often fits entirely in the memory of the majority of modern servers; therefore, without adjustments, the Web-server workload measures the file system's in-memory performance. If in-memory performance is not the real goal, then the number of files should be increased so that the total fileset size is several times larger than the available file system cache. Specific data set-to-cache ratio varies a lot from one environment to another.

Similarly, it is important to pick an appropriate duration of an experiment. By default, timed Filebench workloads run for only 60 seconds, which is not enough time to warm the cache up and cover multiple cyclic events in the system (e.g., bdflush runs every 30 seconds in Linux). Our recommendation is to monitor file system performance and other system metrics (e.g., block I/O and memory usage) during the run and ensure that the readings remain stationary for at least ten minutes. We added a `psrun` command to Filebench 1.5 that prints performance numbers periodically. Using these readings, one can plot how performance depends on time and identify when the system reaches stable state. Anecdotally, we found that such plots often allow one to detect and fix mistakes in experimental methodology early in the evaluation cycle.

As with any empirical tool, every Filebench-based experiment should be conducted several times, and some measure of the results' stability needs to be calculated (e.g., confidence interval, standard deviations). To get reproducible results it is important to bring the system to an identical state before every run. Specifically, in a majority of the cases, one needs to warm the cache up to the same state as it would be after a long run of the workload. In other words, the frequently accessed part of the data set (as identified by the storage system) should reside in the cache. Therefore it is preferable to start the workload's execution with a cold cache, wait until the cache warms up under the workload, and then, if appropriate, report performance for warm cache only. Note, however, that regardless of whether the cache is cold or warm, in order to ensure sufficient I/O activity for a *file system* benchmark, the workload size should exceed the size of the system memory (historically it was considered at least 2×).

Furthermore, before executing an actual workload, Filebench first creates filesets, so parts of the filesets might be in memory before the actual workload runs. This might either benefit or hurt further workload operations. We recommend to drop caches between the fileset preallocation and the workload run stages. To achieve that for standard Linux file systems add

```
create fileset
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"
```

before the `run` or `psrun` commands. The `system` command allows one to execute arbitrary shell commands from WML.

Users often want to measure file system performance while varying some workload parameter. A typical example is benchmarking write or read throughput for different I/O sizes. We found it convenient to write shell scripts that generate WML files for different values of the same parameter (I/O size, in this example). It is also helpful to save any generated `.f` files along with the results so that later on one can correlate the results to the exact workload that was executed.

Future

Filebench is a powerful and very flexible tool for generating file system workloads. We encourage storage scientists, engineers, and evaluators to explore the functionalities that Filebench offers to their fullest. We plan to improve Filebench further to accommodate changing realities and user requests. Here, in conclusion, we only mention major directions of future work.

First, Filebench provides a unique platform for both quick development of new workloads and (formal or informal) standardization of workloads that are universally accepted as reflecting reality. Standardization only makes sense if a broad storage community is adequately involved. Moreover, we believe the involvement should be continuous rather than one-time because the set of widespread workloads changes over time. To that end, we plan to make further efforts to build stronger community and conduct BoF and similar meetings at storage conferences. We invite everyone interested in this direction to communicate with us [2].

Second, from the technical side, Filebench currently translates flowops to POSIX system calls only. However, the internal design of Filebench is based on flowop engines that map flowops to specific low-level interfaces. Specifically, we consider adding NFS and Object interfaces to Filebench. With the advent of very fast storage devices, overheads caused by the benchmark itself become more visible. In fact, we fixed several performance issues in Filebench over the last few years. More generally, we plan to work on the overhead control system that is integrated into Filebench itself.

Although Filebench already has rudimentary support for distributed storage systems benchmarking, it is not enough from both functionality and convenience points of view. We plan to design and implement features that will make Filebench practical for distributed system users.

Acknowledgments

We would like to acknowledge several people from the Filebench community who contributed and continue to contribute to Filebench: George Amvrosiadis is a devoted Filebench user and developer who added multiple improvements, with importing the external file system tree feature being just one of them; Sonam Mandal and Bill Jannen contributed to duplicated and entropy-based data generation; Santhosh Kumar is the individual behind custom variables; and, finally, we borrowed both expertise and code on statistical distributions from Geoff Kuenning.

References

[1] N. Agrawal, A. C. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Generating Realistic Impressions for File-System Benchmarking," in *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[2] Filebench: http://filebench.sf.net.

[3] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, "Operating System Profiling via Latency Analysis," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[4] G. Kuenning, Mersenne Twist Pseudorandom Number Generator Package, 2010: http://lasr.cs.ucla.edu/geoff/mtwist.html.

[5] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok, "Extracting Flexible, Replayable Models from Large Block Traces," in *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, 2012.

[6] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok, "Virtual Machine Workloads: The Case for New Benchmarks for NAS," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '13)*, 2013.

[7] Transaction Processing Performance Council, TPC Benchmark C, Standard Specification: www.tpc.org/tpcc, 2004.

Calling SREs and Sysadmins

Register Today!

SREcon16

April 7–8, 2016 | Santa Clara, CA, USA

www.usenix.org/srecon16

Call for Participation Now Open!

SREcon16 EUROPE

July 11–13, 2016 | Dublin, Ireland

www.usenix.org/srecon16europe

Call for Participation Now Open!

LISA16

Dec. 4–9, 2016 | Boston, MA, USA

www.usenix.org/lisa16