



The **DOs** and **Don'ts**

of File System Benchmarking

BY VASILY TARASOV, ZHEN CAO, MING CHEN, AND EREZ ZADOK

As we know, “everything is a file in Unix” ^[14]. Documents, executables, hard disk drives, memory, resource utilization statistics, and even system settings are all accessed and modified through files. As such, the file system is the cornerstone of any Unix derivative and its performance and efficiency are extremely important for overall system speed. Over the years, a myriad of file systems with diverse goals, designs, and implementations have been proposed and developed. File systems that persistently store and retrieve data are, of course, of special importance. The one common element of all file systems is that they all provide an identical (POSIX) API. For example, an application using a POSIX-compliant file system can portably create, open, read, and write a file; make, list, and modify hierarchical directory trees; access and change a file’s metadata; and so on.

Running applications execute file system operations and often need to wait for operations to complete. In many production systems the amount of time that applications spend waiting for file system operations is the main contributor to the total execution time. In this case, the application is said to be *I/O bound* and the best way to improve its performance is by increasing file system speed. But one cannot improve what cannot be accurately measured. File system and storage benchmarking are therefore crucial processes for both evolutionary and disruptive improvements of computer system performance.

In practical scenarios, benchmarking is often used to compare alternatives—for example, when deciding which file server to purchase from several available models or when choosing among different local file systems. Sometimes users need to pick between completely different storage architectures (e.g., local vs. shared storage). With the advent of new storage technologies, users often wonder if upgrading to expensive Flash memory or PCM will improve file system performance enough for their demands. Alas, modern file systems have a large number of configuration parameters that strongly impact their performance [16]; selecting the optimal parameter values can therefore mitigate I/O bottlenecks and eliminate or postpone the need for costly upgrades [4]. To answer these and similar questions, one needs to properly and fairly compare the performance of different systems. Yet this is not a simple task, mainly because so many major and minor details need to be taken into account.

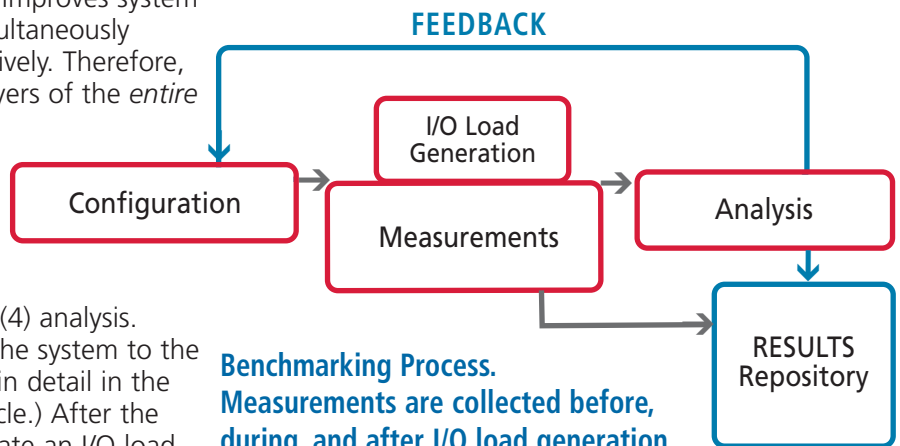
Inexperienced users (and even experts) make mistakes in benchmarking methodology [18, 21]. In this article we describe important principles and techniques to properly assess the performance of file-system-based storage systems. The guidelines presented here are the result of many years of experience with benchmarking file systems and storage in a diverse set of research and engineering projects.

File systems are an integral part of the I/O stack, which consists of at least I/O libraries, VFS layers, file system layers, block layers, and hardware. For a network file system, the networking stack plays an additional, important role in performance. When virtualization is employed, parts of the I/O stack are duplicated inside a VM, effectively doubling the total number of layers [20]; similarly, storage virtualization like RAID, LVM, and stackable file systems add even more layers. Every single layer has a number of alternative configurations: the specific configuration used greatly impacts the overall performance of an I/O-bound application. It is nearly impossible to measure the performance of a file system in isolation, and even if it were possible, measuring a “middleware” layer such as a file system without the rest of the software, OS, and hardware it runs on would not yield meaningful results. The same file system (e.g., FreeBSD’s *ffs* [11]) would definitely exhibit different performance when run in a different environment. Even smaller differences, like the prefetch size at the block layer, can improve (or degrade) performance of a file system. Moreover, due to software and hardware complexity, even if increasing the values of two parameters independently improves system performance, increasing both values simultaneously might impact system performance negatively. Therefore, in practice one always benchmarks all layers of the *entire* I/O stack rather than an individual file system layer.

We divide the benchmarking process into four major steps as depicted in the figure at right:

(1) system configuration, (2) I/O load generation, (3) measurements, and (4) analysis. System configuration brings all parts of the system to the desirable states. (We will cover this step in detail in the System Configuration section of this article.) After the system is configured one needs to generate an I/O load with appropriate characteristics (described in the I/O Load Generation section) and simultaneously start measuring system behavior (explained in the Measurements section). In fact, to capture the initial and final system states, measurements are also taken before and after the I/O load generation. Finally, the results from measurements are analyzed, visualized, and conclusions are made (as detailed in the Analysis section).

Benchmarking is a time-consuming, highly repetitive and iterative process; note the blue *feedback* arrow in the figure. When the goal of an experiment is to compare several alternative configurations or workloads, the whole process is often repeated multiple times for every possible setup. Frequently, the results of the analysis demonstrate unexpected results, and require changes to configurations, workloads, measurements, or all. For example, if during the analysis step file-system memory utilization turns



out to be lower than expected, it might further reveal an inadequately low limit set on the cache size. Then one might decide to repeat the exercise with a higher cache limit. Similarly, intense benchmarking can tickle bugs not seen before, causing system crashes or revealing serious “performance bugs,” which requires code changes (and sometimes design changes), after which the entire set of benchmarks must be rerun. This feedback cycle sometimes has to be repeated many times; not surprisingly, researchers have found the time spent on this benchmarking cycle is far longer than the initial development of the first system prototype.

Before going into the heart of this article we have several important disclaimers. First, benchmarking is a field with a lot of open debates and conflicting opinions. We convey our point of view and hope that this will promote a healthy discussion on how to conduct storage evaluation properly. Second, as with any rules, there are exceptions to benchmarking rules. We try to cover the most common scenarios, but there are plenty of legitimate cases when the rules we present can and even should be changed. Third, we focus on performance, but this is not the only metric by which one decides which file system is “better.” The set of features supported, acquisition cost and total costs of ownership, and power consumption, are just a few examples of other factors that impact the decision on which storage solution to use. Fourth, in real life, experimenters are limited by practical time constraints (e.g., software release cycles, paper deadlines). Therefore, we note both ideal approaches that assume an unlimited amount of time and less perfect, but practical techniques. We believe it is acceptable to deviate somewhat from the ideal methodology provided and that it is well understood which assumptions are made.



System Configuration

System configuration is the very first step of the benchmarking process. Although this step is absolutely crucial for the success of the whole process, it is often unfairly disregarded or considered trivial. Without a detailed understanding of the system’s configuration, it is not possible to properly analyze results, deduce generalized conclusions, or repeat identical or derived experiments at a later time. The three main goals of system configuration are to (1) learn a configuration space, (2) consciously and reliably set system parameters to the required values, and (3) collect exhaustive configuration descriptions for future reference.

Most often the complete configuration space is not known before benchmarking starts. If that is the case, then all configuration parameters and their possible values should be carefully studied, understood, and listed. The configuration space can be roughly divided into two parts: parameters that can be modified only manually (e.g., by installing different network or storage devices), and parameters that are easy to change programmatically (e.g., file system types or `pdflush` frequency). Although some parameters cannot be set programmatically, they can often be read programmatically (e.g., a disk drive’s model is visible in kernel boot logs). It is important to record all parameters even if one expects them to remain unchanged throughout all planned experiments. Benchmarking plans have a tendency to change. In our experience, recording the complete information on the system configuration saves a lot of time when plans inevitably change. In addition, knowing the details of hardware setup significantly facilitates the analysis stage. Specifically, we recommend to note HDD and SSD sizes, models, speeds, and capabilities (e.g., whether the `TRIM` command is supported and what is the parallelism level of an SSD); network card models and speeds; hardware RAID controller models and settings (e.g., read/write cache size, modes); the number and models of CPUs, cores, and their speeds; main memory size and topology; system BIOS settings; and more. Certain software parameters also remain static and cannot be easily changed but should still be recorded (e.g., kernel, distribution, and library versions). Configuration harvesting should be as automated as possible (e.g., using shell scripts).

Parameters that can be modified programmatically should be set to the appropriate values before every experiment. This ensures that between experiments (which sometimes spans multiple weeks), system parameters are not changed accidentally. Many configuration parameters have default values. For example, one often does not need to specify block or inode sizes when formatting a file system. We highly recommend never to trust the defaults because they vary with environments. By assuming that defaults are the same everywhere, one risks comparing two different configurations without even knowing so. Therefore, set all parameter values explicitly. As we mentioned earlier, one must not limit these rules to the file system settings. All I/O stack parameters (e.g., `pdflush` frequency, dirty memory high/low watermarks) should be set explicitly. If a software volume manager or iSCSI is used, then the corresponding settings need to be examined and set as well. We also recommend collecting the list of running processes and verifying that no unexpected processes that potentially disrupt the experiments are running (e.g., `cron` jobs, background daemons).

After setting all parameter values, we recommend reading all of them back. This is an extra safety measure that ensures all the settings actually took place and there were no mistakes in the configuration-setting commands. We find it necessary to read parameters twice per experiment—in the beginning and at the end of the run to ensure that parameters did not change during the experiment. Save the output of all commands and store those configurations along with the results.

Some modern storage systems support advanced features like deduplication, compression, encryption, etc. It is important to be aware of such features. For example, if compression is supported, it will be important at the I/O load generation step to write data with a consciously selected compression ratio. At times one might decide to temporarily disable certain advanced features to simplify the results analysis or to identify a feature's performance cost and efficiency.

In modern infrastructures, hardware is often shared (e.g., a JBOD connected through a SAN or an NFS server). Traffic from other users might disrupt the experiments, causing the results to be sensitive to the environment and hurting the ability to reproduce the results. Whenever possible, try to ensure that nobody else uses shared resources during the experiments. For example, one can disable remote logins to the machines and limit NFS mounts to a set of specific nodes. If it is not possible to technically restrict access to the shared resources, ask other users not to use the resources during your experiments. Although this is not a guaranteed approach, it still increases the chances of undisturbed results.

Professionals often resort to using several identical machines in parallel to speed up experiments. For instance, one can split a set of experiments to execute one part on one machine while the other part runs on another machine. In ideal cases, this allows one to complete experiments twice as fast. Using two machines is not ideal because even seemingly identical machines do not perform exactly the same. In fact, it was shown that the performance of modern HDDs can vary as much as 20% within the same model line [10]. However, because of practical time constraints, it is often necessary to use multiple machines. In this case we recommend spreading the instances of exactly the same experiments across different nodes and then reporting the average values along with a variance metric (e.g., standard deviation). This way the differences between machines are included in the performance numbers (instead of inadvertently contributing to the performance differences between workloads or configurations). Note that the benefit of such an approach is that the results describe system performance not on one specific machine but rather on a population of "identical" machines, which is more general and has a higher value.

Another common approach for speeding up experiments is to artificially limit the RAM size so that smaller datasets can be used while still generating ample I/O activity (more on the dataset size in the section on I/O Generation). In this case both the initial dataset creation and the cache warmup phases run faster. The hidden assumption is that if the RAM-to-dataset ratio remains the same, then performance with larger datasets (and larger RAM) remains the same. Logically this makes sense but we are not aware of any studies that have verified this assumption. Therefore, if time permits, it is better not to limit RAM size artificially and use the complete dataset sizes. In case of a time constraint, we recommend ensuring that after limiting the RAM size, there is still the same number of DRAM slots accessible to every CPU node (relevant in NUMA nodes only).

Before every experiment, the system needs to be configured to exactly the same state as in other experiments in the series—except, of course, the differences dictated by the experiment's objective (e.g., comparing one configuration to another). Configuration steps should be designed so that they bring the system to the same starting point before the actual workload runs. For file system benchmarking, we recommend formatting the file system before every experiment. Remounting the file system is not enough because the history of file system usage in previous experiments can impact performance in the following experiments. After mounting a file system, it is sometimes desirable to age the file system to a realistic (though reproducible) state because file system performance deteriorates over time [1]. If aging cannot be performed (e.g., it takes too long or the tools are not available), then using an empty file system is an alternative. At the very least, we recommend filling the file system with as much data as it expects to store in production because many storage devices perform more slowly as more data is written to them. For instance, HDD throughput is higher on the outer tracks (which are filled first) because they rotate at a higher linear speed. SSDs, on the other hand, use unwritten space for pre-erased blocks, so their performance also decreases with higher utilization. In fact, we recommend overwriting SSDs completely before experiments to trigger garbage collection typical in the long-running production systems. Executing the **TRIM** command on an SSD may not make sense because at best it brings the SSD to the optimally performing state, which usually is not the goal of a realistic evaluation. Worse, however, is that the SSD can be left in a different state before every experiment because **TRIM** can run asynchronously and it is not known when the Flash Translation Layer (FTL) actually gets to erase all blocks.

It is worth noting that modern systems often inherently have randomness in them. For example, some file systems allocate disk blocks randomly to provide consistent performance over the lifetime of a file system [15]. In this case it is practically impossible to bring the system exactly to the same state. Every single configuration procedure will produce a slightly different setup. Such differences, however, are perfectly fine. They either will not impact performance numbers significantly, or if they do, then the user will become aware of system's sensitivity to the initial state (e.g., report higher standard deviations). After that, one can decide if it is an acceptable performance variance for a specific environment.

During configuration and the steps that follow, it is important not to omit any errors returned by the tools or the system. We recommend stopping the experiment immediately if any single command exits with a non-zero status or if there is an error in the kernel or system logs. After the error is fixed or identified as irrelevant, the run should be resumed from scratch.

For distributed file systems there are many servers and clients in the mix and the steps mentioned above need to be repeated by each node.



I/O Load Generation

After successfully configuring the target systems, the next step is to generate the I/O load with the required characteristics, which we call the *I/O workload*. The two most important things that users need to keep in mind during this process are: (1) understanding **what** benchmarks or applications do in as much detail as possible, and (2) thinking carefully about **how** to run the benchmarks or applications.

Below we describe four common methods for generating file system loads—micro-benchmarks, macro-benchmarks, I/O traces, and application-level benchmarks—each applicable to specific scenarios [21].

- **Micro-benchmarks.** These benchmarks are designed to exercise a few (usually one or two) types of file-system operations—e.g., measuring how many creates-per-second a file system can achieve. Such benchmarks are useful when the goal is to measure the impact of a small change in a system, to (later) better understand the results of macro-benchmarks, or when users want to isolate the effects of a specific part of the system. The results of micro-benchmarks are more valuable and meaningful if presented with results from other types of benchmarks. A few examples of the commonly used micro-benchmarks include fio [6], iозone [3], and some of Filebench's personalities [5].
- **Macro-benchmarks.** The goal of macro-benchmarks is to provide an estimate of the system's performance when deployed in production. These benchmarks exercise multiple file system operations and are designed by observing and characterizing real-world workloads and then simulating them. Examples of macro-benchmarks include Filebench's Web/Mail/File personalities [5] and SPEC SFS®.
- **I/O traces.** Storage developers recognized early that in complex cases simple counters are not enough to analyze system behavior, and therefore added the ability to record every operation in the system. I/O traces are a collection of timestamped records about file-system or block-level operations captured on a specific system. After a trace is collected in one system, it can then be replayed on other systems to evaluate their performance. Trace replay can provide an accurate estimate of storage performance, but users still need to ensure that the actual traces used are representative of the real workload. For example, traces should cover extended periods of time to capture as many as possible occurring usage patterns.

There are several methods for replaying I/O traces, which cause debates on which replay method is the most appropriate [21]. Some replay traces with the original timings. However, traces are usually collected on older and slower systems, so the use of original timings does not stress newer and faster storage enough. Another approach is to replay traces as fast as possible ignoring the timings and fixing the total number of outstanding requests. In this case the interdependencies between requests are ignored and the results might differ from reality. Finally, as a middle ground, one can replay the trace with a fixed speedup factor and limit the maximum number of simultaneously outstanding requests. When using trace replay for evaluation, we recommend using all of the above methods and making appropriate conclusions. To the best of our knowledge, there are no widely available file-system trace replayers; the `btreplay` tool can be used for replaying block-level traces.

- **Application-level benchmarks.** Previously described benchmarks only mimicked real applications. Application-level benchmarks, on the other hand, exercise targeted systems by deploying real applications on them. The benchmarks then simulate the way users operate such applications in real life. For example, the TPC-C benchmark [22] requires the deployment of a real database software which it exercises in a manner representative of complex OLTP application environments, portraying the activity of a real-life wholesale supplier.

I/O workloads observed in real life and produced by benchmarks are often characterized using a set of common metrics, such as operation ratios, I/O size distributions, level of parallelism, sequentiality, and dataset sizes. The premise behind such characterization is that the performance of storage systems depends mainly on macroscopic statistical properties rather than on small details of workloads [19]. For example, the operation ratio is the percentage of each file system's operation type (e.g., read, write, create) in the overall mix. Similar but coarser characterization distinguishes metadata-intensive (high percentage of namespace operations) vs. data-intensive workloads. File system performance is typically fairly sensitive to this characteristic because namespace-management operations are designed and implemented quite differently from the data management. In an evaluation, one usually needs to use the benchmark that produces a workload with characteristics close to the ones observed in the target environment.

One especially important property of a workload is the dataset size. Some workloads can fit their datasets completely in RAM; and then the underlying storage does not impact file system performance. More frequently, however, the dataset size is larger than the size of the file system cache, and, therefore, both memory and storage subsystems are exercised. We recommend setting the dataset size several times larger than the available RAM size. Even better is to experiment with different dataset sizes, keeping all other workload characteristics unchanged. We would like to stress that file system performance can be very sensitive to dataset and RAM sizes. We have shown that a dataset size increase as small as 6MB can result in almost 10 times lower throughput [18].

In many production deployments, workloads do not cause the system to reach its peak performance. In other words, there is a lot of idle time between requests submitted to the file system. When evaluating a storage system one might decide to respect realistic idle time and observe system behavior in a non-stressed scenario (e.g., measure request latencies when the rate of incoming requests is relatively small). Another approach is to observe how the system behaves under the highest load, which allows one to measure system's peak performance. We recommend evaluating systems varying the load from modest to highest levels—and reporting performance of all points along this continuum.

Several nontraditional aspects of workloads have become important in recent years. If a storage system supports deduplication or compression, then benchmarks should generate content with appropriate compression or deduplication levels. Many older benchmarks wrote zeros or arbitrary data to a file system, both of which do not evaluate the system properly. One needs to pick a compression ratio that is expected in a specific environment—e.g., if a target storage system is used for storing documents, then a typical compression ratio is in the 3 to 5 times range [9]. Newer versions of fio, Filebench, and other benchmarks support compression features. DEDISbench [13] is an I/O benchmark for deduplication systems and it includes the distribution of duplicate content as one of its inputs.

The next step in benchmarking is to understand the condition of when to terminate the I/O load. There are two approaches to specify a stopping condition: *time-based* and *job-based*. A time-based approach specifies the fixed duration of the benchmark run. An example of this category is running stress tests to see how many requests a web server can handle during the peak hour of a day. In contrast, job-based approaches specify the amount of work that needs to be completed for each benchmark run. For example, when testing the speed of sorting a 1TB text file in a big data system, a job-based approach is more convenient and makes more sense than a time-based approach. Note that in this case, the running time actually becomes a relevant metric reflecting the performance of the system. In practice, both approaches are valid, and choosing the right one depends on the use case.

While deciding how long an experiment should run, users also need to consider the typical cycles in system operations, such as cache flushes, log wrapping, major and minor compactions, etc. The benchmark should at least cover multiple iterations of the longest operation cycle, so that all modes of the system behavior are evaluated. Moreover, some benchmarking tools have a warmup phase before the actual run of simulated workloads to let the system reach a steady state. During the warmup phase usually no metrics are collected. We consider it unnecessary to treat the warmup phase differently because it is often important to understand what was happening to a system with respect to performance and other metrics during the warmup. So, users should treat the warmup as part of the run, collect all measurements periodically, and later, if needed, users can discard data collected from the warmup stages of the experiments.

After finishing the benchmark run, some recommend executing an *fsync* operation. However, we consider this a superfluous step as it really depends on the specific characteristics of the real workloads. If no *fsync* operations occur at the end of the real workload, there is no need to add an *fsync* after the benchmark. In fact, if the working set size and the duration of the benchmark are sufficiently large, then

dirty data flushes should occur many times during the run and are included in the performance results.

Last but not least, we encourage the avoidance of custom-made storage benchmarks. This complicates any results verification, reproduction, and hurts their trustworthiness.



Measurements

While generating the workload, one needs to collect metrics that both measure system performance and describe system behavior in a broader sense. Measurements serve as an input to the results analysis—the final step of benchmarking. The quantity and the quality of the measurements to a large extent determine the speed and efficiency of results analysis.

The metrology of file system benchmarking asks two main questions: (1) **what** to measure, and (2) **how** to measure. The first question is about what are the important metrics we need to collect to facilitate the following analysis; and the second question is about how to accurately and efficiently measure those relevant metrics. The answers to both questions are not absolute, and depend largely on the storage devices used, file system type, workloads, and the analysis goal. In this section we discuss general guidelines to help answer these questions.

—What to Measure—

The task of identifying relevant metrics can be challenging considering the complexities of modern file and storage systems; knowing the common characteristics of “good” metrics can thus simplify this task. A good metric has four decisive characteristics: (1) informative, (2) well-defined, (3) quantitative, and (4) simple. Because the purpose of metrics is to help engineers and researchers understand the system, each metric should be informative enough to facilitate one or more analysis tasks, including sanity checks, environment monitoring, system behavior analysis, performance evaluation, and troubleshooting. An informative metric is often a clear indicator of system environment (e.g., temperature, network load), resource utilization (e.g., CPU or memory use), system performance (e.g., throughput, latency), or system events (e.g., page faults, interrupts, allocations of new blocks).

A good metric should also be well-defined and without any ambiguities. A well-defined metric should have a clear context under which the metric is collected. The same type of metric can be significantly different in different contexts. For example, in a setup that uses Network File System (NFS), throughput measured at the application level (say, in MB/sec) is not equal to the throughput measured at the NFS client level because of the client-side page cache; nor will application level throughput match the throughput measured at the RPC level because of the client side persistent cache such as FSCache [8]. Furthermore, the throughput measured at the block layer of the NFS server is another completely different metric.

A well-defined metric should also have clear and meaningful boundaries that mark the start and the end of the metric. Many benchmarking studies fail in this regard, for example, by choosing a somewhat arbitrary warmup period and reporting only performance metrics after that warmup. The problem is that a fixed warmup period is not necessarily meaningful: the warmup may be intended to fill a cache, but the degree of cache fullness after the fixed warmup period varies in different workloads. Then the seemingly reasonable warming up will cause the reported results to depend on unknown and probably different initial cache states. As already mentioned in the section on I/O Load Generation, a better alternative is to mark the start of the metric at the very beginning of the experiment (before warmup), and measure until the work is finished.

Being quantitative is another important feature of good metrics. Quantitative metrics tend to be more accurate and reproducible than qualitative metrics; and quantitative metrics also lead to more objective analysis than qualitative ones. As Lord Kelvin—the famous physicist who accurately measured the absolute zero temperature—said, “When you measure what you are speaking about and express it in numbers, you know something about it, but when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.” For example, describing throughput as “bursty” when writing data to an empty page cache is less clear than “a throughput of 1GB/sec drops to 80MB/sec after 5 seconds.”

The fourth characteristic of a good metric is simplicity, which makes the metric easy to measure. A simpler metric also makes the analysis following the measurement more intuitive and less error-prone.


Now that we understand what desirable features to look for when searching for good metrics, we discuss how to identify relevant metrics. Measuring a single performance metric is a bad practice; the metrics we collect for an experiment should be comprehensive enough to conduct a wide range of analysis tasks following the experiment. Therefore, we should collect metrics in all of the following categories:

- **Performance metrics.** These metrics, such as throughput and latency, are direct indicators of performance, and thus the most important. There may be multiple throughput metrics and multiple latency metrics. In NFS, for example, there are ops/sec and MB/sec throughput metrics at different levels of the storage and networking stack; there are several latency metrics ending at different layers as well (e.g., latency upon page-cache-hits, and latency upon FSCache-hits).
- **Resource utilization.** Use of CPU, memory, storage I/O, and networking are also important metrics. They are direct indicators of system efficiency, and are helpful in understanding system behavior and troubleshooting.
- **Metrics of the setup.** We should measure metrics that validate our setup and serve as evaluation baselines. For example, is the caching device indeed much faster than the primary storage device, and what is the maximum possible speed-up with 100% cache-hit-ratio? If possible, we should measure the metrics of our setup by ourselves because the metrics provided by manufacturers are often too simple or too optimistic to be trusted.
- **System-specific metrics.** For example, compression and deduplication ratios reported by the systems that support these features.
- **Metrics of external events.** These metrics, such as the request rate of external clients, are especially important in benchmarking environments we do not fully control.
- **Environment metrics.** Such as temperature of the server machines, and the degree of congestion of the storage networks.


—How to Measure—



After knowing the metrics to measure, the next step is to carry out the measurements. A good measurement should have the following features:

- **Accuracy.** Being accurate means the measured value of a metric is close to the real value of the metric. Accurate measurements form the foundation to make correct decisions.
- **Precision and stability.** Being precise means the results of multiple measurements of the same metric



Rack-mount networking server
Designed for BSD and Linux Systems
Up to **5.5Gbit/s** routing power!

Made for  FreeBSD


PERFECT FOR

- ▶ BGP & OSPF routing
- ▶ Firewall & UTM Security Appliances
- ▶ Intrusion Detection & WAF
- ▶ CDN & Web Cache / Proxy
- ▶ E-mail Server & SMTP Filtering
- ▶ Anti-DDoS and clean pipe filtering


1 Gbit/s Copper	Ports	Chipset
L800-G808-1	8x Gbe RJ-45 ports	8x Intel i210 AT; PEX8618
L800-G808-2	8x Gbe RJ-45 ports	8x Intel i210 AT; PEX8618
L800-G428-1	4x Gbe RJ-45 ports	1x Intel i350 AM4
L800-G428-2	4x Gbe RJ-45 ports	1x Intel i350 AM4
1 Gbit/s SFP (Fiber)	Ports	Chipset
L800-S406-1	4x Gbe SFP ports	i350-AM4
10GbE Copper	Ports	Chipset
L800-T202-1	2x 10Gb RJ-45 ports	Intel X540
L800-T203-1	2x 10Gb RJ-45 ports	Intel X540
10GbE SFP+ (Fiber)	Ports	Chipset
L800-X204-1	2x 10Gb SFP+	Intel 82599ES
L800-X205-1	2x 10Gb SFP+	Intel 82599ES
L800-X405-1	4x 10Gb SFP+	Intel 82599ES; PEX8724

KEY FEATURES


- ▶ 6 NICs w/ Intel igb(4) driver w/ bypass
- ▶ Hand-picked server chipsets
- ▶ Netmap Ready (FreeBSD & pfSense)
- ▶ Up to 14 Gigabit expansion ports
- ▶ Up to 4x10GbE SFP+ expansion




DESIGNED FOR
FreeBSD



DESIGNED FOR
GNU / Linux



DESIGNED FOR
FreeBSD
Network Appliances



DESIGNED FOR
Sense

Designed. Certified. Supported

contactus@serveru.us | www.serveru.us | 8001 NW 64th St. Miami, FL 33166 | +1(305) 421-9956

are close to each other. High precision makes us confident in the consistency and reproducibility of the measurements.

- **Efficiency.** Efficiency is important to ensure our measurement does not introduce excessive system loads that skew the system performance.
- **Automation.** Automated measurement is more reproducible and less prone to uncertainties caused by human errors.

We also recommend the following practices when measuring metrics of file and storage systems:

- **Repeat experiments and measurements multiple times.** Multiple runs make us confident of the precision; for metrics that vary a lot, multiple runs, especially when shown as box plots (or CDFs), describe a more factual picture of the range of the metric. For example, a distribution of network latencies, instead of a single latency, would better illustrate the dynamic nature of networking and multi-layer structure of storage.
- **Measure metrics periodically during experiments instead of only once at the end.** Systems are dynamic and metrics often vary during the run. When shown in time series graphs, frequent measurements can capture the dynamics of the metric and filter out random noise. In our experience a period of 10 seconds between measurements provides enough granularity without causing an excessive overhead.
- **Tools like `vmstat`, `iostat`, and `nfsstat` are not the original sources of measurements.** These tools read numbers from the `/proc` file system, perform extra calculations, and then print results to the user. The calculations, however, might not be trivial and in combination with poor labeling may lead to wrong conclusions. One common example is the service time metric in `iostat` that is valid for old HDDs and makes little sense for highly-parallel SSDs [2, 12]. Therefore, when possible, we recommend sticking to the original data sources and performing calculations on your own during the analysis phase.
- **Record common timestamps for every measurement.** These timestamps can help coordinate events from different subsystems (e.g., storage and networking, client and storage server). When the experiment involves multiple machines, it is important to synchronize the time using the Network Time Protocol (NTP).
- **Ensure that the performance penalties of measurements are negligible** by running experiments with and without the measurements.
- **Save measurements along with the configuration and workload descriptions used in the experiment.** It is convenient to make every experiment's folder completely self-sufficient.

→ Analysis

The analysis step either reaches the final goal of the whole evaluation or guides the design of additional experiments (see *Feedback* arrow in the figure on page 15). In this section we share several analysis practices that we have found useful.

- **Formatting.** Measurements are often collected in wildly diverse and not post-processing-friendly formats—e.g., outputs of `iostat` and `vmstat` tools are quite different and cannot be directly fed to Gnuplot or a spreadsheet tool. We find it convenient to first format all measurements to some common structured format. In our experience a CSV format provides the best common ground: it is accepted by most analysis or visualization tools, can be read by humans, and parsed by programs. Files in CSV format are also convenient for sharing with other involved parties who might be using different toolsets. We usually use CSV files where the first column contains the timestamps and the remaining columns have corresponding measurement values.

After running many experiments, an evaluator ends up with many experimental results. Having a consistent file-naming convention and directory structure is highly useful. This not only simplifies navigation through results for humans, but also allows us to easily use the same analysis and visualization scripts for all experiments.

- **Visualization.** With so many numbers collected, it is practically impossible to comprehend them without summarizing the numbers with graphs. We found Gnuplot and Python's `matplotlib` to be powerful tools that are easy to script; they dramatically accelerate the plotting of many experiments.

We always recommend starting with time series graphs—the graphs where the X-axis shows the time and the Y-axis shows one of the measured metrics. For example, throughput-time series graphs allow us to quickly understand if performance was stable during the whole run. If not (e.g., due to the caching effects in the beginning of the run), when reporting the average throughput, one might consider only the throughput after the cache became warm. Similarly, plotting the memory utilization versus time allows us to confirm the point at which the cache became full. In many experiments one can see periodic dips in performance due to periodic cache flushes or segment compactions.

When preparing graphs that compare several independent experiments, it is important to show the variability of every metric. Specifically, instead of using bar graphs with a single average number per experiment, box plots show more detailed statistical information using the same space (mean vs. median, outliers, upper and lower quartiles, and more).

- **Sanity checks.** One of the first tasks of analysis is to verify that the measurements make general sense. For example, if the throughput at the application layer turns out to be several times higher than the HDD bandwidth (and the dataset does not fit in RAM according to your experimental setup), then something is probably wrong with the experiment. It can be that the dataset is much smaller than expected, data is being completely deduplicated, or something else. Another example is when many metadata operations are observed in an experiment that was designed to be data-intensive. Of special concern should be when the results appear to be “too good to be true” as they are often not. Another task of sanity check is to verify that all logs do not contain error and warning messages.

- **Overhead.** Frequently we see that people refer to degradation in performance as overhead—e.g., if after enabling deduplication, throughput falls from 100MB/sec to 80MB/sec, then it is sometimes said that the overhead of deduplication is 20%. This, however, is not precise. Overhead is about the resource utilization, such as CPU cycles, memory usage, I/O bandwidth, but not about performance [7]. For the above example, I/O bandwidth usage might have increased twofold (e.g., for bringing the deduplication index to RAM), so the actual I/O overhead is 100%. A contrary example is when, after enabling some feature, performance improves by 50%. It is not a good practice to report only the fact that the feature increases performance 1.5 times. In fact, it is quite possible that CPU utilization grew from 25% to 75%, which means that CPU overhead of the feature is 300%. Often the increase in performance is not acceptable if the overhead is too high, so the analysis should report both performance improvements and resource utilization.

- **Performance.** Two basic metrics of file system performance are throughput and latency. Depending on the context, throughput can be defined as IOPS (I/O operations per second) or as MB/sec. We recommend to always start from IOPS because this is a more universal and less ambiguous metric that applies equally well to both metadata and data operations. Depending on the operation mix and I/O

RootBSD

Premier VPS Hosting

RootBSD has multiple datacenter locations,
and offers friendly, knowledgeable support staff.
Starting at just \$20/mo you are granted access to the latest
FreeBSD, full Root Access, and Private Cloud options.



www.rootbsd.net

size, different IOPS can translate differently to MB/sec. For one workload, high IOPS might still mean low MB/sec (metadata operations or random I/O with small I/O sizes), while low IOPS for another workload might translate to high MB/sec (large multi-MB write operations). After the IOPS metric has been analyzed, one can translate it to MB/sec as necessary.

Note that for some software, high throughput is important, whereas for others, low latency has a higher value. Average latency for a single-threaded system (one request is in flight at all times) can be computed as a reciprocal of IOPS. However, most of the storage stack is multi-threaded, which increases throughput manifold when multiple requests are submitted simultaneously. But if too many requests are submitted, then latency can start to grow as a result of queueing. We recommend analyzing how throughput depends on latency—e.g., putting throughput on the X-axis and average latency on the Y-axis allows us to see for which throughput system latency is still acceptable. Monitoring system queues often helps better understand this behavior.

Conclusions

There is a wide variety of storage hardware, software, and workloads. Every time a new system is built or an old one upgraded, decisions need to be made on which storage setup to use. After functionality, performance is the second most important system characteristic. Although it is easy to determine if the functionality offered by the storage layer suffices, it is much harder to ensure that performance is at the required level. Storage benchmarking is the discipline devoted to answering this complex question. Yet people have been benchmarking storage for a long time and we still see a lot of poor practices: unclear configuration, inappropriate workloads, poorly selected metrics, and erroneous analysis. In this article we have presented several tips and techniques that we learned the hard way over the years. We hope the information presented here will be useful to readers and we also hope and encourage the community to use and publish more quality storage performance evaluations in the future. ●

Thank you!

The FreeBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.



The
FreeBSD
FOUNDATION

Are you a fan of FreeBSD? Help us give back to the Project and donate today!
freebsdfoundation.org/donate/

Iridium



Platinum



Gold



Silver



Please check out the full list of generous community investors at freebsdfoundation.org/donate/sponsors

AUTHOR BIOS

EREZ ZADOK received his 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 the faculty in 2001. His current research interests include file systems and storage, operating systems, energy efficiency, performance and benchmarking, security, and networking. He has received the SUNY Chancellor's Award for Excellence in Teaching, the U.S. National Science Foundation (NSF) CAREER Award, two NetApp Faculty awards, and two IBM Faculty awards.

MING CHEN received his BS and MS in Computer Science from Beihang University in China. He is a fifth-year PhD candidate at Stony Brook University, working with Professor Erez Zadok in the File Systems and Storage Laboratory (FSL). His research interests include analysis, design, and implementation of distributed storage systems and cloud-computing systems.

ZHEN CAO received his BS in Software Engineering from Fudan University, China. He is a third-year PhD candidate at Stony Brook University, working with Professor Erez Zadok in the File Systems and Storage Laboratory (FSL). His interests include benchmarking and auto-tuning complex storage systems.

VASILY TARASOV is a researcher at IBM Almaden Research Center. He received his PhD from Stony Brook University in 2013. His interests include system performance analysis, design and implementation of distributed systems, and efficient I/O stacks for ultra-fast storage devices. He maintains and contributes extensively to the popular Filebench benchmarking framework.

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] M. Brooker. Two Traps in iostat: %util and svctm. (2014)
- [3] D. Capps. IOzone file system benchmark. www.iozone.org.
- [4] E. Zadok, A. Arora, Z. Cao, A. Chaganti, A. Chaudhary, and S. Mandal. Parametric Optimization of Storage Systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*. (2015)
- [5] Filebench. <http://filebench.sf.net>.
- [6] fio—flexible I/O tester. <http://freshmeat.net/projects/fio/>.
- [7] G. Heiser. Systems benchmarking crimes. <https://www.cse.unsw.edu.au/gernot/benchmarking-crimes.html>.
- [8] D. Howells. FS-Cache: A Network File System Caching Facility. In *Proceedings of the 2006 Linux Symposium*, volume 2. (2006)
- [9] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*. (2009)
- [10] E. Krevat, J. Tucek, and G. R. Ganger. Disks are like snowflakes: No two are alike. In *Proceedings of HotOS XIII: The 13th USENIX Workshop on Hot Topics in Operating Systems*. (2011)
- [11] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197. (August 1984)
- [12] B. Mildren. Monitoring IO performance using iostat & pt-diskstats. (2013)
- [13] J. Paulo, P. Reis, J. Pereira, and A. Sousa. DEDISbench: A Benchmark for Deduplicated Storage Systems. In *Proceedings of the International Symposium on Secure Virtual Infrastructures (DOA-SVI)*. (2012)
- [14] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375. (1974)
- [15] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*. (2002)
- [16] P. Sehgal, V. Tarasov, and E. Zadok. Optimizing Energy and Performance for Server-Class File System Workloads. *ACM Transactions on Storage (TOS)*, 6(3). (September 2010)
- [17] SPEC SFS@ 2014. <https://www.spec.org/sfs2014/>.
- [18] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of HotOS XIII: The 13th USENIX Workshop on Hot Topics in Operating Systems*. (2011)
- [19] 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)
- [20] 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)*. (2013)
- [21] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80. (May 2008)
- [22] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification. www.tpc.org/tpcc. (2004)