The Case for Application-Specific Benchmarking

Margo Seltzer, David Krinsky, Keith Smith, Xiaolan Zhang
Harvard University

Most performance analysis today uses either microbenchmarks or standard macrobenchmarks (e.g., SPEC, LADDIS, the Andrew benchmark). However, the results of such benchmarks provide little information to indicate how well a particular system will handle a particular application. Such results are, at best, useless and, at worst, misleading. In this paper, we argue for an application-directed approach to benchmarking, using performance metrics that reflect the expected behavior of a particular application across a range of hardware or software platforms. We present three different approaches to application-specific measurement, one using vectors that characterize both the underlying system and an application, one using trace-driven techniques, and a hybrid approach. We argue that such techniques should become the new standard.

1 Introduction

According to David Patterson, “For better or worse, benchmarks shape a field” [13]. If this is indeed the case, then bad benchmarks will lead us astray. And, in fact, we have seen evidence that the industry’s obsession with benchmarking can be misleading [11].

As researchers, our goals for benchmarking and performance analysis are twofold. We strive to understand why systems perform as they do, and we seek to compare competing ideas, designs, and implementations. In order for this second goal to be met, it is essential that the benchmarks enable meaningful comparisons; raw benchmark numbers are useless to other researchers and customers unless those raw numbers map into some useful work. In this paper, we present techniques that provide more meaningful benchmark results.

The primary reason that today’s benchmark results lack meaning is that they are not designed to address any particular user’s performance problem. Instead, they are intended to quantify some fundamental aspect of the performance of a system, whether it be hardware or software. Although such *microbenchmarks* can be useful in understanding the end-to-end behavior of a system, in isolation they are frequently uninformative about overall system performance—primarily because that performance is highly workload-dependent [17].

The goal of this paper is to demonstrate the need for application-specific benchmarking and suggest three different methodologies that researchers can apply to provide this type of benchmarking.

The rest of this paper is structured as follows. Section 2 describes some of the most commonly used benchmarks, emphasizing how they are unable to answer simple questions. Section 3 presents a trivial case study that demonstrates where conventional measurements fall down. Section 4 presents our taxonomy of application-specific benchmarking techniques, and Section 5 concludes.

2 Benchmarks and Answers

Let us assume that benchmarks are designed to answer questions. From a consumer’s point of view, the most commonly asked question is, “What machine (or software system) will best support my workload?” The question may be asked in any one of a number of domains: CPU or overall system performance, file system performance, or application performance.

Consider the task of purchasing “the fastest PC available.” How can one determine which machine to buy? The first option is to limit oneself to a particular architecture (e.g., an Intel-based PC) and select the system with the greatest megahertz rating. At first glance, this requires no benchmarks and yields the fastest machine. The reality is not this simple. There are a plethora of questions left unanswered: Should the machine be a Pentium II or Pentium Pro? Which cache size should it have? How much memory should it have? Which disk should be used? Which is the best Ethernet controller? This approach rapidly boils down to a random selection of components.

A second approach is to look for the machine with the best SPEC rating. If the intended application domain of this machine is software development on UNIX, this may, in fact, be a reasonable criterion. However, let us assume that the machine will be used with a GUI-based operating system (e.g., Windows). SPECmarks do not necessarily reflect anything meaningful about the performance of such systems [4][5]. If the machine is destined to be a Web server, the right answer may be different yet. The bottom line is that "fastest machine" is context-sensitive; it depends on the use to which the machine will be applied. Even collections of system microbenchmarks [2][10], which are useful for comparing the behavior of low-level primitives, do not provide an indication of which aspects of the system's behavior are important for a particular application.

Let us consider a second, and potentially simpler task, "Find the fastest file server." There are a number of standard file system benchmarks such as Bonnie [1], IOStone [12], and the Andrew File System Benchmark [7], as well as some that are designed particularly for remote file service, such as LADDIS [19] and SPEC SFS [16]. Unfortunately, while Bonnie reveals a great deal about the underlying disk system, it does little to measure file system performance. IOstone tries to depict file system behavior more realistically, but its file system model bears little relationship to reality [18]. Andrew was designed to measure file system scalability, but does little to reveal the behavior of the underlying system itself [7]. SPEC SFS is a well-specified scalable benchmark. However, its operation mix is designed to "closely match today's real-world NFS workloads" [16]. This seems to imply a single "real-world NFS workload." Furthermore, it does nothing to tell us about workloads that are not NFS-based.

Today's benchmarks are not designed to describe the performance of any particular application. If our goal is to produce performance numbers that are meaningful in the context of real applications, then we need a better set of tools.

3 A Case Study

In this section, we present a simple example that demonstrates how some of today's benchmark results can be uninformative. Our system domain is file systems and our application domain is netnews. Netnews has a unique, identifiable access pattern, but one that is not captured by traditional benchmarks. A file server hosting news will have an abundance of small files in very large directories (directories typically contain hundreds or thousands of files). The steady-state behavior of a news file system is that millions of files are created and deleted each week, while users simultaneously read, in a seemingly random fashion, from the file system [17].

An examination of netnews access patterns and performance bottlenecks revealed that updating the access time of each news article every time it was read was problematic [6]. As the access time is not critical to the application's behavior, an acceptable performance optimization is to modify the file system to omit this update. In this section, we will attempt to use standard benchmarks to reveal the value of this modification. We will compare two file systems: the Fast File System (FFS) and NewsFS (FFS modified to omit the access time update). We chose three commonly used file system benchmarks: the Andrew File System benchmark [7], Bonnie [1], and IOStone [12]. Table 1 presents the results of these three benchmarks

	FFS	**NewsFS**
Andrew (sec)	8 (0.64)	8 (0.64)
Bonnie per-char read (KB/s)	9740 (41)	9758 (36)
IOStone (IOStones/s)	1,969,402 (311016)	1,979,093 (363677)

Table 1: Standard Benchmark Results. These results are the arithmetic means of ten iterations with standard deviations given in parentheses. The two file systems are practically indistinguishable from one another.

	FFS	**NewsFS**
16KB file read (MB/s)	4.516 (0.034)	6.073 (0.028)

Table 2: News Benchmark Results. These results are the arithmetic means of ten iterations with standard deviations given in parentheses. By simulating the application in question, this benchmark is able to reveal a significant difference between the two file systems.

Unfortunately, the results for these common benchmarks are nearly indistinguishable, which could lead someone to conclude that the modified file system is of little use. We then constructed a benchmark designed to model a news workload. The benchmark reads 8192 16KB files, simulating users reading news articles. These results are shown in Table 2.

Not surprisingly, these results indicate that NewsFS is a win in this situation. This simple example demonstrates our principle: in order to effectively evaluate file system (or any system) performance, a benchmark must reflect the intended workload.

4 What to do

As we saw in the last section, a relatively common and somewhat simple workload is not well-served by any of today's standard benchmarks. We are not the first to observe this particular problem, and in fact, Karl Swartz developed a benchmark specific to Net News [17]. However, it is both impractical and unnecessary to design a new benchmark for every new application. Instead, we prefer to develop a benchmarking methodology that can be applied to any application and across several different domains.

We have developed three approaches to application-specific benchmarking: *vector-based, trace-driven,* and *hybrid.* Each methodology addresses a different set of benchmarking requirements and constraints. The vector-based approach characterizes an underlying system (e.g., an operating system, or JVM) by a set of microbenchmarks that describe the behavior of the fundamental primitives of the system. The results of these microbenchmarks constitute the system vector. We then characterize an application by another vector that quantifies the way that the application makes use of the various primitives supported by the system. The dot-product of these two vectors yields a relevant performance metric. In our trace-based methodology, we preprocess traces of actual usage, producing user and activity profiles that can stochastically generate loads representative of the activity applied to the system. The hybrid approach uses a combination of vector-based and trace-based analysis. The following sections describe the three approaches in more detail.

4.1 Vector-Based Methodology

The fundamental principle behind vector-based performance analysis is the observation that in a typical computer system, each different primitive operation, whether at the application, operating system, or hardware level, takes a different amount of time to complete. Traditional benchmarking techniques ignore this fact and attempt to represent the overall performance of a computer system or subsystem as a scalar quantity. However, for many modern applications, this representation proves inadequate, as an application's performance may be highly dependent on a specific subset of the functionality provided by the system.

As their name suggests, vector-based techniques address this problem by representing the performance of underlying system abstractions as a vector quantity. Each component of this system characterization vector represents the performance of one underlying primitive, and is obtained by running an appropriate microbenchmark.

To obtain a meaningful result, the system vector is combined with a second vector that represents the demand an application places on each underlying primitive. Each component represents the application's utilization of the corresponding primitive, either as a frequency (e.g., the number of times a system call is used) or another quantity (e.g., the number of bytes transferred through a certain pathway). The application's performance on a given underlying system can be calculated by taking the dot product of the two vectors: the time due to each component is the time that its primitive operation takes to execute, multiplied by the demand placed upon that primitive. Assuming that the vectors represent a thorough cross-section of the abstractions used, the total execution time should be the sum of these components.

This approach is similar to the *abstract machine model* presented by Saavedra-Barrera [14], where the underlying system is viewed as an abstract Fortran machine and each program is decomposed into a collection of Fortran abstract operations called *AbOps*. The *machine characterizer* obtains a machine performance vector, whereas the *program analyzer* produces an application vector. The linear combination of the two vectors gives the predicted running time. This approach requires extensive compiler support for obtaining the accurate number of *AbOps* and is highly sensitive to compiler optimization and hardware architecture [15]. The techniques are limited to programming languages with extremely regular syntax. Our model is more general in the sense that it can be applied to any system that consists of a well-defined API between the application and the underlying platform [3].

Brown's analysis of Apache web server performance demonstrates this approach [3] in the operating system domain. First, using the hBench benchmarking suite [2], he measured the performance of a number of operating system abstractions on the NetBSD operating system. Then, using system call tracing facilities, he characterized Apache's usage of operating system primitives as it served a single HTTP request. The components of the two resulting vectors did not match exactly as the methods used to derive them were quite different. However, it was possible to transform the application vector using a change of basis so that each component was either matched with an appropriate microbenchmark or approximated by a related one. The results, though preliminary, were excellent; by multiplying the application vector by the system characterization vectors of multiple platforms (both Intel and Sparc-based platforms, including one multiprocessor), the results produced a correct ordering and a close approximation to the absolute performance of all of the systems.

More recently, we have begun to apply this technique to a collection of standard UNIX utilities (e.g., ls, tar). We enhanced hBench to include a few simple file system microbenchmarks and using this extended suite, we can produce encouraging predictions. To date, our predictions always rank the different systems correctly (e.g., machine #1 performs better than machine #2) and the relative performance of the systems is reasonably well-predicted (e.g., we predict that machine 1 outperforms machine 2 by a ratio of 5:1 while the actual ratio is 6:1), but the absolute time predictions are still not as accurate as we would like (e.g., we predict a running time of 0.018 seconds and the actual running time is 0.055 seconds). Of the three metrics (relative ranking, relative performance, absolute performance), we are most interested in obtaining accurate results for relative performance, so we find these preliminary results encouraging.

A second domain in which vector-based analysis appears promising is in the evaluation of Java Virtual Machines (JVMs). In this case, we also use a microbenchmark suite to characterize the performance of the machine's primitives. We then use application profiling to quantify an application's use of these primitives. Each process produces a vector, and once again, the dot-product of the two vectors gives an indication of application-specific performance.

JVMs introduce an additional complexity: garbage collection. As the garbage collector runs asynchronously with respect to the application, its impact is nondeterministic. Our strategy is to characterize the performance of the garbage collector as a function of the activity induced by an application and add this to the dot-product result. More formally, let $\vec{v}$ be the JVM performance vector, $\vec{a}$ be the application performance vector, and $gc(\vec{a})$ be the function that computes the garbage collection overhead as a function of the application vector. Then, our application-specific performance metric is: $\vec{v} \cdot \vec{a} + gc(\vec{a})$

4.2 Trace-Based Methodology

For certain applications, performance is dependent, not only on the application itself, but on the particular stream of requests submitted to that application. Examples of applications in this category include Web servers, Web proxies, and database servers. In such cases, a simple vector-based approach is not sufficient. Instead, we use a trace-based benchmarking methodology, retaining the ability to tailor the benchmark to a particular workload.

Rather than characterizing the underlying system and application, we characterize the trace, and use the characterization to generate a stochastic workload that can be executed as a benchmark. Consider the case of a Web server. Different Web sites have dramatically different access patterns [8], and evaluating a server's performance is best done in the context of the expected load on a site. A Web server's workload has two major components: the set of files that comprise the site and the access pattern of the users visiting that site. Our Web server benchmarking methodology consists of processing Web server logs, creating a set of files to mimic the site and creating a set of user profiles that describe the access patterns of different sets of users [9]. Using the file set and user set, the benchmark generates a series of requests to the Web server, producing a load that is closely representative of the original load on the server. The load is parameterized in terms of files and users, allowing us to scale the load to reflect growth patterns or changes in the user community and to run "what-if" experiments.

We are applying a similar methodology to evaluating proxy caches. The overall structure of the benchmark is the same, with a bit of additional complexity. The benchmark environment must also simulate the Web servers from which the proxy obtains pages. When processing the trace, we create characterizations of the loads placed on these external servers, including their response time distribution, and then use these characterizations to model servers during the benchmark run.

This trace-based methodology has a number of advantages beyond offering site-specific benchmarking. First, while it is simplest to use actual traces to create workload characterizations, it is not essential. One can construct a characterization manually to mimic an anticipated server load. This manual characterization can then be used to drive the benchmark workload generator, producing performance results for a not-yet-existent workload. Second, the stochastic workload generation tool has the capability to analyze a collection of logs that have accumulated over time. Using the pattern of accesses present in the existing logs, we can generate predicted future access patterns to facilitate capacity planning. Third, logs gathered on one machine can generate characterizations that can be used to test other hardware/ software configurations. Fourth, log files are often considered proprietary to the organizations on whose servers they were produced, so such logs are rarely made available to the research community. However, the file set and user set characterizations, while enabling the generation of representative loads, are sufficiently anonymized that they can be freely distributed without revealing corporate secrets. This freedom will enable researchers to evaluate the performance and Web behavior of sites that are traditionally unavailable for such analysis.

While more complex than the vector-based methodology described in Section 4.1, our trace-based methodology provides a flexible framework in which to analyze workload-dependent applications. The resulting benchmark is specific to an application/workload combination; the ability to modify trace characterizations (i.e., “turn the knobs” to parameterize a workload) provides flexibility to specify workloads that do not yet exist.

4.3 Hybrid Methodology

The hybrid methodology combines the two approaches outlined in the previous sections. This methodology, which arose from our attempts to apply vector-based analysis to file systems, provides the best of both worlds.

At first glance, the analysis of file system performance seems to demand a trace-based approach. How else could we capture the importance of locality of reference in an application’s request stream? Unfortunately, a purely trace-based approach does not provide the insight into the underlying behavior of the system that we see when using vector-based analysis. With the vector-based approach, a researcher can determine the components of the system that have the greatest effect on the application’s performance by comparing the products of the corresponding elements of the system and application vectors. This highlights the areas where further optimization would be most beneficial.

While we can characterize the behavior of the file system via a microbenchmark suite, the interaction of successive requests in the various file system caches makes a simple characterization of an application impossible. The performance of any individual file system request will typically depend on the previous items in the request stream. A pathname lookup, for example, will perform differently depending on whether the desired translation is in the name cache. To capture the performance of a sequence of requests, we use a simulator to convert an application trace into an application vector.

First we use microbenchmarks to create a system-vector, which includes elements for the sizes of the various caches in the file system (the buffer cache, the name cache, and the metadata cache). We then use an application trace as input to a cache simulator (parameterized by the appropriate values from the performance vector), deriving the mix of operations that comprise the application vector.

The simulation pass simulates the caching behavior of the system and categorizes the application’s accesses into access patterns that have specific meanings in terms of the file system microbenchmarks (e.g., whether the targets of two successive requests are the same file or files in the same directory.) This simulation pass can be viewed as an application-specific change of basis (as discussed in Section 4.1), where the application is specified by the trace of its file system activity.

To illustrate this process more clearly, assume we have a trace of a workload. A naive approach might work as follows. Process the trace generating an application vector that identifies the number of each type of operation (read, write, create, etc.). Then multiply the application vector by a system vector derived from microbenchmark results. Unfortunately, the results of such a process are not accurate, as it assumes the performance of each operation is independent of the other operations in the trace. In reality, file systems are typically optimized for common sequences of operations, using techniques such as caching and clustered disk layout.

Using the hybrid approach, we first send the trace through our simulator to produce an application vector that identifies the number of each type of operation. This processed vector takes into account the cache behavior of the target system and also generates a more detailed list of operations. For example, instead of simply reporting the number of read operations, we report the number of logically sequential read operations that hit in the cache, the number of logically sequential read operations that went to disk, the number of random reads to cache, and the number of random reads to disk. As each of these operations has a significantly different performance profile, decomposing the class of read operations is essential for accurately predicting performance.

The result of the simulation pass is the quantification of the detailed set of operations, such as the four types of read operations mentioned above. This is the application vector that we then multiply by the system vector to yield an application-specific performance metric.

5 Conclusion

In this paper, we have argued that system performance should be measured in the context of a particular application, and we have presented three related methodologies that enable such measurements. Performance results that are not connected to real workloads provide little value to end-users and researchers. However, the application-directed benchmarking techniques we have presented provide researchers with the tools to understand the behavior of systems and to work on improving those aspects that actually matter to real users.

6 References

[1] Bray, T., Bonnie Source Code, Netnews Posting, 1990.

[2] Brown, A., Seltzer, M., “Operating System Benchmarking in the Wake of *Lmbench:* Case Study of the Performance of NetBSD on the Intel Architecture,” *Proceedings of the 1997 Sigmetrics Conference,* Seattle, WA, June 1997, 214–224.

[3] Brown, A., “A Decompositional Approach to Performance Evaluation,” Harvard University, Computer Science Technical Report TR-03-97, 1997.

[4] Chen, B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., Smith, M., “The Measured Performance of Personal Computer Operating Systems,” *ACM Transactions on Computer Systems*, 14 (1), January 1996, 3–40.

[5] Endo, Y., Wang, Z., Chen, B., Seltzer, M., “Understanding Latency in GUI-based Applications,” *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, WA, October 1996, 195–199.

[6] Fritchie, S., “The Cyclic News Filesystem: Getting INN to do More with Less,” *Proceedings of the Eleventh System Administration Conference (LISA XI)*, San Diego CA, October 1997, 99–111.

[7] Howard, J., Kazar, M., Menees, S., Nichols, S., Satyanarayanan, M., Sidebotham, R., West, M., “Scale and Performance in a Distributed System,” *ACM Transactions on Computer Systems*, 6 (1), February 1988, 51–81.

[8] Manley, S., Seltzer, M., “Web Facts and Fantasy,” *Proceedings of the Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997, 125–134.

[9] Manley, S., Courage, M., Seltzer, M., “A Self-Scaling and Self-Configuring Benchmark for Web Servers,” Harvard University Computer Science Technical Report, TR-17-97, 1997.

[10] McVoy, L., Staelin, C., “lmbench: Portable Tools for Performance Analysis,” *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, January 1996, 279–295.

[11] Mogul, J., “SPECmarks are leading us astray,” *Proceedings of the Third Workshop on Workstation Operating Systems*, Key Biscayne, FL, April 1992, 160–161.

[12] Park, A., Becker, J., “IOStone: A Synthetic File System Benchmark,” *Computer Architecture News*, 18(2), June 1990, 45–52.

[13] Patterson, D., “How to Have a Bad Career in Research/Academia,” Keynote, 1994 USENIX Symposium on Operating System Design and Implementation, `http://www.cs.utah.edu/~lepreau/osdi/keynote/slides-1up.ps.gz`.

[14] Saavedra-Barrera, R. H., Smith, A. J., Miya, E., “Machine Characterization Based on an Abstract High-Level Language Machine,” *IEEE Transactions on Computer Systems*, 38(12), December 1989, 1659-1679.

[15] Saavedra-Barrera, R. H., Smith, A. J., “Analysis of Benchmark Characteristics and Benchmark Performance Prediction,” *ACM Transactions on Computer Systems*, 14(4), November 1996, 344-384.

[16] SPEC-SFS97: Information on SPEC-SFS 2.0, `http://www.spec.org/osg/sfs97`.

[17] Swartz, K., “The Brave Little Toaster Meets Usenix,” *Proceedings of the Tenth Systems Administration Conference (LISA ‘96)*, Chicago, IL, October 1996, 161–170.

[18] Tang, D., “Benchmarking Filesystems,” Harvard University Computer Science Technical Report, TR-19-95, 1995.

[19] Wittle, M., Keith, B., “LADDIS: The Next Generation in NFS File Server Benchmarking,” *Proceedings of the 1993 Summer USENIX Conference*, Cincinnati, OH, June 1993, 111–128.