Practical Issues in Dependability Benchmarking

David Oppenheimer, Aaron B. Brown, Jonathan Traupman, Pete Broadwell, and David A. Patterson
University of California at Berkeley, EECS Computer Science Division
387 Soda Hall #1776, Berkeley, CA, 94720-1776, USA
{davidopp,abrown,jont,pbwell,pattrsn}@cs.berkeley.edu

Abstract

Much of the work to date on dependability benchmarks has focused on costly, comprehensive measurements of whole-system dependability. But benchmarks should also be useful for developers and researchers to quickly evaluate incremental improvements to their systems. To address both audiences, we propose dividing the space of dependability benchmarks into two categories: competitive *benchmarks that take the holistic approach, and less expensive* developer *benchmarks aimed at day-to-day development tasks. In this paper we differentiate the goals of these two types of benchmarks, discuss how each type might be appropriately realized, and propose simplifying assumptions for making them cost-effective.*

1. Introduction

Benchmarks serve a number of important functions. For end-users, they allow systems to be evaluated and compared prior to purchase. For researchers and system implementors, they provide a way to quantify design tradeoffs and a yardstick that helps to measure and inspire progress. Because the primary goal of benchmarks has traditionally been to allow comparison of competing systems, benchmark design has historically emphasized comprehensiveness, repeatability, representativeness of real-life situations, fairness, relevance over time, and independence from the specifics of the system under test.

Performance benchmarks have been an integral part of computer systems research for more than twenty years and have addressed these issues in various ways. History has shown that performance benchmarks are commonly used for two purposes. They are used for competitive head-to-head comparisons and evaluation of complete systems on full workloads in scenarios designed to emulate a production environment. But they are also used by developers and researchers who are attempting to evaluate design tradeoffs and measure incremental design and implementation improvements.

Because this latter use of benchmarks leads them to be run more frequently, developers and researchers commonly use a standard comprehensive benchmark as a starting point, and then make simplifying assumptions to reduce running time and cost at the expense of some sacrifice in benchmark realism or comprehensiveness. For example, researchers commonly run individual benchmarks from the SPEC CPU [18] benchmark suite, run the benchmarks on reduced data sets, or run the benchmarks for a fraction of their normal running time. Similarly, researchers may run just some of the queries of the TPC-C [17] benchmark, may run that benchmark using a scaled-down dataset and query workload, or may ignore certain tests such as those that verify proper ACID semantics. These scaled-down benchmark runs allow researchers and developers to leverage the work that has gone into creating the comprehensive version of the benchmark while avoiding the high cost of full, audited benchmark runs.

Recent commercial interest in system dependability has spurred the development of practical *dependability benchmarks* [4] [2] [19]. The recent proposals have generally defined full-scale competitive benchmarks that attempt to fully address issues such as completeness, representativeness of real-life operational environments, fairness, and repeatability. Because they strive for comprehensiveness, these monolithic benchmarks are expensive to run. Moreover, certain attributes of such dependability benchmarks make them inherently more time consuming, and therefore more expensive, to run than performance benchmarks--for example, the need to inject and measure the effect of many different kinds of perturbations and to collect data to develop a realistic error model.

We believe that as is the case for performance benchmarks, it is important for developers and researchers to be able to run dependability benchmarks in-house and on a small scale. Because they would be run more often--in the course of day-to-day research and development--these benchmarks must be much less expensive per run in terms of time and money than full-scale competitive benchmarks.

With this observation in mind, we propose dividing the space of dependability benchmarks into two categories: *competitive benchmarks* that take the comprehensive, holistic approach, and less expensive *developer benchmarks* that target specific dependability approaches, use simplified error models, and explore a subset of the space of relevant perturbations. In this paper we differentiate the goals of these two types of benchmarks and propose simplifying assumptions for making them cost-effective while sacrificing as little realism as possible. Our intention is to take an initial stab at mapping out the issues related to this one particular dimension of the benchmark design space; as a result, this paper is tackles the issue at a very high level rather than describing an actual implementation.

The remainder of this paper is organized as follows. Section 2 discusses related work in benchmarking. Section 3 discusses the goals of our framework and maps out the space of competitive versus developer benchmarks. Finally, in Section 4 we conclude.

2. Related work

Benchmarks have been recognized for many years as an essential way to objectively evaluate design choices and full systems. Benchmarks were initially used primarily to measure and compare performance. Examples of such benchmarks that are still in use today include the SPEC CPU benchmark for processor performance [18] and the TPC-C benchmark [17] for online transaction processing systems.

As commercial and research interest in system dependability has grown during the past decade, so has interest in benchmarking not just performance, but also system dependability. The systems proposed thus far generally fall into two categories: those that measure a system's overall end-to-end dependability by quantifying system response to realistic injected errors, and those that informally assess particular aspects of a system's dependability. These categories can be likened to macrobenchmarks and microbenchmarks. Proposals that fall into the first category include an e-mail server benchmark [2], a benchmark of web server response to disk failures in a RAID system [4], and a benchmark for traditional fault tolerant systems [18]. Systems that fall into the second category include ones that evaluate the robustness of UNIX utilities [11], the robustness of operating system calls [7], application response to C library errors [1], and the time it takes an application to recover from failures [21].

Madeira and Koopman describe a general framework for defining dependability benchmarks [9]. Our division of the benchmarking space into small-scale (developers) and holistic (competitive) benchmarks impacts on most of the benchmark components they describe (*e.g.,* life cycle phase, operating environment, user perspective, measures, workload, system under test, upsetload, and procedures and rules). In this paper is we elaborate on how this dimension of the benchmarking space impacts on these components. Our approach is to start with an abstract definition for a holistic benchmark, and then indicate simplifying assumptions that can be used to make smaller, cheaper, faster, more targeted versions of such a benchmark.

3. Framework proposal

We characterize a system's dependability as its ability to provide the desired quality of service in the face of internal and external perturbations. For the purposes of this paper we assume a dependability benchmark model in which an application's delivered quality of service is measured over time while a workload and various perturbations are presented to it. The workload is defined as a representative external workload, perhaps taken from a standard performance benchmark, in addition to various operator maintenance tasks. Perturbations take the form of hardware and software errors, as well as the effects of operator actions, such as scaling, upgrading, reconfiguring, or repairing the system, be they beneficial or harmful. Although incorporating human operators into dependability benchmarks presents logistical challenges, studies indicate that human error is the largest single cause of failure in large-scale Internet services [14]; it is therefore essential to include human operator activities in any representative benchmark [2].

The goal of our framework is to explore a dimension along which dependability benchmarks can be characterized. This axis can be thought of as somewhat analogous to macrobenchmarks versus microbenchmarks. Dependability benchmarks in general should be able to measure a system holistically. They should also be adaptable to function on a small scale, for relatively short, inexpensive, iterative, in-house system evaluation. In describing this space, we focus particularly on simplifying assumptions for making developer benchmarks cost-effective and therefore reasonable to run frequently.

A number of factors differentiate competitive dependability benchmarks from developer dependability benchmarks. In this section we describe those related to metrics and workload, perturbation load, and maintaining benchmark relevance and fairness over time.

3.1. Context, metrics, and workload

3.1.1. Approach-specific benchmarks

An important feature of competitive dependability benchmarks is that they measure quality of service from the user's perspective given a standard application workload. This gives the benchmarks two important properties. First, they are independent of, and portable across, the specific platforms that might be benchmarked (hardware, operating system, operating environment including operators, application, *etc.*). Equally importantly, they are agnostic to the particular dependability technique(s) used in the system under test, evaluating only the net effect of whatever techniques are used. A particular system might use one or more of various techniques to achieve dependability, for example,

- fault avoidance
- fault removal
- making code modules more robust using wrappers
- improving a system’s usability by operators
- improving a system’s ability to function in degraded mode in the face of failed components
- improving a system’s ability to recover from arbitrary failure modes using recovery techniques such as operator undo or component restart
- reducing time to detect and repair component failures, through improved system diagnosis and repair functionality.

Competitive benchmarks evaluate user-perceived system behavior in the face of a representative workload and perturbation load, so they measure the effectiveness of the mixture of dependability techniques used in the system. But the use of dependability benchmarks that is of most interest to developers on a day-to-day basis is more analogous to component-level regression tests. Developers are generally interested in assessing how much they are improving a particular system with respect to a particular dependability property (bugs, robustness, operator usability, *etc.*). In this case a holistic benchmark may obscure or dilute the impact of the particular modification that is being made.

We call targeted benchmarks that evaluate one attribute of system dependability *approach-specific benchmarks*. Such benchmarks have already been developed for use in particular domains. Examples include robustness tests for operating system interfaces and language libraries, static and dynamic correctness tests, user interface benchmarks, recovery benchmarks, and fault-tolerance benchmarks. An analogy can be drawn to running only the integer or floating point component of the SPEC CPU benchmark suite when working on improving one of those components of performance, or only a particular benchmark within an operating system benchmarking suite such as *lmbench* [10] when working on improving one aspect of operating system performance.

3.1.2. Metrics and workload

One of the crucial issues in designing a benchmark is deciding what to measure. For competitive benchmarks, it is important to measure quality of service as perceived by the system’s end-users. This takes the form of performance measures such as latency and throughput, as well as data quality measures such as correctness, numerical accuracy, and data freshness. For developer benchmarks, only one or a few of these attributes might be measured at a time. For example, increasing data storage redundancy might mitigate the decrease in throughput during a failure, but should not affect data quality during a failure. Also, in developer benchmarks these attributes might be measured at the level of individual components--for example, the response of a RAID subsystem to failure--rather than at the system’s end-user interface.

Another important issue in designing a benchmark is the workload. Generating a workload for a competitive benchmark can be costly because it requires a full-scale workload generator that draws from an accurate model of user and operator interaction with the service. A developer benchmark might simply use a random stream of requests, or one targeted to exercise just the dependability features on which the developer is working, rather than a fully realistic workload. Similarly, scripted operator tasks that are targeted at specific areas of concern could be used instead of real operators or a complete model of operator behavior.

3.2. Perturbation load

Perhaps the most important issue in designing a dependability benchmark is its perturbation model. In a competitive benchmark, the set of injected perturbations should be as realistic as possible, so that the benchmark predicts how the system will behave in a realistic production environment. This means that ideally perturbations should be taken from a trace of the perturbations encountered in a similar system deployed in a production environment. The trace should include all relevant attributes of perturbations, including error storms (correlated or cascaded errors) and failures due to operator error.

Assuming that developer benchmarks aim to assess progress in fault removal and improvements in the effectiveness of fault detection, tolerance, and recovery mechanisms, the perturbations that are tested in a developer benchmark should be chosen to maximize completeness rather than focusing on representativeness. In other words, they should focus on covering the largest possible range of perturbations rather than only those that have been detected in live systems. One reason for this is that the system under development is likely to be somewhat different from the system that is used to establish a “representative” set of perturbations. Moreover, even if the “representative” perturbation load came from a deployed instance of the system under development, changes that have been made or will be made to the system could change the type of perturbations experienced by the system.

System robustness to the widest range of perturbations possible is arguably key during development. On the other hand, it is time-consuming to instrument for and exercise every conceivable perturbation. Therefore it would be reasonable to use techniques such as control flow analysis or coverage analysis to select the minimal set of injected perturbations to maximize completeness. A less exact method would be for a designer to develop an approximate perturbation model (one based on intuition about likely perturbations rather than on real data) by analyzing the design of the system. For example, for human error, cognitive walkthrough could be used to identify the possible sources of operator error. Note that the approaches we have described for generating a perturbation load for developer benchmarks are less costly than collecting event data from real systems, in line with our goal of minimizing costs for developer benchmarks.

3.2.1. Reducing the cost of operator involvement

Another important difference between the perturbation load of competitive and developer benchmarks is in the the use of human operators as a source of perturbations.

Studies of Internet services [14], large servers [19], and the public telephone network [8] indicate that human error is the largest single cause of service unavailability. Thus, it is important to inject perturbations due to operator error. These errors may happen when an operator is performing normal maintenance tasks on a system or in the course of repairing another failure. Because human error is very system-specific, the best way to inject human errors is to use a set of live operators, giving them regular maintenance tasks and tasks related to diagnosing and repairing other injected errors [2].

Because competitive benchmarks aim to maximize realism and are run infrequently enough to make tolerable a reasonably high cost, it makes sense to incorporate humans into the benchmarking process. But because we anticipate developer dependability benchmarks to be run often, it would be too costly for them to involve humans every time. Therefore we suggest injecting simulations or manifestations of human errors during developer dependability benchmarks. First a set of human errors would be defined, based on experience with similar systems or an analysis of all possible human errors that could be made. Psychology research recognizes two general categories of human error: *mistakes*, which are errors in planning, and *slips* or *lapses*, which are errors in execution [15]. An example of a mistake is configuring a load balancing switch incorrectly due to a misunderstanding about which machines are part of a service's front-end and which are part of its back-end. An example of a slip is a typo in a command that results in deleting the email mailbox of the wrong user.

Simulating human errors without real humans can be done in two ways. It can be done by scripting operator actions at user interface level, *e.g.,* using shell scripts for command-line interfaces or GUI scripting languages for GUI interfaces. Alternatively, it can be done by injecting the manifestation of the operator error at the appropriate lower layer of the system. For example, a RAID device driver instrumented for error injection can be used to simulate an operator who removes the wrong disk after one disk fails in a RAID array.

Determining an appropriate set of human errors to inject is different for slips and lapses as compared to mistakes. In both cases one might start with a list of the goals that an operator would have in administering the system, whether for routine maintenance tasks or in diagnosing and repairing a problem. To simulate slips and lapses, one might take each of those goals and enumerate all of the "correct" ways to implement each of them. Then, one would randomly add, delete, or modify operations and map them onto the appropriate user interface operations or lower-level manifestations. To simulate mistakes, one might enumerate "incorrect" ways to implement each of the operations and then map those onto the appropriate user interface operations or lower-level manifestations.

The task of simulating slips and lapses would be well served by a mechanism that allows operator intent to be expressed declaratively and that then generates correct and incorrect implementations of those intents and triggers them at the appropriate point in the benchmark run. Note that it is more costly to simulate mistakes than slips and lapses, since for the latter one needs a model of how operators mis-plan rather than just how they mis-execute. Observations of the mistakes operators make during the competitive benchmarks can help determine what operator errors to inject in developer benchmarks.

We also note that injection of simulated human errors is an iterative process: the injected workload will necessarily change as systems improve and what used to be important causes of failure are replaced by other, previously more minor causes of failure that are not yet addressed.

Humans could be used occasionally in developer benchmarks if the cost of doing so can be sufficiently reduced, *e.g.,* by using as few human subjects as possible. This raises the question of how to make statistically meaningful conclusions about human error from small operator populations. Fortunately, user interface studies suggest that only between five and twenty subjects are truly necessary to obtain reasonably representative results [3] [13]. If a small group of operators is used repeatedly, there is the question of how to prevent those operators from learning all the tasks they will need to perform, thereby being able to anticipate the tasks and thus not acting as an operator who performed a particular task for the first time would. To address this problem, one could adapt the approach that is used when testing pilots in cockpit simulators, having the benchmark auditor select the errors and maintenance tasks that the operator will have to deal with, randomly chosen from a larger set of possible tasks. This selection of a random subset of tasks also allows the same operators who have already participated in the study to participate again, since they will likely be exposed to new tasks that they won't have learned.

3.2.2. Exploring the perturbation space

Another issue related to benchmark cost, whether for competitive or developer benchmarks, is how to select which perturbations to inject out of the space of all possible perturbations. Even without human operators it is too costly to inject all possible perturbations—whether the perturbations are due to hardware, software, or humans, they each may require a nontrivial amount of time to recover from. Therefore it is useful to select only a subset of all possible perturbations to invoke.

Our proposal is to expand the random task selection technique described at the end of the previous section, to apply to all perturbations injected in a benchmark. The benchmark designer would establish a set of equivalence classes of perturbations. and randomly select some subset of each equivalence class as the set of perturbations to invoke. For example, a few possible equivalence classes of software perturbations might be pointer errors, incorrect API usage, synchronization errors, and control flow errors. A few possible equivalence classes of hardware perturbations are CPU failures, memory failures, network interface failures, and disk failures. A few possible equivalence classes of perturbations due to human error are software configuration errors, hardware configuration errors, and errors in managing user accounts and storage.

Although the most complete benchmark would inject all possible perturbations, and the most representative benchmark would inject perturbations according to a model of observed events, the most *cost-effective* benchmark should inject the minimal set of perturbations necessary to assess the dependability behavior of the system with respect to the perturbation classes that are of interest, thereby decreasing the amount of time for which the benchmark needs to be run.

3.3. Benchmark relevance and fairness over time

An issue that arises in both competitive and developer dependability benchmarks is the need to keep the benchmark relevant over time. Four factors are at work to make a benchmark obsolete: changes in the typical workload, changes in the typical perturbation load due to changes in the system(s) being benchmarked, and attempts to game the benchmark. The question then is how to keep the benchmark relevant without incurring undue cost.

By using standard performance benchmarks as the workload, the workload of dependability benchmarks automatically tracks the typical workload for the benchmarked application. Developers might want to benchmark individual system components, but to do that they can record and replay the component API-level interactions that were caused by the standard benchmark load.

Changes in the system(s) being benchmarked invalidate a benchmark's assumptions with respect to perturbation load. These changes happen for different reasons in competitive and developer benchmarks. In competitive benchmarks, we expect the most common errors to change over time as the dependability field addresses the important system dependability challenges. Indeed, longitudinal studies of failure causes over time suggest that the frequency of various causes of failure does change with time as technology develops [12].[1] For competitive benchmarks, we can use a standard error dataset collected from real systems to determine an appropriate perturbation load.

In development benchmarks, for a single system we expect that the set of problems most deserving attention (*i.e.*, that incur the longest time to repair, that cause the most significant degradation in quality of service, or that cause data loss or corruption) will change as the system's developers improve those attributes of the system. For example, user interface improvements made to the operator's interface might drastically change the set of possible human perturbations, rendering some irrelevant or not applicable, and adding other new ones.

Finally, for competitive dependability benchmarks to be relevant over time, they must be designed so that they are difficult to game. Attempts to game the benchmark might include hardcoding error predictors specific to the benchmark in order to identify and react to the precursors of benchmark-specific errors. Likewise, operators might be artificially trained to identify error precursors and to prevent a failure that would normally result. We attempt to address these issues using stochastic selection of errors to inject from the equivalence classes described in Section 3.2.2. Over longer time-scales, gaming is addressed by changing the standard perturbation workload.

4. Conclusion

We have described a division of the space of dependability benchmarks into competitive benchmarks and developer benchmarks. Competitive benchmarks aim to maximize realism without cutting corners, but they may

[1] Collecting data about the causes of failure in real systems is time-consuming and expensive. We therefore advocate the creation of a neutral organization that would collect and aggregate failure data from deployed services to reduce the cost of obtaining this data. It would be logical for this to be done by the same organization that maintains and audits the dependability benchmark.

incur significant time and effort to run and therefore would be run infrequently. Developer benchmarks are approximate, targeted versions of the competitive benchmarks designed to be run frequently during the development process. They sacrifice some realism and comprehensiveness in return for cost savings. The two types of benchmarks differ in whether they benchmark specific techniques or systems as a whole, the design of their perturbation model, and their approach to incorporating human operators, among other factors.

The issues described in this paper lead to a number of additional research questions. One of the more interesting is how to correlate the results from developer benchmarks to those of competitive benchmarks and/or to overall system dependability. A related question is how to combine the results of developer benchmarks into a measure of overall system dependability, in the absence of a single holistic test, much as SPEC CPU synthesizes an overall performance metric from multiple benchmarks in the SPEC CPU suite. Finally, the notion of approach-specific benchmarking raises the question of whether it is feasible to compare the cost-effectiveness of different approaches to dependability. One possible approach that could apply for all dependability techniques is to measure the financial cost of implementing the various techniques and to compare those results to the prevented cost of downtime for a generic service. In closing, we note that an implementation of one or more real dependability benchmarks is necessary to truly demonstrate the usefulness of any of the ideas presented here.

Acknowledgments

The authors gratefully acknowledge the anonymous reviewers for their comments. This work was supported in part by DARPA under contract DABT63-96-C-0056, the NSF under grant CCR-0085899 and infrastructure grant EIA-9802069, and the California State MICRO Program.

References

[1] P. Broadwell, N. Sastry, and J. Traupman. FIG: A prototype Tool for Online Verification of Recovery Mechanism. *Proceedings of Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, 2002.

[2] A. Brown, L. C. Chung, and D. A. Patterson. Including the Human Factor in Dependability Benchmarks. *2002 DSN Workshop on Dependability Benchmarking*, 2002.

[3] A. Brown and D. A. Patterson. To Err is Human. Proceedings of the *First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, 2001.

[4] A. Brown and D. A. Patterson. Towards availability benchmarks: A case study of software RAID systems. *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

[5] R. Chillarege and N. Bowen. Understanding large system failure--a fault injection experiment. *FTCS-19*, 1989.

[6] M. Ivory and M. Hearst. The state of the art in automating usability evaluation. *ACM Computing Surveys*, vol. 33, no. 4, 2001.

[7] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, 1997.

[8] D. Richard Kuhn. Sources of Failure in the Public Switched Telephone Network. *IEEE Computer*, vol. 30, no. 4, 1997.

[9] H. Madeira and P. Koopman. Dependability benchmarking: making choices in an n-dimensional problem space. *Proceedings of the first Workshop on Evaluating and Architecting System Dependability*, 2001.

[10] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. Proceedings of *USENIX 1996 Annual Technical Conference*, 1996.

[11] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, vol. 33, no. 12, 1990.

[12] B. Murphy and T. Gant. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International.* vol 11, 341-353, 1995.

[13] J. Nielsen and T. K. Landauer. A mathematical model of the finding of usability problems. *Proceedings of ACM INTERCHI '93*, 1993.

[14] D. Oppenheimer. Studying and using failure data from large-scale Internet services. *10th ACM SIGOPS European Workshop*, 2002

[15] J. Reason. *Human Error.* Cambridge University Press, 1990.

[16] Standard Performance Evaluation Corporation. http://www.spec.org/

[17] Transaction processing performance council. http://www.tpc.org/

[18] T. Tsai, R. Iyer, and D. Jewett. An approach towards benchmarking of fault-tolerant commercial systems. *FTCS-26*, 1996.

[19] M. Vieira and H. Madeira. Recovery and performance balance of a COTS DBMS in the presence of operator faults. *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.

[20] J. Xu, Z. Kalbarczyk, and R. Iyer. Networked Windows NT system field failure data analysis. *Proceedings of the 1999 Pacific Rim Dependability Conference*, 1999.

[21] J. Zhu, J. Mauro, and I. Pramanick. System recovery benchmarking. *2002 DSN Workshop on Dependability Benchmarking*, 2002.