Performance Evaluation in Operating Systems Research: Approaches and Challenges

Matthew Brehmer
Imager Laboratory for Graphics, Visualization, and HCT
Dept. of Computer Science, University of British Columbia
brehmer@cs.ubc.ca

ABSTRACT
A survey of four research papers pertaining to performance evaluation of operating system is presented. This survey and its related discussion highlight the approaches to system evaluation along with its associated challenges and tradeoffs. The article speaks to methodological issues of realism, accuracy, the granularity of measurement, the portability of measurement techniques and tools, as well as the comparability and reproducibility of methods and results.

Author Keywords
Performance evaluation; methodology; benchmarking; trace-based evaluation; reproducibility; survey.

ACM Classification Keywords
D.4.8. Operating Systems: Performance

General Terms
Experimentation; Measurement; Performance; Standardization.

1. INTRODUCTION

It has been observed that in operating systems research, most of the intellectual effort goes into the implementation of novel systems [10]. This is not surprising, as researchers must understand the complex multifaceted relationships and design tradeoffs between speed, security, extensibility, scalability, portability, and complexity. However, it has been argued that more effort is needed when it comes to evaluating the efficacy of these novel systems [12, 13, 15], and that performance evaluation protocols and benchmark workloads often lead researchers to "measure what is easily measured" [10]. A fair and valid evaluation of system performance requires its own set of tradeoffs and challenges: representativeness of real-world behaviour, accuracy and the granularity of measurement, the portability of methods and measurement tools, the comparability of results between systems, as well as the reproducibility of methods and results. Furthermore, the protocols and results of operating system performance evaluations are seldom reported in a consistent or standardized fashion, relative to other areas of computer science [10, 13], particularly in research pertaining to file systems [15].

Computer science is still a nascent discipline, and there are signs of emerging standardization with regards to how computer hardware and software are evaluated. As the execution and reporting of evaluation experiments becomes increasingly transparent and reproducible, and as methodologies become increasingly transferable, it is believed that this will lead to higher rates of technology transfer from research to industry, to the successful deployment and adoption of novel operating systems [5, 10].

This paper describes the approaches and challenges associated with the empirical evaluation of operating systems and their components. It was motivated by my own research interest in experimental methodologies, owing to my background in human factors and the cognitive sciences; while I am familiar with methods for studying human behaviour and human-computer interaction, I was unfamiliar with how system behaviour is studied prior to performing this survey. My questions related to how system performance is measured, which metrics matter, and how procedures and results ought to be reported.

To facilitate a review and comparison of research papers pertaining to operating systems performance evaluation, I began my survey with several relevant position and workshop papers [2, 8, 10, 12]. My intent was to complement my own opinions on the need for evaluation, on experimental design and reporting, as well as on issues of experimental reproducibility and validity. These position papers also directed my literature search, leading me to three of the articles [3, 4, 15] summarized in the survey presented in Section 3.

The structure of this paper is as follows: in Section 2, I describe the several dominant approaches to performance evaluation in systems research, indicating when said approaches are appropriate, highlighting their advantages and disadvantages. In Section 3, I provide a survey of four research papers [3, 4, 5, 15]. These articles

Submitted as a final course project for CPSC 508: Operating Systems, Dept. of Computer Science, University of British Columbia, December 2012.

speak to several methodological issues relating to performance evaluation in systems research, namely the challenges and tradeoffs mentioned above. Following this, I discuss common themes and draw comparisons between the articles in Section 4. Finally, in Section 5, I suggest areas for future work along with my conclusions.

2. PERFORMANCE EVALUATION: APPROACHES

Mogul [10] reviewed the various approaches to measuring and reporting operating system performance, and observed a tension between realism and reproducibility.

2.1 Benchmark Performance

A common approach involves the use of *benchmark* protocols or workloads, often these are the product of prior academic research, or they are made available by technology vendors. These benchmarks are often easy to use and provide simple numeric results. Liedtke et al. [8] elaborates on the uses of benchmarks: while benchmarks serve a useful commercial purpose in communicating system capabilities to customers, benchmarks are used in research to understand a complex system, to characterize or predict the effects of a modification, to help identify performance bottlenecks, to invalidate a theory, or to formulate a hypothesis.

Preexisting benchmarks are often inflexible in that they tend to lack configurable workload sizes or runtimes [10]; in addition, Liedkte et al. describes several other problems associated with the use of benchmarks: One of which is *non-transitivity*, in that combining multiple optimizations proven separately by benchmarks may not result in a positive increase in overall system performance, due to unforeseen interactions between the optimizations. Another problem is *instability*, where low-level optimizations have a positive effect on benchmark performance but not on application performance, even when benchmarks are designed with target applications in mind. Finally, there is the problem of *non-linearity*, where different system behaviour and performance may result from alternative concurrent and sequential benchmark runs.

Both Mogul [10] and Liedtke et al. [8] discern between *microbenchmarks* and *macrobenchmarks*. Microbenchmarks are useful for measuring the performance of a system primitive in isolation, but a microbenchmark in itself cannot by itself be used to predict the performance of the whole system. Traeger et al. [15] suggest that microbenchmarks and macrobenchmarks be used in conjunction as a means of evaluating a system in its entirety. Seltzer et al. [12] argue that microbenchmarks which do not consider application usage patterns and representative loads on these primitives are of little use to fellow researchers or potential adopters of the system.

When microbenchmarks are used to evaluate low-level system primitives, Bershad et al. [2] highlight some of the additional assumptions that often contribute to misleading results of microbenchmark experiments. Microbenchmarks often do not account for cache and buffer effects incurred by real applications: performance can vary from run to run of an application, or between different versions of the same program when cache effects occur. As a result, microbenchmarks tend to overestimate the use of some system primitives, using them in ways that wouldn't be reflected in practice. To make microbenchmark results more reproducible, Bershad et al. suggest that the cache be explicitly flushed between trial runs, thereby revealing worst-case, albeit reproducible, performance.

Macrobenchmarks are intended to measure and predict the higher-level end-to-end performance, simulating application workloads by performing a mixture of operations, while maintaining the analyzability that a benchmarking tool affords. Since macrobenchmark measurements are distinct from directly measuring the performance of any particular applications, the representativeness of a macrobenchmark workload is debatable.

2.2 Application Performance

Another common approach to performance evaluation is what Mogul [10] refers to as "running a bunch of programs". While a seemingly ad hoc selection of applications raises doubt about whether they are indicative and representative of general system performance, this approach does provide a realistic indicator of performance if the system under measurement was specifically designed to support these particular applications. On the other hand, application-specific benchmarks are often not reproducible, and it is difficult to make meaningful comparisons across systems or over periods of time.

Mogul [10] calls upon researchers to design benchmarks that are more realistic and representative of real-world applications, while maintaining the ability to measure and predict absolute performance in production environments. This sentiment is shared by Seltzer et al. [12], who proposed several methodologies for application-specific benchmarking. One of which is a *vector-based methodology*, in which a characterization of a system in terms of its underlying system primitives is represented as a vector. Meanwhile, a second application vector describes the demand the application places on each primitive. The combined dot product of these vectors is reported as a measure of system performance. The advantage of this methodology is that it generalizes whenever applications are implemented using a common well-defined operating system API. A disadvantage pertains to the granularity of the results; relative application performance is often correctly measured, but absolute performance measures across system and application configurations is more difficult to attain.

2.3 Trace-Based Approaches

Tracing system behaviour is an approach to system evaluation that involves recording and replaying recorded real-world workloads. Tracing can provide highly realistic performance data, but since changing the system implementation often changes user behaviour, Liedtke

et al. [8] warned that the recorded trace may no longer be representative, and any evaluation of the changed system that uses the trace may lack external validity. When replaying a trace, it is not clear when to trigger recorded actions in response to system behaviour, because one cannot account for how a user's behaviour changes in response to novel system behaviour. In addition, studying user interaction is a problem for reproducibility, even when no system changes have been made, since user behaviour is inherently irreproducible and unpredictable.

Nevertheless, replaying traces on modified systems can still be informative. If one is confident in the representativeness of a trace despite a change in system behaviour, confident that any effects on user behaviour will be minimal, the trace could simulate realistic load for comparing different policy implementations, such as those for process scheduling or page allocation. Mogul [10] observed that trace-based evaluations are also prevalent in mobile and embedded systems research, where controlled laboratory settings are not representative of the context in which these systems will be deployed.

Selzer et al. [12] described a *trace-based methodology* for constructing application-specific benchmarks, in which usage loads are modelled after particular streams of application use. These trace-based benchmarks make it is easy to construct what-if scenarios and mimic anticipated application workloads. Additionally, these trace-based benchmarks can be disseminated and shared more easily than application logs, which are often proprietary or known to contain private information. However, these traced-based benchmarks are specific to an application-system combination, and do not give insight into the use of underlying system primitives seen in the aforementioned vector-based methodology.

When designing a trace-based evaluation, one must be concerned with the granularity of the trace [8]; traced events should be large enough to label with semantic meaning, without being intrusive enough to have the user notice the overheard incurred by the trace. Related to the issue of trace granularity is whether there is enough information contained in a trace to identify individual users, based on recorded use of the system. If traces are to be shared and disseminated for research purposes, care should be taken to ensure that traces are anonymized.

2.4 Hybrid Approaches

Incorporating multiple approaches is an effective way to characterize and evaluate an entire system [3, 15]. However, it can be difficult to relate results gathered from disjoint experiments performed at different levels of granularity; as such, structured hybrid methodologies may be more useful.

Selzer et al.'s [12] *hybrid methodology* combines trace- and vector-based methodologies for application-specific benchmarking, using a simulator to identify system primitive operations and results used in the trace (e.g. cache hits and misses), thereby converting an application trace into an application vector.

Liedtke et al. [8] proposed a similar idea for constructing a representative workload from a trace: *stochastic benchmarks* randomly sample recorded user activity at a suitable level of granularity. A drawback to the stochastic approach is the incomplete knowledge of initial and global system state, as initial system state when replaying a trace cannot be assumed to be identical to the state when the trace was recorded. As a result, replaying full or stochastically-generated user traces restricts the types of possible experiments one could perform.

2.5 Approaches and Challenges

In reading the aforementioned workshop and position papers [2, 8, 10, 12], I came to understand which evaluation approaches were common, as well as how evaluation was discussed and prioritized within the operating systems research community. In addition, I was made aware of the challenges faced by practitioners in this domain.

The use of previously published benchmarking protocols and benchmark workloads is common in systems research, and it is often questionable as to the extent to which these transfer from one experimental context to another. It is similarly difficult to assess when a heavily-used benchmarking protocol or workload becomes obsolete. Portability is another major concern, especially when comparing performance between architectures or operating systems. An example is Chen et al.'s comparison of three commodity operating systems [4], discussed in Section 3.2. A unique problem affecting reproducibility in systems research is the use of proprietary or commercial benchmarking tools, which tend to be less accessible or freely available [15]. A related issue is the inability to publish performance results of logged behaviour of commodity software or systems, as stipulated by some end-user license agreements [1, 12], which makes reporting comparisons between research system and commodity systems difficult.

In considering these challenges, I gained another lens with which to examine the surveyed research papers in Section 3, culminating in the comparative discussion of Section 4.

3. PERFORMANCE EVALUATION SURVEY

In the preceding section I reviewed the major approaches to performance evaluation prevalent in operating systems research, summarizing their advantages and disadvantages with resect to experimental design and the reporting of results, as well as issues of realism, portability, reproducibility. With these issues in mind, I have examined four research papers [3, 4, 5, 15] which involve these approaches to performance evaluation.

3.1 From a Set of Tools to a Methodology

Those intent on studying and comparing the performance of low-level system primitives must consider the

granularity, portability, and statistical rigour of an evaluation protocol and the measurement tools involved. Microbenchmarking tools and protocols, as described above, affords such low-level analysis differs, differing from macrobenchmarking or kernel profiling approaches; the former doesn't isolate the many variables of interest required to construct a decomposable hierarchy, while the latter approach is often infeasible due to unavailable or proprietary kernel source code. Brown and Seltzer's *hbench:OS* [3] is a benchmarking methodology and test suite that revises McVoy and Staelin's *lmbench* suite [9], allowing researchers to study the hardware-architectural basis of operating system performance. In this section, I concentrate on Brown and Seltzer's revision [3], as well as their results and observations. However, I first provide a short summary of the original *lmbench* suite.

3.1.1 lmbench

McVoy and Staelin's *lmbench* [9] is a microbenchmark suite for evaluating the basic low-level system primitives, examining the transfer of data between the processor, cache, network, and disk. The suite is intended to be widely available and highly portable across architectures and operating systems, including both uniprocessor and multiprocessor systems; although in 1996 it did not incorporate any multiprocessor-specific tests. Benchmark suites prior to McVoy and Staelin's were described as being either too focused on a single low-level feature, not portable, not widely available under public license, had poor timing granularity, or either provided too many or too little tests, complicating subsequent analysis. The suite's tests, all run in main memory, recorded measurements of latency and bandwidth:

- The *bandwidth benchmarks* measure the rate at which a particular facility can move data, including memory read, write, and copy, as well as IPC and cached I/O.
- The *latency benchmarks*, measured at a clock tick granularity, include several measurements of memory, system calls, signal handling/CPU interrupt, process creation, IPC, file system operations, and disk operations.

The *lmbench* benchmark suite addresses cache and memory size issues by increasing the size of data used by factors of two, thereby measuring cache and memory performance separately. To account for variability, benchmarks are run multiple times and only the minimum result is recorded. To account for uncertainty with regards to whether the data is in the cache, the benchmark is run several times and only the last result is recorded.

3.1.2 hbench:OS

The contribution of Brown and Seltzer's [3], beyond the *hbench:OS* suite, is a portable benchmarking methodology, whereas they argue that *lmbench* is simply a set of tools without operational guidance. *hbench:OS* is an improvement upon *lmbench* in that the tests are more rigorous, self-consistent, reproducible, and conductive to statistical analysis, thereby being easier to analyse. It should be noted that in later versions of *lmbench* [14], many of Brown and Seltzer's revisions were integrated into the suite, improving portability and extensibility.

Modifications to *lmbench* included updating the timer resolution with hardware cycle counters, removing overheads introduced by the timing mechanism. They also divorced data analysis from the data measurement components of the benchmark. Rather than taking a minimum measurement, *hbench:OS* takes a n% trimmed mean, discarded both the worst and overly optimistic values. This is useful when results are not normally distributed, such as a bimodal distribution. Cache priming is also done by running one iteration of the test before collecting data. The *hbench:OS* tests are also more parameterizable than *lmbench*'s tests, allowing for distinctions between dynamically and statically linked processes. Modifications to the memory read, write, and copy bandwidth tests to allow for measurement of the L1 and L2 caches separately. Context switching latency was not highly portable in the original *lmbench* suite, due to its inability to detect cache conflicts, a common microbenchmark problem also observed by Bershad et al. [2]. As a result, *lmbench* sometimes reported negative or zero-sized context switch latencies. *hbench:OS* measures only the true context switch time, approximated from cache and memory read bandwidths. They retain the original *lmbench* context switch test, noting that the new *hbench:OS* test has standard deviations that are lower ($\sim$3% vs. 10%<). Finally, memory bandwidth tests were modified such that direct and indirect memory referencing could be more easily compared.

Brown and Seltzer's *hbench:OS* methodology [3] aims to characterize the whole system using a bottom-up approach, from subsystem bandwidth performance to the operating system and application level, where one must consider the latency of system calls, process creation, as well as file and network access. Thus the performance can be decomposed while varying features of the hardware, and features of one layer can be related to those at a layer above or below it: from hardware capabilities to low-level kernel primitives to high-level operating system services and finally to application performance. When described in this way, it is possible to see how *hbench:OS* methodology combines microbenchmarking and microbenchmarking, such that results of one can be directly attributed to results of the other. Naturally, a total reconstruction is not always possible, and middle levels of the performance hierarchy may be inaccessible to measurement or adjustment; these can be bypassed, leaving only operating system-dependent application performance at the top, and hardware capabilities at the bottom of the hierarchy. In these cases, if the hardware can be varied in a controlled manner, thus still providing some useful information relating the top and bottom of the hierarchy.

The authors report a case study of *hbench:OS* on eight machine configurations running the NetBSD 1.1 operating system over the Intel x86 architecture. They mea-

sured bulk data transfer bandwidth, a bottom up analysis from the hardware level to the kernel and application levels, as well as process creation latency, a top-down analysis involving both static and dynamically linked processes. The bulk data transfer scenario is representative of bandwidth-sensitive applications such as web servers and multimedia applications, incorporating reading files, sending and receiving data via TCP, mapping files into an address space, where memory accesses were decomposed into memory reads, writes, and copies. With these decomposed results, the authors were able to predict performance at the operating system and application level. Overall, their case study found that despite CPU optimizations, hardware-level features of the memory system dominated operating system and application-level performance. Where measurements differed from those predicted from top-down analysis, it became possible to account for these differences, by isolating and examining optimizations or flaws in the hardware, in one case discovering that some kernel-level primitives did no depends on memory hardware performance.

What is important to retain from this article is that prior system evaluation benchmark tools, such as the original *lmbench*, forced a choice between levels of analysis, without providing a methodology or the guidance required to resolve results collected from hardware profiling, system primitive microbenchmarking, and application-level macrobenchmarking. *hbench:OS* introduced an evaluation alternative that was methodology-driven, rather than tool-driven, facilitating reporting and analysis of results while retaining portability and flexibility.

3.2 The Challenge of Comparative Evaluation

Chen et al. [4] compared the performance of three commodity operating systems running over a Pentium architecture. This article highlights the challenges of evaluation granularity, the identification of comparable cross-platform metrics, and the portability of evaluation methodologies and benchmark workloads. The authors measured system performance at the low level of system primitive operations using microbenchmark protocols as well as at the level of individual applications with representative workloads. Windows (for Workgroups), Windows NT, and NetBSD Unix are three commodity operating systems that are comparable because they are widely available and support the same typical patterns of use. However, the systems differ substantially at the kernel implementation level, with Windows lacking protected address spaces, preemptive multitasking, as well as high-level system abstractions such as pipes and background jobs. Due to these differences in system configuration, a fair low-level evaluation was difficult to design, as many preexisting microbenchmark suites cannot produce results that are comparable given these differences. In addition, the authors express their frustration with previously published benchmarks protocols and workloads, predating other who have made similar admissions [10, 12, 15]. Chen et al. argue that these benchmarks, originally intended for descriptive commercial purposes, are misleading or incomplete, and cannot often be used for accurately answering comparative or predictive research questions.

Their methodology involved first gathering, sequencing, and running a set of microbenchmarks. Then, descriptive performance results were collected for a small number of cross-platform applications, which were selected based on the expected load they placed on the system primitives assessed using the microbenchmarks. The application performance was subsequently interpreted, again with regards to the microbenchmark results.

Their microbenchmark testing suite was largely based on the *lmbench* suite [9], however this predated the *bench:OS* revision contributed by Brown and Seltzer [3]; Chen et al. 's study was likely a motivating factor behind the development of *bench:OS*. Chen et al. contribute a few new measurements to the *lmbench* suite (noted in the following list); these were particularly useful for informing the design of the application-level benchmarks. Their microbenchmark suite contained the following tests:

- A null test: the baseline overhead latency to access the hardware counter itself (not in the original *lmbench* suite).
- The latency of a system call, an indication of the cost of accessing functionality implemented in the system.
- Running a trivial program, averaged over 50 invocations.
- A test of memory access time, referencing a large array with a stride of 128 bytes.
- A mix of file system operations, reflecting actual usage: accessing files that hit in the disk cache, accessing small files and go to disk, and file creation.
- A graphics subsystem benchmark (not in the original *lmbench* suite), which allowed them to inform and understand the *ghostscript* application workload (see below).
- A network throughput test, informing the web server application workload (see below).

The dependent performance measures for these microbenchmark tests were collected with calls to event and cycle hardware counters, accessible via device driver-level kernel extensions. The low-level events that could be observed and measured included data reads and writes, instruction reference misses in the cache and TLB, data reference misses in the cache and TLB, segment register loads, instructions executed, cycle counts, and hardware interrupts.

The differences between the three operating systems limited what could be compared at the microbenchmark level of system behaviour. In particular, Chen et al. were not able to discern and compare events logged at different protection levels, so they were unable to attribute performance results to user- or kernel-level events. The systems also differed in terms of how idle time was measured; as a result some time-based metrics in the *lmbench* microbenchmark suite could not be used. Instruction

formats were hard to compare across the three operating systems; while instruction counts can be useful for comparing total work on RISC processors, the Pentium architecture on which they performed their experiments allowed for multi-cycle instructions, which were used differently in the implementation of Windows, Windows NT, and NetBSD. Since instruction counts were not always interpretable, they opted to instead to use cycle counts for comparing the total latency of computations. A drawback to this compromise was that they couldn't easily assign cycles to specific events. Finally, the different cache and TLB policies of the three operating systems made it difficult to absolutely compare the read, write, and miss metrics, however relative performance could still be discerned from the results.

At the level of application workload performance, applications were chosen based upon their compatibility with all three operating systems, and the belief that their performance could be explained by the microbenchmark results. Three applications were selected:

- `wish`: a *tcl-tk* command interpreter; a CPU intensive application with heavy use of the windowing system, requiring many context switches between application and the graphics server.
- `ghostscript`: a postscript viewer application, which also placed heavy demand on the windowing system; *ghostscript* is a a dynamically linked application, and was informed by graphics subsystem microbenchmark.
- A web server application that placed heavy demand on the network and file systems.

Application behaviour was somewhat predictable, in that each had a working set under 32MB and caused no significant paging during execution. During the application workload tests, system background activity was limited, the network subsystem was disabled except in the tests that required network access, and used single-user mode was enabled. Controlling these factors allowed for more precise measurements, at the cost of realism. Arguably the authors could have conducted a larger factorial experiment, thereby isolating the cost of these factors and measuring realistic performance; the current results could have served as a baseline control condition. On the other hand, this choice would have complicated the design and reporting of the experiment. Finding applications that were compatible with all three platforms also proved to be a struggle, as the authors were limited to open source applications. This constraint meant that the results may not generalize to a wide range of applications, and are likely biased toward the NetBSD system, since the applications used in the evaluation were originally developed for UNIX systems.

Despite the challenges Chen et al. faced in designing these experiments, as well as in recording and interpreting results, their aggregate results did convey apparent relative performance differences between the three operating systems. They found that Windows for Workgroups, which does not have protected address spaces, performed worse than expected. As a result, the authors questioned the unified address space model which was showing promise at the time, such as in the exokernel project [6]. In addition, they observed Windows' use of "hooks", a mechanism that intercepts system calls, intended for application flexibility and backwards compatibility. These hooks further contributed to low-level performance differences between the operating systems, making comparisons more difficult.

This article demonstrates several of the challenges and tradeoffs prevalent in operating system evaluation. It raises questions about the portability of widely available benchmark protocols and workloads such as *lmbench*, and whether there exists a limitation to the type of comparative evaluation experiments one could perform. Due to the differences in operating system primitives and the availability of cross-platform applications, it's not surprising that making realistic and accurate comparisons proved to be a difficult endeavour.

3.3 The Challenge of Reproducibility

Mogul [10] and Clark et al. [5] both advocate that more research effort on realistic and reproducible performance evaluation will result in the increased deployment and adoption of novel systems, believing that "the ability to accurately predict performance [will] translate directly into higher profits" [10]. There are several ways to make a system performance experiment more reproducible. The system's source code should be well documented and made available to other researchers; benchmark protocols and workloads should also be open, along with their testing scripts and parameter settings.

An example of repeated systems research was a project performed by Clark et al. [5], who in 2004 successfully reproduced the performance results of Barham et al.'s SOSP 2003 paper *Xen and the art of virtualization* [1], using nearly identical hardware. In addition, Clark et al. asked if the performance claims made in the *Xen* paper regarding scalability could be replicated using cheaper commodity hardware. *Xen* was designed to provide isolation and scalable performance for up to a target of 100 guest operating systems. Furthermore, these guest operating systems would be largely unmodified commodity operating systems running potentially thousands of industry standard applications. Clark et al. also inquired about the portability of *Xen*, whether it could be run on cheaper commodity hardware, less than $\$2,500$, comparing how multiple guest operating systems running as guests over *Xen* on a commodity PC compares to Linux virtualized an IBM *zseries* mainframe (valued near $\$200,000$ in 2004). This question meant extending the original *Xen* research protocol, adding an additional comparison across different hardware installations.

Reproducing the prior results from [1] required assembling and running all the benchmarks used in the original *Xen* paper, which included the *lmbench* microbenchmark suite [9], writing the necessary scripts and setting

parameters for these benchmarks. While much of the information required to reproduce the original results could be gleaned from the published *Xen* article, some parameters and finer points of the evaluation protocol were not fully specified; Clark et al. were fortunate in that the *Xen* authors were willing to divulge this information in private correspondence. However, not all of the benchmarks could be acquired; in the case of the proprietary *SPECweb99* web benchmark [1], an analogous benchmark was built to simulate it using a trace measurement tool. In reproducing the original scalability evaluation ([1] §4.5), Clark et al. discovered that Barham et al. only allocated 15MB for each guest, which Clark et al. described as being not realistic nor sufficient for an industry standard web server. A 128MB memory size per guest would be more typical, however this would require over 12GB of memory. Clark et al. stated that an upper bound of 16 guests is more realistic for a system with 4GB of memory. As a result, Clark et al. allocated 98MB per guest operating systems, whereupon they observed that *Xen* was able to successfully scale to 16 guest operating systems.

Clark et al. also observed that the original *Xen* evaluation compared against Linux with SMP (Symmetric Multiprocessor System) support disabled in some but not all experimental conditions. This choice in parameter settings may have accounted for some of Linux's relative performance in the results presented in the original *Xen* paper [1]. In all conditions, Clark et al. compared Xen against Linux both with and without SMP support.

With regards to Clark et al.'s question of whether *Xen* could be used on an older commodity PC, they found that *Xen* could be successfully installed on such a system, albeit with a smaller number of guest operating systems; the performance of *Xen* on a commodity PC is similar to that of the Linux virtualization on an IBM *zseries* mainframe, which is remarkable considering the difference in cost is nearly two orders of magnitude.

Clark et al.'s article is a successful instance of repeated research: not only were previous performance results repeated, but they also pointed out inconsistencies and details absent in the original *Xen* article [1], in the scalability evaluation in particular. In addition, they extended the original research, posing and answering a new question about the portability and cost of a *Xen* installation. They conclude by arguing that researchers should strive to make their research more reproducible, as repeated research provides confidence to novel systems, encouraging technology transfer and industry adoption.

3.4 Guidelines, Pitfalls, and Claims Debunked

In the previous sections I have summarized the challenges associated with evaluating the performance of operating systems and subsystems, serving to illustrate the need for more guidance in this regard. Traeger et al. [15] presented such guidance, along with a survey of file system evaluation as reported nine years' worth of research papers from high-impact operating systems venues; in total, 106 SOSP, OSDI, and USENIX papers published between 1999 and 2007 were included in the survey. Specifically, they examined and compared performance evaluation methods and methodologies and the reasoning behind the use of benchmarks. This article contributes more than a descriptive survey; it is also prescriptive in that indicates shortcomings and strengths, and it provides guidance for future research regarding how to create and use benchmarks effectively, as well as how to present results.

Throughout the article, Traeger et al. also report on their own experiments, either reproducing the results or debunking the claims of the surveyed papers. In some cases, these experiments serve the purpose of illustrating methodological assumptions or insufficient reporting of experimental protocol in previous research. Their own experiments followed a benchmarking methodology in which each test was run at least 10 times; 95% confidence intervals were computed for mean elapsed, system, and user times using the student-t distribution. They disabled unrelated system services and rebooted the machine between successive sequences of benchmark runs, ensuring consistent cache states. They automated benchmark runs using the Autopilot benchmarking suite, a prior project of one of the authors. Throughout their survey, they demonstrate that different benchmarking decisions can hide overheads and latencies, particularly with regards to compile time macrobenchmarks.

While focused on file system evaluation, many of the observations and guidance presented in Traeger et al.'s article may be generally applicable to other areas of systems research; the authors position this paper in a broader context of related systems work, some of which discussed previously in this report [10, 12]. Mogul [10] also surveyed system performance evaluation as reported in SOSP and OSDI papers prior to 1999. He observed a trend in which papers in a particular topic area, such as file systems, do not often share evaluation protocol and benchmark choices, nor is there an agreed-upon approach for reporting results. He also surveyed a similar-sized body of research from recent computer architecture publishing venues, observing that performance evaluation design, analysis, and reporting tended to be more consistent within that community. A more exhaustive survey of this nature can be found in an earlier technical report from Seltzer's research group [13], which span topic areas and focuses on the lack of statistical rigour in the analysis and presentation of system performance results, motivating the work on application-specific benchmarking discussed earlier [12]. In addition to the problems inherent to all systems-related evaluation, Traeger et al. mention that evaluating file and storage systems often requires extra care, in that these systems may interact in complex ways with other subsystems, and may differ from other subsystems in terms of their underlying media, their storage environment, and their expected

[1] `www.spec.org/web99`

workloads. I will not reproduce the descriptive elements of Traeger et al.'s survey; instead, my summary will focus on the empirical observations and prescriptive guidance contributed by the authors.

Above all, when evaluating a file or storage system, Traeger et al. insist upon *reporting what was done in as much detail as possible*, and *explaining why it was done that way*. Throughout their survey, they observed that many research papers seldom include both of these components, the former necessary for reproducibility and the latter necessary for understanding the intended contribution of the system or systems under study.

What was done?: detailed reporting of the experimental context and the state of the system should be reported: is the cache warm or cold? For disk storage benchmarks, where are the partitions located? Is the file system empty, or has the system been aged or subject to real-world use, and if so, for how long? Are other nonessential services running during the evaluation; what interactions occur? Are workloads multithreaded?

With regards to performance data collection and analysis, Traeger et al. compiled operational guidance for executing an evaluation protocol, reporting said protocol, and reporting results. Care should be taken such that each test run is identical, standard deviations and confidence intervals should be computed in the same way. Automated scripts may perform these tasks, thereby limiting human error. When reporting a protocol, one should include the number of benchmark runs, the benchmark runtimes, the number of benchmarks, and a description of the system state, including the state of the cache.

When reporting experimental results, confidence intervals are recommended over standard deviation, as the former produces a better sense of the true mean, generally decreasing as more test runs are performed; the standard deviation, on the other hand, captures the variation between successive runs, and may not decrease over time. When analyzing the results and computing statistical measures, a normal distribution can only be assumed for more than 30 runs; less than 30 is considered to be a small sample size, where a student-t distribution is more appropriate. Anomalous results, large confidence intervals or non-normal distributions should not be discarded; a software bug or erroneous benchmarking script may be the cause.

Why was it done that way?: the purpose of an evaluation should also be clear. One's intent may be to compare against other similar systems, to examine the performance when subjected to an expected workload, or to identify the causes of performance overheads or improvements. Of these, the first is most meaningful to readers, so it is often worthwhile to include a comparison to an alternative system in an evaluation whenever possible. The latter often requires testing several configurations in turn. It is also encouraged to evaluate both high-level and low-level performance, usually satisfied by either macrobenchmarks or a trace-based evaluation (high level) and a set of microbenchmarks (low level). It is often important to question the realism, granularity, accuracy, and scalability of the both macrobenchmarks and traces. Results provided by microbenchmarks are more meaningful when used to explain low-level performance differences, highlighting worse-case behaviour, isolating specific effects or interactions.

Next, Traeger et al. discussed the types of evaluations as reported in the papers in their survey. Here they discuss trends and common approaches for using and reporting on macrobenchmarks, trace-based evaluations, and microbenchmarks, defined earlier in Section 2.

3.4.1 Macrobenchmarks

Macrobenchmarks aim to simulate a real-world application workload, however many papers fail to describe the reasoning for opting to use a macrobenchmark. An example is the *Postmark* macrobenchmark [7], which uses a synthetic workload, but doesn't perform any actual application processing itself. The workload size is not configurable, so it doesn't scale to modern systems (as of 2008). Another inefficiency is *Postmark*'s file selection algorithm, which is $O(N)$ on the number of files. Traeger et al. suggest that a configurable and accurately measurable run time would be more scalable than a configurable workload size, thereby affording better longitudinal comparisons as hardware improves.

Compile benchmarks are another flavour of macrobenchmark, however these vary considerably across architectures and compiler chains. The authors empirically debunk the assumption that file systems see similar workloads with compile benchmarks, independent of the software being compiled; in reality these workloads vary from run to run, even on the same machine.

Traeger et al. also point out that many popular research and commercial macrobenchmarks do not provide configurable operation mixes, and many suffer from from being outdated, being not reflective of modern application behaviour and their use of the cache. Commercial macrobenchmarks from the TPC, SPC, and SPEC organizations are not widely or freely available. The authors also observe that these benchmarks are often not run according to their specifications.

3.4.2 Trace-based Evaluation

Replayable traces are also used in file system evaluation. While the recorded trace is a real-world workload, questions of generalizability must still be asked. Some papers recorded traces of macrobenchmarks, which is as questionable as the macrobenchmarks themselves, potentially negating the realism of the trace methodology; although, as some macrobenchmarks are proprietary or expensive, traces offer a compromise by providing a benchmark-generated workload. Traeger et al. observed, as Liedtke et al. [8] had a decade earlier, that there is no clearly-defined or agreed upon way to record and replay traces, nor are trace recording tools made

available or reported in sufficient detail. Traces can be recorded at several levels, from the level of system calls to that of network or driver protocols, requiring a trade-off decision between trace granularity and portability.

Trace replaying is likely to occur at the level it was recorded; this may involve aging a file system before replaying, and this process is seldom explained. Replay speed should also be justified: often, a trace is run either at the speed it was recorded, or it might be run as fast as possible.

Finally, traces should be made available for promoting reproducible research, however precautions must be taken to ensure the anonymity of trace data.

3.4.3 Microbenchmarks

Microbenchmarks test a small number of low-level operations, highlighting performance overheads or benefits implied by macrobenchmark or trace results, or to isolate a specific aspect of the system. Popular microbenchmarks share many of the same considerations described above for macrobenchmarks: scalability, cache effects, generalizability, portability, configurability, and accuracy. Some microbenchmarks are trivially simple to reproduce, even if not publicly or freely available, slight variations in microbenchmark implementations can lead to significant differences in results; Traeger et al. empirically show how five subtly different implementations of the *Sprite LFS small-file benchmark* [11] produce significantly different results.

Ad hoc single-use microbenchmarks are the most difficult to reproduce and are most prone to bugs, however these may still be useful in conjunction with other microbenchmarks, such as for explaining some unexpected or anomalous result. Ad hoc single-use microbenchmarks may also be useful in the initial phases of benchmarking, to explore the behaviour of the system, to guide the selection of more widely accepted and available benchmarks.

Another form of microbenchmarking is the use of standard system utilities as a means of creating a representative workload, such as `wc`, `grep`, `cp`, `diff`, `tar`, and `gzip`. These utilities are widely available and understood; however, versions of these utilities are subject to change, and they may not scale, given different input files.

3.4.4 Unmet Needs

Traeger et al.'s findings highlight a need for empirical methods that allow for absolute and relative comparisons of multiple workloads. Methods are also required for normalizing performance results for the hardware and operating system on which they were collected, thereby facilitating cross-system comparisons.

4. DISCUSSION

Given the common interrelated challenges associated with the various approaches to system evaluation described in Section 2, it is possible to draw several comparisons between the four surveyed papers [3, 4, 5, 15].

Realism/Representativeness: in retrospect, Selzter et al. [12] have admitted that even a system benchmark such as *hbench:OS* [3] may not provide realistic indicators of system performance specific to particular applications. While Traeger et al.'s [15] article pertained to file system research, the challenges discussed were shared by the other papers, which dealt with other topic areas. As such, it is possible that some of the methodological guidance presented in that article is transferable. According to Traeger et al., triangulating on a representative indication of system performance may be possible by combining a hierarchical system benchmark such as *hbench:OS* [3] with a trace-based approach or a *stochastic* benchmarking approach [8], recording and replaying a trace of operations. Another option is combining benchmarking with application workload testing, as in the case of Chen et al. [4], however in that study, realism was hampered due to the choice of UNIX-centric applications. If one is interested in the performance of a particular set of applications, the *hybrid-based approach* [12] described in Section 2.4, where an application trace is converted to an application vector, subsequently combined with a system primitive vector, resulting in an indication of overall performance.

Despite a mixture of evaluation methods, unrealistic experimental protocols, workloads, and parameters may not be initially apparent when reading a research paper, and these concerns may only become apparent upon reproducing the work. Such was the case with Clark et al. [5]'s reproduction of the earlier Xen paper [1], who found that Barham et al.'s scalability experiment was not representative of typical web server configurations.

Granularity of Measurement: the *lmbench* [9], used by Chen et al. [4] in their evaluation of three commodity operating systems, was developed in response to the imprecise granularity of prior benchmarks, whose measurements did not capture system primitive performance and yet neither were they representative of application-level performance. *hbench:OS* [3] took this a step further, providing means to measure system performance from the hardware level to the application level, with fine control over the granularity of timing measurement. Combining evaluation approaches is another means to ensure that performance measurements are observed at several levels of granularity, using microbenchmarks and macrobenchmarks, as advocated by Traeger et al. [15] and as reported by Clark et al. [5], who reproduced earlier work by Barham et al. [1].

Comparability/Portability: While Traeger et al. [15] suggested many ways to improve the comparability of results across systems and over time, facilitating relative comparisons, they saw the task of performing accurate comparisons as being an open question deserving of future work. The works of Chen et al. [4] and Clark et al. [5] were immediately concerned with the comparability of results and the portability of experimental protocols and parameters; the former attempted this within

the same experiment across three commodity operating systems, while the latter aimed to compare against prior work [1] using different underlying hardware.

Reproducibility/Transferability: Many of the papers surveyed speak to the need for reproducible results and transferable evaluation methodologies. This was an overarching motivation for the development of *hbench:OS* [3], described as being more than just a set of tools but also a methodology. Traeger et al. [15] stressed the need to specify one's methodology, documenting *how* an evaluation was performed. They lead by example, documenting their own experiments reported throughout their article in great detail. Interestingly, Chen et al. [4] do not speak about reproducibility in their article; relative to the other papers surveyed, I found the description of their experimental protocol to be the most imprecise and lacking in rationale; it would likely be difficult to transfer their hardware-specific methodology to three other operating systems. Finally, the importance of repeated research is no less stated than in Clark et al.'s [5] article, whose principal contribution is a successful reproduction of prior research findings, which also served to reaffirm the strength of the original work [1].

5. CONCLUSION AND FUTURE WORK

A line of potential research projects could involve compiling evaluation guidance particular to specific topic areas within systems literature, akin to Treager et al.'s [15] survey of recent file systems research. A new survey of recent papers with respect to statistical rigour in performance evaluation might also be of value, especially if compared to similar work performed in the late 1990s [13]. Development of new experimental protocols is also promising direction of future work, an aim of which should be to address the unmet needs identified in Section 3.4.4.

In this paper I presented a survey of four research papers pertaining to operating system performance evaluation. I summarized the approaches to system evaluation, which included micro- and macro-benchmarking, trace-based evaluation, application-based evaluation, as well as combined or hybrid approaches.

The surveyed position, workshop, and research papers illustrated the prominent challenges and tradeoffs associated with evaluation: realism, the granularity of measurement, the portability of experimental protocol, as well as the comparability and reproducibility of methods and results. The existence and impact of the papers surveyed have demonstrated that systems research, being representative of the nascent field of computer science, is moving toward a rigorous and standardized science of measurement and evaluation.

6. REFERENCES

1. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. ACM Symp. Operating Systems Principles (SOSP)* (2003), 164–177.
2. Bershad, B. K., Draves, R. P., and Forin, A. Using microbenchmarks to evaluate system performance. In *Proc. IEEE workshop on Workstation Operating Systems* (1992), 148–153.
3. Brown, A. B., and Seltzer, M. I. Operating system benchmarking in the wake of lmbench. *ACM SIGMETRICS Performance Evaluation Review 25*, 1 (1997), 214–224.
4. Chen, J. B., Endo, Y., Chan, K., Mazières, D., Dias, A., Seltzer, M., and Smith, M. D. The measured performance of personal computer operating systems. *ACM Trans. Computer Systems (TOCS) 14*, 1 (1996), 3–40.
5. Clark, B., Deshane, T., Dow, E., Evanchik, S., Finlayson, M., and Herne, J. Xen and the art of repeated research. In *Proc. USENIX Annual Technical Conf.* (2004), 135–144.
6. Engler, D. R., Kaashoek, M. F., and O'Toole, J. J. Exokernel: An operating system architecture for application-level resource management. In *Proc. ACM Symp. Operating Systems Principles (SOSP)* (1995), 251–266.
7. Katcher, J. Postmark: A new filesystem benchmark. Tech. rep., TR3022, Network Appliance, 1997.
8. Liedtke, J., Islam, N., Jaeger, T., Panteleenko, V., and Park, Y. Irreproducible benchmarks might be sometimes helpful. In *Proc. ACM SIGOPS European workshop on Support for Composing Distributed Applications* (1998), 242–246.
9. McVoy, L., and Staelin, C. lmbench: Portable tools for performance analysis. In *Proc. USENIX Annual Technical Conf.* (1996), 279–294.
10. Mogul, J. C. Brittle metrics in operating systems research. In *Proc. IEEE workshop on Hot Topics in Operating Systems* (1999), 90–95.
11. Rosenblum, M. *The design and implementation of a log-structured file system.* PhD thesis, Electrical Engineering and Computer Sciences, Computer Science Division, University of California, 1992.
12. Seltzer, M., Krinsky, D., and Smith, K. The case for application-specific benchmarking. In *Proc. IEEE workshop on Hot Topics in Operating Systems* (1999), 102–107.
13. Small, C., Ghosh, N., Saleeb, H., Seltzer, M., and Smith, K. Does systems research measure up? Tech. rep., TR-16-97, Harvard University., 1997.
14. Staelin, C. lmbench: an extensible micro-benchmark suite. *Software: Practice and Experience 35*, 11 (2005), 1079–1105.
15. Traeger, A., Zadok, E., Joukov, N., and Wright, C. P. A nine year study of file system and storage benchmarking. *ACM Trans. Storage 4*, 2 (2008), 1–56.