Benchmarking Operating Systems[*]

Nicholas Hatt
Oberlin College '08
nhatt@cs.oberlin.edu

Axis Sivitz
Oberlin College '08
asivitz@cs.oberlin.edu

Benjamin A. Kuperman[†]
Oberlin College
Computer Science
kuperman@cs.oberlin.edu

ABSTRACT
Benchmarking of computer systems is an important, albeit sometimes tedious task that gives insight into the performance of a system, exposes flaws, and allows for comparison between systems or versions. Current benchmarking suites for the UNIX operating system are strongly weighted towards CPU and hardware performance. As operating systems grow more complex, so do the applications running on them. The type of open benchmarks that are widely available today do not take this into account, making them somewhat unrealistic and uninformative. We present our solution to this problem which was used to benchmark a library for UNIX we were developing this summer. We discuss tools used, rationale, and conclusions based on the efficacy of each test. We also present drawbacks and future directions for the work.

1. INTRODUCTION
Benchmarking produces valuable data, exposes bugs and flaws in a system, and can offer a straightforward, numerical comparison of the performance difference between two systems. As the complexity of computer systems increases, it is harder to discover whether new software will *increase* performance, *decrease* performance, or have no effect at all. Determining impact is very important. For example, a new feature meant to increase performance at the cost of code complexity may not warrant inclusion if the benefit is not measurable, or a new security feature may significantly decrease performance. A developer can save time, effort, and money by immediately discovering how his or her changes impact performance. A benchmarking program can give the developer rudimentary feedback on the impact of their changes without having to get feedback from users.

[*]Result of summer undergraduate research in 2007
[†]Faculty Advisor

2. LEGACY BENCHMARKS
Several benchmarking suites are available for Linux, including LMbench[1], UnixBench[2], and HBench-OS[3]. These three benchmarks have not been updated since the 1990's. In the time since then, operating systems have grown in complexity and these tools do not exercise a large part of a system's functionality. The benchmarks rely on CPU intensive benchmarks, e.g., floating point calculations, so they are useful in comparing two different hardware configurations. In our research, we want to see what kind of performance impact our changes have on total system usage. Hardware benchmarking tells us nothing. With few alternatives, however, these legacy tools are still in use today. [3]

[1]Website at `http://www.bitmover.com/lmbench/`
[2]Available at `http://www.tux.org/pub/tux/niemi/unixbench/`
[3]Website at `http://www.eecs.harvard.edu/vino/perf/hbench/`

2.1 LMbench

LMbench is a tool that is designed to test the speed of data transfer between processor, memory, cache, disk, and network.[4] To a system designer, these present the biggest bottlenecks in hardware. Specifically, latency and bandwidth issues can be found using LMbench. However, because these tests specifically measure hardware performance, they are not useful as system software benchmarks.

2.2 HBench-OS

HBench-OS is benchmarking suite developed as part of a senior thesis at Harvard.[1] Based on our own analysis of the source code, it is almost identical to LMbench in terms of actual tests done, as it is branched from LMbench. The difference is in the testing methodology; specifically, making the tests easier to set up and run. This provides better data, but not better benchmarks.

2.3 UnixBench

UnixBench is a benchmarking suite with that seeks to test overall system performance. The tests focus on different aspects of OS functionality (process spawning, interprocess communication, filesystem throughput), but only using small, focused programs. In other words, the tests cover a large amount of functionality, but in a way that is unlike the normal operation of the system. Thus, the results cannot be trusted as representative of normal system performance.

All three benchmarking suites contain tests which measure certain aspects of OS performance. These include a C compiler test, a system call overhead test, and a `fork`()+exec() test. These tests have the right idea, but fail to fully encompass the complexity of a modern operating system. For example, the C compiler test compiles the famous "Hello World" program, only 3 lines of code. It then does this over and over to get an average. While compilation may be a good measure of everyday use, the lack of complexity in this benchmark does not paint as accurate of a picture as we would like. After all, who besides a CS student would compile Hello World?

The system call (syscall) overhead and the `fork`()+exec() benchmarks have the same flaw. For syscalls, the benchmark will loop, calling a syscall, and see how long it takes to perform a set number of iterations. Fork+exec is the same way, looping and creating as many new processes as possible. These are both critical OS primitives, but they do not take into account the complex interaction between these elements in a normal program or system operation. The benchmark is asking "How fast can you do this one thing?" which can be a useful measure of changes to either `fork()` or `exec`; however, we really want to know how it behaves while performing any number of different tasks simultaneously. For this reason, these benchmarks skew towards being CPU-dependent benchmarks. Furthermore, modern computer systems make extensive use of caching to obtain performance gains in highly repetitive situations. Thus, small benchmarks might be hiding the actual performance cost in a normal situation behind the performance benefits of the cache.

2.4 The Need to Move Away From Traditional Benchmarks

The types of changes that we need benchmarks to test are system-wide software changes that do not require new hardware at all. For example, if we install a new version of libc, we need to see how it performs compared to older versions. In the case of LMbench, the syscall benchmark will only tell us how one of the hundreds of syscalls is performing. Or, when testing a new compiler, traditional benchmarks cannot really tell us if the new compiled code is any faster. These benchmarks do not really measure the performance of real world applications, which is why we need a system benchmarking mechanism.

3. MODERN SYSTEM BENCHMARKING

System benchmarking is necessary but largely neglected and non-standardized. In late July, 2007, the community of Linux kernel developers debated the merits of two different CPU schedulers.[2] The community had to rely largely on user testimonials because the existing tools were insufficient. In general, current problems of CPU scheduling are related to the "interactivity" of the computer system, or rather, the ability for the CPU scheduler to discern which programs are most important to the user. Existing benchmarking tools are not equipped to measure interactivity. The developer of one scheduler tried to quantify his suspected improvements by writing his own benchmarking software, but also admitted they were far from ideal.[5]

4. FINDING A BETTER BENCHMARK

System benchmarks should be based on actual system usage, so it makes sense to run similar programs to those that the deployed system would run. However, a benchmark, by definition, should be reproducible and give specific information on its performance. Interactive programs are useful for this type of benchmarking but it is hard to reproduce results in such an environment. Some, however, include their own benchmarking tool that requires no user interaction. These tools may serve as starting points or building blocks for a suite of system benchmarks. However, because they are limited in scope to the behavior of only one program, they cannot measure the performance of an entire multi-user system.

The tool we developed, Audlib, is a interposable library which may potentially affect the performance of every userspace program in the system. Therefore, it is very important to be aware of possible system performance degradation. Any one program may not show decreased performance, even if another does, so it is important that we perform a large variety of benchmarks.

We have selected tools that meet the following requirements:

1. **Open source:** We are developing an open source tool and if our results need to be duplicated we should use software that is freely available. Open source is free, which removes a prohibitive factor from the deployment of the benchmark.
2. **Complex:** The benchmark should not be based on raw throughput but rather accomplishing a complex task similar to real-world usage.
3. **Quantifiable:** We want benchmarks, and the ability set up a controlled environment is essential.

4.1 Candidates

4.1.1 Apache Webserver

Apache web server is the most popular web server in use today. In July of 2007 Apache served 52% [6] of all websites. It is an open source application and has a built-in benchmark. This tool, `ab`, or Apache Benchmark, stress tests the HTTP server by making multiple requests on it and collecting data. It allows the user to specify the number of requests made, as well as number of concurrent requests to be made. This makes it a good test of concurrent processes, as the `httpd` program spawns several processes to handles requests. The tool `ab` can be invoked with any server as an argument. This allows it to be run from a remote machine, adding more control to the benchmark.

4.1.2 Linux Kernel Compilation

The Linux kernel[4] is a large, complex, widely available and widely used open source project.[5]

[4]http://kernel.org

[5]http://en.wikipedia.org/wiki/Linux

Building the kernel requires a large number of potentially parallel compilations, which also makes it an interesting benchmark for multi-processor systems. It exercises various subsystems, including memory management, I/O control, and process scheduling, among others. The actual kernel built, and therefore the time it takes to build, changes depending on the configuration used. We used the default configuration, which can be generated with `make menuconfig` followed by immediately exiting. Options may be added or removed if an easier or harder test is desired. To actually gather information on the time it takes to compile, run `(make clean; time make)`.

4.1.3 *Vim Build Test*

Vim[6] is a modern update to a standard UNIX text editor `vi`, used by many programmers and UNIX command line users.[7] After building the program, Vim includes a test suite which exercises the various components of the editor. Since text editing is a common task on a multi-user computer system, the Vim self-test should serve as a good benchmark. The command to run the test is `(make clean; make; time make test)`.

4.1.4 *Quake III*

Quake III is a 3-D computer game released in 1999 by id Software[7]. The source code was released in 2005 under the GPL open source license. The game includes benchmark functionality in which a demo is run as quickly as possible and the average number of graphical frames rendered per second is calculated. This can be run by starting up the game and then entering the commands `timedemo 1` and `demo four` into the console. While much of the performance for 3-D games relies on the graphics hardware, we wanted to make sure that we had no negative performance impact in such an interactive environment.

[6]http://www.vim.org

[7]`http://www.idsoftware.com/games/quake/quake3-arena/`

4.2 Evaluation

4.2.1 *Apache Webserver*

The `ab` tool was run multiple times to eliminate the effects of caching and possible network interference. Each run with 100,000 requests produced about 50 MB of data from our library. This indicated that the benchmark was exercising the system at a good rate. The ability to specify the number of requests is an advantage. The tester can easily adjust the length of the test this way, or increase the load by making more concurrent requests. The flexibility of the benchmark combined with the reporting tools that it offers, make it a key benchmark for anyone wanting to gauge system performance. The only drawback would be the fact that setup is slightly difficult. For those inexperienced in setting up a webserver, there may be a few hurdles, but nothing unsurmountable, as the process is well documented.

4.2.2 *Linux Kernel Compilation*

The full build took roughly twenty minutes without our program, and almost twenty six with it. This demonstrates the significant impact our tool may have on system performance, depending on the configuration. In this situation, kernel compilation acts as a good benchmark for us because its performance is sensitive to the software configuration of the system.

4.2.3 *Vim Build Test*

The Vim build test automatically performs many editing commands on many temporary files. This should mimic the text editing actions of a normal user, but of course, it runs much faster than a normal user would type. On a sample run (in which we chose to include the compilation step), we logged over 1.4 million calls to the `printf` library function and over 150 thousand calls to `open`. This shows a significant amount of terminal and disk I/O.

4.2.4 *Quake III*

The Quake III test showed little slowdown, and in some cases a speedup under our new libraries. A drawback to using this benchmark is that it cannot be easily automated. The commands need to be run from inside the game's terminal, requiring user input on each run of the benchmark. Aside from this however, the test can be reproduced accurately, which makes this a good benchmark.

5. BUILDING A BETTER BENCHMARK

In the computing world, benchmarks have a much higher profile role than helping software developers with their work. The hardware industry has long had access to a variety of reproducible, sufficiently complex benchmarks. No graphics card or CPU escapes benchmark evaluation, and purchases are often made after only considering those results. Could a system benchmark hold such a high level of esteem?

The key to a system benchmark's success may be its robustness in measuring different aspects of the system software.

The ideal tool would obtain specific performance information on all components of a system, including:

1. **Memory** A modern system gains many performance improvements from the proper use of available memory. A benchmark should measure how quickly critical information can be retrieved.
2. **Filesystem** Files are organized by the filesystem. The theory and implementation of the file system can drastically impact the read and write time of every file. A benchmark should minimize the role of the actual media that the filesystem manages.
3. **Process Control** Modern systems can handle many different programs running in parallel. A benchmark should put the system under extreme load and measure the degradation of performance. This would provide especially useful information on multiprocessor systems, where scheduling algorithms are more important and more complex.
4. **Display** Because almost every modern system relies on a graphical user interface, a benchmark should test a system's ability to render standard widgets.
5. **User Input** "Interactivity" is essentially the system's ability to respond quickly and efficiently to user input. A benchmark should simulate variety of inputs (mouse, keyboard, etc.) and evaluate the response.

6. CONCLUSIONS

Would the software industry welcome a standard system benchmarking suite? Benchmarks are widely used in the hardware industry because they specifically showcase the fruit of their efforts. New hardware is designed, built, and sold with performance in mind. In general, software does not have the same obsession with speed. The hardware industry has actually boomed largely to compensate for *slower* systems. Computers generally require the same amount of time to boot up and they show the same historic responsiveness, or lack thereof. Few new operating systems are released claiming increased performance. That is not to say that software developers neglect performance completely. Rather, performance work is a means, not an end. It is necessary to balance the increased complexity of new features, but not desired in and of itself.

7. FUTURE WORK

Much of the hard work here is done. We have researched and evaluated several candidates and have a clear picture of what to include in our own "suite" of benchmarks. The next step would be to begin implementing the system benchmark suite. Next summer, we could take the system benchmarks we have worked with (complex compilation, text editing, 3D game, etc...) and either strip them of some of their complexity, or create a test from scratch that is inspired by one of the benchmarks we used. For instance, we could compile a smaller section of the Linux Kernel, and we could make a standalone 3D program that requires no user input. The idea would be to automate the tests, make them run within a reasonable amount of time (probably less than a minute for each), and also remove most dependencies. The benchmarking suite should be a stand alone program. Next, we should go through our list of system components and build tests that stress areas that the existing programs don't cover. Finally, benchmark output should be tweaked to give meaningful output, so that comparisons can be easily made between different systems.

8. REFERENCES

[1] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems Annual Technical Conference*, June 1997. `http://www.eecs.harvard.edu/vino/perf/hbench/sigmetrics/hbench.pdf`.

[2] Linux: CFS and 3D Gaming, July 2007. `http://kerneltrap.org/node/14023`.

[3] Linux: Tuning CFS, August 2007. `http://kerneltrap.org/node/14055`.

[4] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference*, pages 279–294, Jan. 1996. `http://www.bitmover.com/lmbench/lmbench-usenix.ps.gz`.

[5] A. Mills. Why I quit: kernel developer Con Kolivas, July 2007. `http://apcmag.com/6735/interview_con_kolivas`.

[6] Netcraft web server survey, July 2007. `http://survey.netcraft.com/Reports/200707/byserver/`.

[7] Vim (text editor), August 2007. `http://en.wikipedia.org/wiki/Vim_%28text_editor%29`.