

# Versatile, Portable, and Efficient OS Profiling via Latency Analysis

Nikolai Joukov, Rakesh Iyer, Avishay Traeger, Charles P. Wright, and Erez Zadok

Stony Brook University

{kolya,riyer,atraeger,cwright,ezk}@cs.sunysb.edu

Operating systems are complex and their behavior depends on many factors. Source code, if available, does not directly help understand the OS’s behavior, as the behavior depends on actual workloads and external inputs. Runtime profiling is a key technique for understanding the behavior and mutual-influence of modern OS components. Such profiling is useful to prove new concepts, debug problems, and optimize the performance of existing OSs. Unfortunately, existing profiling methods lack in important areas: they do not provide much of the necessary information about the OS’s behavior; they require OS modification and therefore are not portable; or they exact high overheads thus perturbing the profiled OS.

The latency of any OS operation contains important information about its execution. Latency can be easily collected but cannot be easily analyzed; it contains a mix of latencies of different execution paths and can be drastically affected by preemption. That is why even early OS profiling tools rejected latency as a viable performance metric in multi-tasking environments. However, the disadvantages of latency measurements have, in fact, three advantages. First, latency contains information about all the operations performed in the kernel. Even asynchronous OS activity such as interrupts or kernel threads influence the latency of OS requests. This allows us to monitor and analyze all these internal events even if direct instrumentation of kernel components is impossible or undesirable. Second, the latency of many operations can be captured entirely from user mode. This means that most of the OSs can be profiled the same way. Also, this allows profiling of OSs for which the source code is unavailable. Third, capturing latency adds only small overheads.

These attractive features of latency motivated some authors to use it in their projects. Some authors used a simple assumption that there is one or several dominant latency contributors known in advance [3, 4]. Others correlated latency distribution changes with system parameters [1, 2].

We have designed a general method of OS profiling based on the latency distributions. We collect latency distributions in exponential buckets. For example, Figure 1 shows profiles (latency distributions) of the `read` and `llseek` operations captured from the user mode. The shapes of the `llseek` and the `read` profiles are correlated for two processes. This suggests that these two operations compete for a semaphore in the kernel.

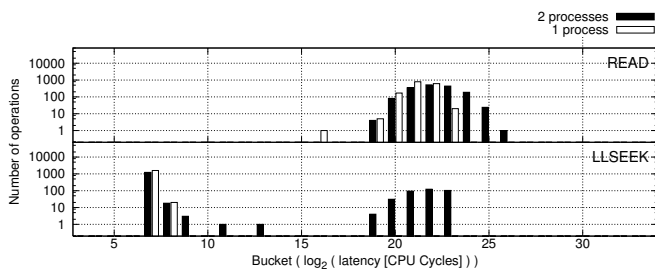


Figure 1: The `llseek` and `read` Linux 2.6.11 operations lock contention. Note: that both axes are logarithmic.

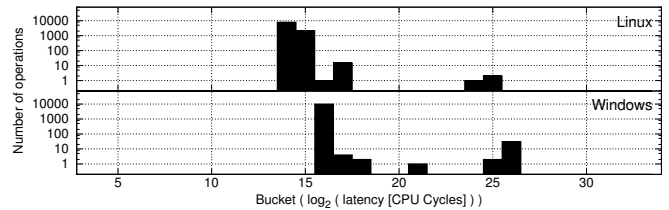


Figure 2: Profile of the `read` operation of an I/O-intensive process that sequentially reads data. Another CPU-only process runs concurrently on the background. Ext3 running on Linux 2.6.11 (top) and NTFS running on Windows XP (bottom).

We have analyzed conditions that allow us to use latency analysis even under preemptive kernels. Thus, we have analytically shown that in most cases preemption effects can be ignored. For example, we have calculated and experimentally confirmed that the probability that an I/O-intensive process is forcibly preempted is small and can be as low as  $10^{-280}$ . For some other workloads, forcible preemption can be used to study the behavior and impact of the scheduler and interrupts processing. Figure 2 shows that the Linux 2.6.11 and Windows XP schedulers behave differently if I/O-intensive and CPU-only processes run concurrently. The peak across the buckets in 24–26 corresponds to the forcible preemption. The peak’s location corresponds to the time the CPU-consuming process was running—its time quantum. We have measured that the same peak is located in bucket 26 if no CPU-consuming process is running. We can conclude that Linux penalizes the CPU-consuming process more by reducing its scheduling quantum by a larger amount.

Our real-time OS profiling method is versatile, portable, and efficient. It uncovers details of the internal OS’s operation and, at the same time, has a small overhead (less than 4% of the CPU time even for intensive workloads). Our method is portable because we can intercept operations and measure OS behavior entirely from the user-level.

## References

- [1] M. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 309–322, San Francisco, CA, March 2004. USENIX Association.
- [2] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI 1996)*, pages 261–275, Seattle, WA, October 1996.
- [3] J. Nugent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Controlling Your PLACE in the File System with Gray-box Techniques. In *Proceedings of the Annual USENIX Technical Conference*, pages 311–323, San Antonio, TX, June 2003. USENIX Association.
- [4] Y. Ruan and V. Pai. Making the “Box” Transparent: System Call Performance as a First-class Result. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, Boston, MA, June 2004. USENIX Association.