Versatile, Portable, and Efficient File System Profiling

A Dissertation Presented

by

Nikolai Joukov

to

The Graduate School

in Partial fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-06-05

December 2006

Stony Brook University

The Graduate School

Nikolai Joukov

We, the dissertation committee for the above candidate for the Ph.D. degree, hereby recommend acceptance of this dissertation.

Dr. Erez Zadok, Advisor
Professor, Computer Science Department, Stony Brook University

---

Dr. Samir Das, Chair of Defense
Professor, Computer Science Department, Stony Brook University

---

Dr. Scott Stoller
Professor, Computer Science Department, Stony Brook University

---

Dr. Ethan Miller
Professor, Computer Science Department,
University of California Santa Cruz

This dissertation is accepted by the Graduate School

Dean of the Graduate School

Abstract of the Dissertation

Versatile, Portable, and Efficient File System Profiling
by

Nikolai Joukov

Doctor of Philosophy
in
Computer Science

Stony Brook University

2006

File systems are complex and their behavior depends on many factors. Source code, if available, does not directly help understand the file system'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 file systems. Unfortunately, existing profiling methods are lacking in important areas: they do not provide much of the necessary information about the file system's behavior, they require OS modification and therefore are not portable, or they exact high overheads thus perturbing the profiled file system.

We developed a direct, real-time file system profiling method based on the analysis of latency distributions. Our method is versatile: a suitable workload can be used to profile virtually any OS component. Our method is portable because we can intercept operations and measure file system behavior from the user level or from inside the kernel without requiring source code. Our method is efficient: it has small overheads (less than 4% of the CPU time). Moreover, if the source code is available, we can use it to reduce overheads even further.

In this dissertation we describe our profiling method, the theory behind it, and the automation of the profile analysis. We demonstrate the usefulness of our method through a series of profiles conducted on Linux, FreeBSD, and Windows, including client/server scenarios. We discovered and investigated a number of interesting interactions, including scheduler behavior, multi-modal I/O distributions, and a previously unknown lock contention, which we fixed. We use our profiling method for performance analysis of a complex RAID-like fan-out stackable file system called RAIF that we have developed.

Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank my advisor, Erez Zadok, for his guidance and thoughtful advice. He not only gave direction to my research but also gave me the freedom to explore new and sometimes risky projects. Erez is always the best example for me of how to budget time, to lead others, and deal with different people. I am very thankful that he was always able to find time to meet with me when I needed it.

My committee members, Samir Das, Ethan Miller, and Scott Stoller, provided many valuable comments and asked interesting questions in a friendly way. I am grateful to Ethan Miller for changing his airplane tickets and coming to my defense directly after a long conference in Florida.

Tzi-cker Chiueh supported my research and helped me during my first years at Stony Brook University. Michael Bender, Klaus Mueller, Alex Mohr, and Steve Skiena provided valuable comments. Andrea and Remzi Arpaci-Dusseau from the University of Wisconsin-Madison reviewed early paper drafts and were very supportive of the project. Bill Yurcik from NCSA helped with the RAIF project presentation.

I would like to thank every member of the File systems and Storage Laboratory for creating a fruitful environment to work and always being available to help before a paper deadline, to prepare a talk, and celebrate our success afterwords. Charles P. Wright and Avishay Traeger are not just the lab mates who shared an aisle with me but also my true friends who helped me many times. Charles coined the term TMAP and is the person who can quickly resolve all kinds of issues. Avishay happened to have a spare bottle of Jägermeister in his fridge every time we had a party. He also spent many sleepless nights capturing our profiles, benchmarking RAIF, and working on the automated profile analysis. Sean Callanan is my good friend and neighbour who proofread all the papers that I have written. Rick Spillane helped me before several deadlines. He is also the only person after I quit kindergarten who managed to bite me. Rakesh Iyer ported our profilers to Windows XP while stretching in his chair. Akshat Aranya tried the idea of latency profiling using Tracefs's aggregate driver. He also contributed to the initial Replayfs prototype. Tim Wong spent months coding and debugging Tracefs and Replayfs. Harry Papaxenopoulos created secure deletion patches between bouts of self-incrmination. Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, and Sunil Satnur significantly improved RAIF. Dave Quigley also helped with its porting to 2.6 Linux kernels. Jeff Sipek was the person who always got excited every time I found bugs in the Linux kernel. Mohan-Krishna Channa-Reddy, Jay Pradip Dave, Puja Gupta, Aditya Kashyap, Devaki Kulkarni, Adam David Alan Martin, Kiran-Kumar Muniswamy-Reddy, Yeugeniy Miretskiy, Gopalan Sivathanu, and Joseph Spadavecchia were among the first people I met at FSL and who helped me to quickly become a true member of the lab.

I would like to thank my dear Jenya, Misha, and Sasha for their understanding and for being a continuous source of joy.

Finally, I would like to thank my parents. They were just a phone call away every time I needed their advice or support.

This work was sponsored by NSF (CAREER EIA-0133589, CCR-0310493, and HECURA CCF-0621463 awards) and HP/Intel (gifts numbers 87128 and 88415.1).

Chapter 1

Introduction

Profiling is a standard method to investigate and tune the operation of any complicated software component. Even the execution of one single-threaded user-level program is hardly predictable because of the underlying hardware behavior. For example, branch prediction and cache behavior can easily change the program execution time by an order of magnitude. Moreover, in a multi-tasking environments, processes compete with each other for a number of shared resources such as CPU, memory, shared data structures, buses, I/O devices, etc. In addition, there are a variety of possible external input patterns. Therefore, only runtime profiling can help understand the actual system behavior even if the source code is available. At first glance, it seems that observing computer software and hardware behavior should not be difficult, because these systems are human-made and therefore can be easily instrumented. However, profiling has several contradicting requirements: versatility, portability, and low overheads.

1.1 Contradicting Profiling Requirements

Versatility. A versatile system profile should contain information about the interactions with all software and hardware components and allow correlation of related information that was captured at different levels of abstraction. For example, a file system operates on files, whereas a hard-disk driver operates on data blocks. However, the operation and performance of file systems and drivers depend on their complex interactions; contention on semaphores can change the disk's I/O patterns, while a file-system's on-disk format can dramatically change its I/O performance.

Portability. To gather the information about all the different system operation aspects at all the levels of system abstraction, one usually tries to instrument the system as much as possible (*e.g.*, DTrace [18] adds tens of thousands of probes to the Solaris kernel). However, there are two big problems associated with this approach.

1. Direct instrumentation of systems is not *portable*. System instrumentation is OS-version–specific or compiler-version–specific and also may depend on the hardware architecture. Therefore, profilers for new OSs are often not available because existing profilers have to be ported to each new OS version.

2. It is not possible to instrument everything. For example, one can spend a lot of time and add many instrumentation hooks into the kernel but there still will be uninstrumented places in the code. More importantly, however, it is not possible to instrument some of the system components because their source is unavailable (*e.g.*, firmware of hard drives or Windows scheduler).

Low overheads. Low overheads are crucial for profiling because high overheads can significantly change the system's behavior. However, per-event instrumentation (*e.g.*, instrumentation of every semaphore) adds overheads on a per-event basis (*e.g.*, for each taken semaphore). To minimize overheads, several hardware components provide profiling help. For example, modern CPUs maintain statistics about their operation [14]. However, only the OS can correlate this information with higher level information, such as the corresponding process. Therefore, some CPU time overheads are inevitable.

As we can see, versatility, portability, and efficiency contradict each other. Versatility requires collecting more information which requires more instrumentation that in turn means less portability and higher overheads. Higher portability means less instrumentation and less OS-specific and hardware-specific performance optimizations. Low overheads require fewer instrumentation points (which decreases versatility) and more non-portable optimizations. As a result, existing profiling tools provide limited information, are not portable (usually even between OS minor versions) and add high overheads.

1.2 Our Approach

We developed a gray-box system profiling method. For example, user applications make requests via system calls and external network requests come via the network interface. The latency of these requests contains information about related CPU time, rescheduling, lock and semaphore contentions, and I/O delays. Capturing latency is fast and easy. However, the total latency includes a mix of many latencies contributed by different execution paths and is therefore difficult to analyze. Process preemption complicates this problem further. All existing projects that used latency as a performance metric used some simplistic assumptions applicable for a particular case. Some authors assumed that there is only one source of latency which can be characterized by the average latency value [27, 30, 46, 88]. Others used prior knowledge of the latencies' sources to classify the latencies into several groups [8, 16, 75]. Past attempts to analyze latencies more generally just looked for distribution changes to detect anomalies [21]. Our profiling method allows the investigation of latencies in the general case.

We accumulate the distributions of logarithms of latencies for each OS operation at runtime, and later process the accumulated results. This allows us to efficiently capture small amounts of data that embody detailed information about many aspects of internal OS behavior. Different OS internal activities create different peaks on the collected distributions. The resulting information can be conveniently presented in a graphical form.

We created user-level profilers for POSIX-compliant OSs and kernel-level profilers for Linux, FreeBSD, and Windows—to profile system activity for both local and remote computers. These tools have CPU time overheads below 4%. We used these profilers to investigate internal file system behavior under Linux and Windows. Under Linux we discovered and characterized several semaphore and I/O contentions. Source code availability allowed us to verify our conclusions and fix the problems. Under Windows we observed internal lock contentions even without access to source code; we also discovered a number of harmful I/O patterns including those for networked file systems.

Our method is a general profiling and visualization technique that can be applied to a broad range of problems. Nevertheless, it requires skills to analyze collected profiles. In this dissertation we present several profile analysis methods and their automation. We also analyze several method-specific problems like process preemption effects, time synchronization on SMP systems and profiles locking on multi-CPU systems.

1.3 OSprof and FSprof

The proposed profiling method can be applied to a wide range of systems ranging from individual hard drives to complex RAID controllers, OSs, and distributed systems. We call our profiling method *OSprof* when used to profile OSs (including distributed ones). *FSprof* is a subset of OSprof and is a file system profiling extension.

In this dissertation we concentrated on file system profiling for three reasons: (1) file system profiles are complex and contain information about most OS components and interactions. Therefore, file system profiles are a good example to demonstrate the power of OSprof. (2) file systems are a substantial part of the OSs. For example, Linux 2.6.11.7 supports 53 different file systems, ranging from memory and disk-based ones (Ext2, Ext3, Reiserfs, XFS, UFS/FFS, and more), to network file systems, (NFS, SMB/CIFS, NCPFS), to distributed ones (*e.g.*, Coda), and many more specialized ones (*e.g.*, `/proc`, `/dev`, debugfs, and more). These file systems total 485,158 lines of complex code, out of 2,997,507 lines of code in the entire Linux 2.6.11.7 kernel (not counting device drivers). In addition, many file systems are developed and maintained outside the Linux kernel [6, 11, 55, 81, 102, 109, 112]. (3) file systems is the main focus of our research group and we used our profiling method to profile all our new experimental file systems.

1.4 Thesis Organization

The rest of this dissertation is organized as follows. We describe background work in Chapter 2. Chapter 3 describes our profiling method and provides analysis of its applicability and limitations. In Chapter 4 we describe FoSgen—our file system source instrumentation system. Chapter 5 describes our implementation. We evaluate our system in Chapter 6. In Chapter 7 we present several usage scenarios and analyze profiles of several real-world file systems. Moreover, in Chapter 8 we show some examples how our file-system–level profiler can be used for profiling without buckets. In Chapter 9 we describe profiling of RAIF file system during its development. We conclude and describe future work in Chapter 10.

Chapter 2

Background

We have described our latency profiling method in several papers [49, 53, 54, 56]. Next, we describe related work done by others about kernel code profiling and kernel code instrumentation.

2.1 Kernel Code Profiling

Most of the existing kernel profilers concentrate on different aspects of the CPU execution. Only a few profilers can profile lock-related behavior on some operating systems. Even fewer tools can profile the system I/O and no tools can satisfactorily correlate I/O requests with the high-level file system requests.

2.1.1 CPU Execution Profiling

The de facto standard of CPU-related code execution profiling is program counter sampling. Unix *prof* [10] instruments source code at function entry and exit points. An instrumented binary's program counter is sampled at fixed time intervals. The resulting samples are used to construct histograms with the number of individual functions invoked and their average execution times. Program counter (PC) sampling is a relatively inexpensive way to capture how much CPU a program fragment uses in multi-tasking environments where a task can be rescheduled at any time. *Gprof* [36] additionally records information about the callers of individual functions, which allows it to construct call graphs. Gprof was successfully used for kernel profiling in the 1980s [69]. However, the instrumented kernels had a 20% increase in code size and an execution time overhead of up to 25%. *Kernprof* [94] uses a combination of PC sampling and kernel hooks to build profiles and call graphs. Kernprof interfaces with the Linux scheduler to count the time that a kernel function spent sleeping (*e.g.*, to perform I/O) in the profile. Unfortunately, Kernprof requires a patch to both the kernel and the compiler.

More detailed profiles with granularity as small as a single code line can be collected using *tcov* [98]. Most modern CPUs contain special hardware counters for use by profilers. The hardware counters allow correlating profiled code execution, CPU cache states, branch prediction functionality, and ordinary CPU clock counts [5, 14]. The counter overflow events generate a non-maskable interrupt (NMI). This allows sampling even inside device drivers as implemented in *Oprofile* [66]. Overall, such profilers capture only CPU-related information.

2.1.2 Locks and Memory Profiling

There are a number of profilers for other aspects of OS behavior such as lock contention [15, 74]. They replace the standard lock-related kernel functions with instrumented ones. This instrumentation is costly: Lockmeter adds 20% system time overhead. Other specialized tools can profile memory usage, leaks, and caches [92].

2.1.3 File System and I/O Profiling

Fewer and less developed tools are available to profile file system performance, which is highly dependent on the workload. Disk operations include mechanical latencies to position the head. The longest operation is seeking, or moving the head from one track to another. Therefore, file systems are designed to avoid seeks [70, 86]. Unfortunately, modern hard drives expose little information about the drive's internal data placement. The OS generally assumes that blocks with close logical block numbers are also physically close to each other on the disk. Only the disk drive itself can schedule the requests in an optimal way and only the disk drive has statistical information about its internal operations. The Linux kernel optionally maintains statistics about the block-device I/O operations and makes those available through the `/proc` file system, yet little information is reported about timing.

Network packet sniffers [37] capture traffic useful for analysis [31]. They are useful for analyzing protocols. Their problems are similar to those of hard disk profilers: both the client and server often perform additional processing that is not captured in the trace: searching caches, allocating objects, reordering requests, and more.

2.2 Latency-Based Profiling

The latency of a file system operation contains important information about its execution. Latency can be easily collected but cannot be easily analyzed because it contains a mix of latencies of different execution paths. Many authors used a simple assumption that there is one dominant latency contributor and that the average latency can characterize it [2, 7, 27, 46]. This simple assumption allowed to profile several OS components including timer interrupts on an idle system [32]. DeBox and LRP investigate average latency changes over time and its correlation with other system parameters [30, 88]. Chen and others moved one step further and observed changes in the distribution of latency over time and its correlation with software versions to detect possible problems in network services [21]. Prior knowledge of the underlying I/O characteristics and file system layouts allows categorization of runtime I/O requests based on their latency [8, 16, 75, 82].

2.3 File System Operations Interception

The addition of control interception points is a well developed research area. We will focus on the four methods most relevant to file systems.

2.3.1 Source Code Instrumentation

The most popular one is direct source code modification, because it imposes minimal overhead and is usually simple. For example, tracking lock contentions, page faults, or I/O activity usually requires just a few modifications to the kernel source code [15, 88]. If, however, every function requires profiling modifications, then the compiler may conduct such an instrumentation (*e.g.*, the `gcc -p` facility). This method has a clear drawback: new code is required not only for every OS and every file system but also for different versions of OSs and file systems.

2.3.2 Dynamic Code Instrumentation

Some modern OSs provide hooks that allow dynamic instrumentation. For example, DTrace [18] on Solaris as well as Linux Trace Toolkit (LTT) [113] and Linux Security Modules (LSM) [107] on Linux provide interception points in many places. However, these instrumentation APIs are not portable across OSs and do not intercept all file system operations. For example, LSM do not intercept memory-mapped operations. Dynamic code instrumentation is possible by inserting jump operations directly into the binary [44]. Similarly, debugging registers on modern CPUs can be used to instrument several arbitrary code addresses at once [24].

2.3.3 Interception from the User-Mode

Some of the file system operations may be intercepted and changed entirely from the user-mode. First, system utilities can be substituted with wrapper scripts or other binaries. Second, system libraries can be instrumented directly. In both cases, some of the programs will not be instrumented either because they are not replaced or because they are statically linked. Moreover, some file system operations cannot be changed this way (*e.g.*, popular memory-mapped operations). FUSE [102] and extended `ptrace` [108] interfaces allow interception of all file system operations but add significant overheads.

2.3.4 Layered Interception

Stackable file systems are portable across OSs and across file systems [115]. They can be mounted over any lower file system, several file systems, or only a single directory or file. However, stackable file systems add overheads for all file system operations even if only a single operation is modified. In addition, stackable file systems use twice as many Virtual File System objects, thus reducing the overall size of file system caches.

Chapter 3

Profiling Method

OSs serve requests from applications whose workloads generate different request patterns. The latencies of OS requests consist of both CPU and wait times:

$$latency = t_{cpu} + t_{wait} \tag{3.1}$$

CPU time includes normal code execution time as well as the time spent waiting on spinlocks:

$$t_{cpu} = \sum t_{exec} + \sum t_{spinlock}$$

Wait time is the time a process was not running on the CPU. It includes synchronous I/O time, time spent waiting on semaphores, and time spent waiting for other processes or interrupts that preempted the profiled request midway:

$$t_{wait} = \sum t_{I/O} + \sum t_{sem} + \sum t_{int} + \sum t_{preempt}$$

$t_{preempt}$ is the time the process was waiting because it ran out of its scheduling quantum and was preempted. We will consider preemption in more detail later in Section 3.3. We begin by discussing the non-preemptive OS case.

Every pattern of requests corresponds to a set of possible execution paths S. For example, a system call that updates a semaphore-protected data structure can have two paths:

1. if the semaphore is available ($latency_1 = t_{cpu_1}$), or
2. if it has to wait on the semaphore ($latency_2 = t_{cpu_2} + t_{sem}$).

In turn, each t_j is a function with its own distribution. We can generalize that the $latency_s$ of paths $s \in S$ consists of the sum of latencies of its components:

$$latency_s = \sum_j t_{s,j} \tag{3.2}$$

where j is the component, such as I/O of a particular type, program execution-path time, or one of the spinlocks or semaphores.

To find all $t_j \in T$, it is necessary to solve the system of linear Equations 3.2, which is usually impossible because $\|T\| \geq \|S\|$ (there are usually fewer paths than time components). Non-linear *logarithmic filtering* is a common technique used in physics and economics to select only the major sum contributors [68]. We used latency filtering to select the most important latency contributors t_{max} and filter out the other latency components δ:

$$log(latency) = log(t_{max} + \delta) \approx log(t_{max}) \tag{3.3}$$

For example, for log_2, even if δ is equal to t_{max}, the result will only change by 1. Most non-trivial workloads can have multiple paths for the same operation (*e.g.,* some requests may wait on a semaphore and some may not). To observe multiple paths at the same time we store logarithms of latencies into buckets. Thus, a bucket b contains the number of requests whose latency satisfies:

$$b = \lfloor log_{2^{\frac{1}{r}}}(latency) \rfloor = \lfloor r \times log_2(latency) \rfloor \tag{3.4}$$

Plugging in Equation 3.3 we get:

$$b \approx \lfloor r \times log_2(t_{max}) \rfloor$$

A profile's bucket density is proportional to the resolution r. We usually used $r = 1$. However, $r = 2$, for example, would double the profile resolution (bucket density). Increasing the resolution adds only a slight overhead to CPU time. However, it increases the memory consumption by r times because higher resolution profiles have more buckets.

Figure 3.1 shows an actual profile of the Windows XP `CreateThread` function called by two processes concurrently. The bottom X axis shows the average buckets' latency in seconds. The top X axis shows the bucket number (logarithm of latency in CPU cycles). The Y axis shows the number of operations whose latency falls into a given bucket. Note that both axes are logarithmic.

Let us consider the profile shown in Figure 3.1 in more detail. We captured this profile entirely from the user level. In addition to this profile we captured another profile with only a single process calling the same `CreateThread` function; we observed that in that case there was only one (leftmost) peak. Therefore, we can conclude that there is some contention between processes inside of the `CreateThread` function. In addition, we can derive the information about (1) the CPU times necessary to complete a `CreateThread` request with no contention (average latency in the leftmost peak) and (2) the portion of the `CreateThread` code that is executed while a semaphore or a lock is acquired (average latency in the leftmost peak times half the ratio of elements in the rightmost and leftmost buckets).

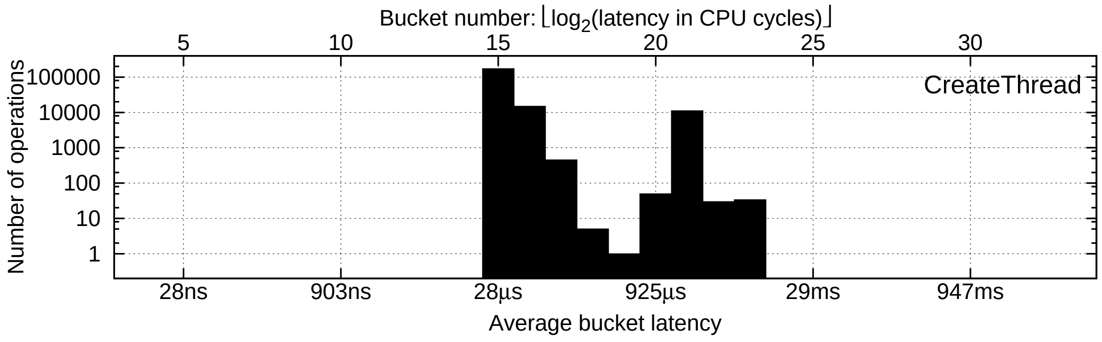


Figure 3.1: A profile of `CreateThread` operation on Windows XP, concurrently issued by two processes. The right peak corresponds to semaphore contention between the two processes. Note: both axes are logarithmic (x-axis is base 2, y-axis is base 10).

3.1 Profile Collection and Analysis

In general, we use the following methods to analyze profiles:

3.1.1 Profiles Preprocessing

A *complete profile* may consist of dozens of profiles of individual operations. For example, a user-mode program usually issues several system calls and a complete profile consists of several profiles of individual system calls. Thus, Figure 3.2 and Table 3.1 show the latencies for Linux 2.4.24 Ext2 for a run of *grep -r* over a Linux source tree. This example of a complete profile immediately informs us about the operations involved, their impact, and sometimes, their mutual dependence. For example, `lookup` is invoked only one less time than `read_inode`. The fact that the number of operations in the corresponding peaks is the same, and that `read_inode` is slightly faster than `lookup`, suggests that `read_inode` is called by the `lookup` operation, which is in fact the case. Ext2's `read` operation is implemented by calling the general-purpose Linux `generic_file_read` function, which then calls the `readpage` operation. Therefore, we can infer from Table 3.1 that the `lookup`, `read`, and `readdir` operations are responsible for more than 99% of the file system's latency under the given workload.

If the goal of profiling is performance optimization, then we usually start our analysis by selecting a subset of profiles that contribute the most to the total latency. We designed automatic procedures to:

- select profiles with operations that contribute the most to the total latency. Unless otherwise specified, figures presented in this dissertation show profiles with operations sorted according to their total latency; and
- compare two individual profiles and evaluate their similarity.

Operation	**Count**	**Total delay (10^6 CPU Cycles)**	**Total delay (ms)**
`readdir`	1,687	7,736.04	4,550.61
`read`	27,408	7,320.15	4,305.97
`lookup`	13,640	3,069.65	1,805.67
`read_inode`	13,641	2,943.22	1,731.30
`readpage`	43,991	477.75	281.03
`sync_page`	20,141	108.73	63.96
`write_inode`	12,107	10.57	6.22
`open`	12,915	1.99	1.17
`release`	12,915	1.29	0.76
`follow_link`	110	0.09	0.05
`write_super`	1	0.00	0.00

Table 3.1: Total count and total delay of VFS operations of Linux 2.4.24 Ext2 for *grep -r* workload. 1 sec. = 1.7 billion CPU cycles.

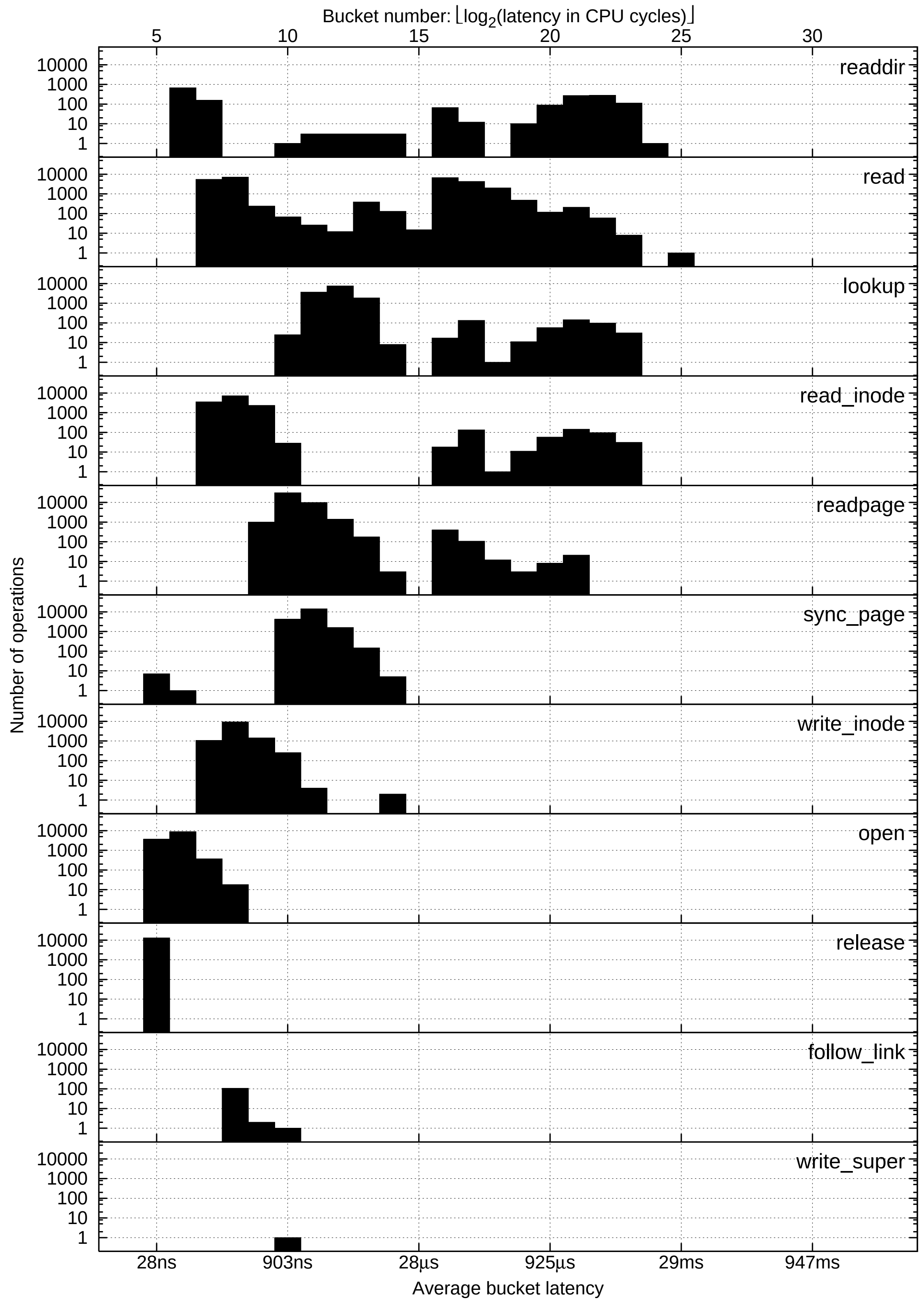


Figure 3.2: Complete profile of Linux 2.4.24 Ext2 under the *grep -r* workload. Operations are sorted from top to bottom by their total latency.

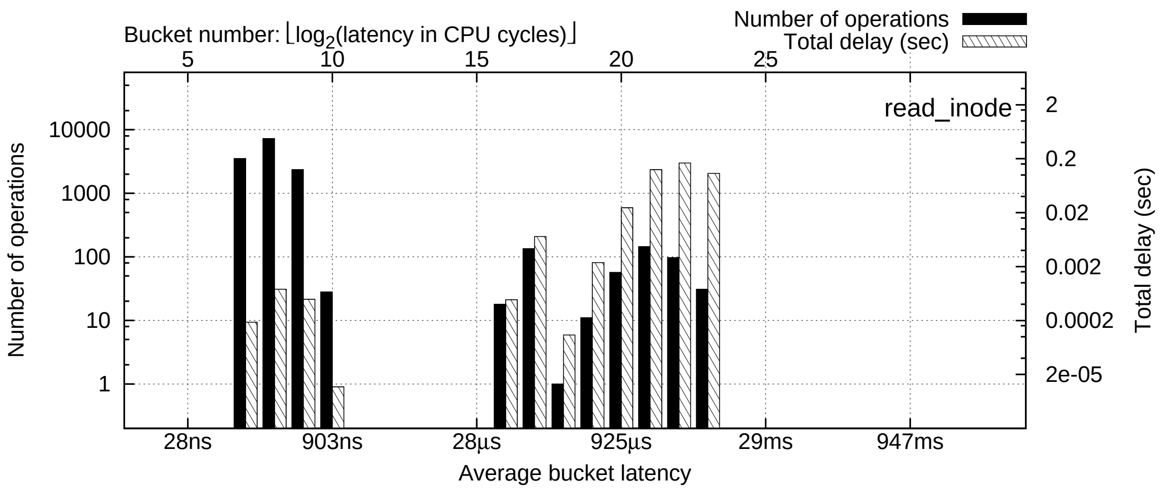


Figure 3.3: Tri-modal Profile of the file `read_inode` operation on a Linux 2.4.24 Ext2 file system captured for a single run of *grep -r* on a Linux source tree.

The second technique has two applications. First, it can be used to compare all profiles in a complete set of profiles and select only these profiles that are correlated. Second, it is useful to compare two different complete sets of profiles and select only these pairs that differ substantially; this helps developers narrow down the set of OS operations where optimization efforts may be most beneficial. We have adopted several methods from the fields of statistics and visual analytics [89]. We further describe these methods in Section 3.2 and evaluate them in Section 6.3.

3.1.2 Prior Knowledge Based Analysis

Many OS operations have characteristic times. For example, on our test machines, a context switch takes approximately 56 μs, full stroke disk head seek takes approximately 8 ms, full disk rotation takes approximately 4 ms, the network latency between our test machines is about 112 μs, and the scheduler quantum is about 58 ms. These characteristic times can be easily measured using specially crafted workloads or tools [12, 71]. Therefore, if some of the profiles have a peak close to these times, then we can hypothesize right away that it is related to that corresponding OS activity. For any test setup these and many other characteristic times can be measured in advance by profiling simple workloads that are known to show peaks corresponding to these times. It is common that some peaks analyzed for one workload in one of the OS configurations can be recognized later on new profiles captured in other circumstances.

Figure 3.3 shows a magnified profile of the `read_inode` operation from Figure 3.2. Here we also show the latency of every bucket using the right axis. This tri-modal distribution is defined by the delays needed to read a file's metadata. We will analyze similar profiles in Section 7.2. However, just knowing the characteristic times of our hard disk, we can see that the rightmost peak corresponds to disk head movement or disk platter rotation delays.

3.1.3 Differential Profile Analysis

While analyzing profiles one usually makes a hypothesis about a potential reason for a peak and tries to verify it by capturing a different profile under different conditions. For example, a lock contention should disappear if the workload is generated by a single process. The same technique of comparing profiles captured under modified conditions (including OS code or configuration changes) can be used if no hypothesis can be made. However, this usually requires exploring and comparing more sets of profiles. We have designed procedures to compare two sets of profiles automatically and select only those that differ substantially. Section 3.2 discusses these profile-comparing procedures in more detail.

3.1.4 Layered Profiling

It is usually possible to insert latency-profiling layers inside the OS. Most kernels provide extension mechanisms that allow for the interception and capture of information about internal requests. Figure 3.4 shows such an infrastructure. The inserted layers directly profile requests that are not coming from the user level (*e.g.*, network requests). Comparison of the profiles captured at different levels can make the identification of peaks easier and the measurements more precise. For example, the comparison of user-level and file-system–level profiles helps isolate VFS behavior from the behavior of lower file systems. Note that we do not have to instrument every OS component. For example, we will show later in this section that we can use file system instrumentation to profile the scheduler or timer interrupt processing. Unlike specialized profilers, our profiling method does not require instrumentation mechanisms to be provided by an OS, but can benefit from them if they are available.

Layered profiling can be even extended to the granularity of a single function call. This way, one can capture profiles for many functions even if these functions call each other. To do so, one may instrument function entry and return points manually or, for example, using the `gcc -p` facility. Similarly, many file system operations call each other. For example, the `readdir` operation of Linux 2.6 Ext2 calls `readpage` operation if the directory information is not found in the cache. Therefore, file-system–level profiling can itself be considered layered profiling.

3.1.5 Profiles Sampling

Our profiler is capable of taking successive snapshots by using a new set of buckets to capture latency at predefined intervals of time. In this case we are also comparing one set of profiles against another, as they progress in time. Therefore, the profile is a 4-dimensional view of profiled operations consisting of:

1. Operation
2. Latency
3. Number of operations with this latency
4. Elapsed time interval

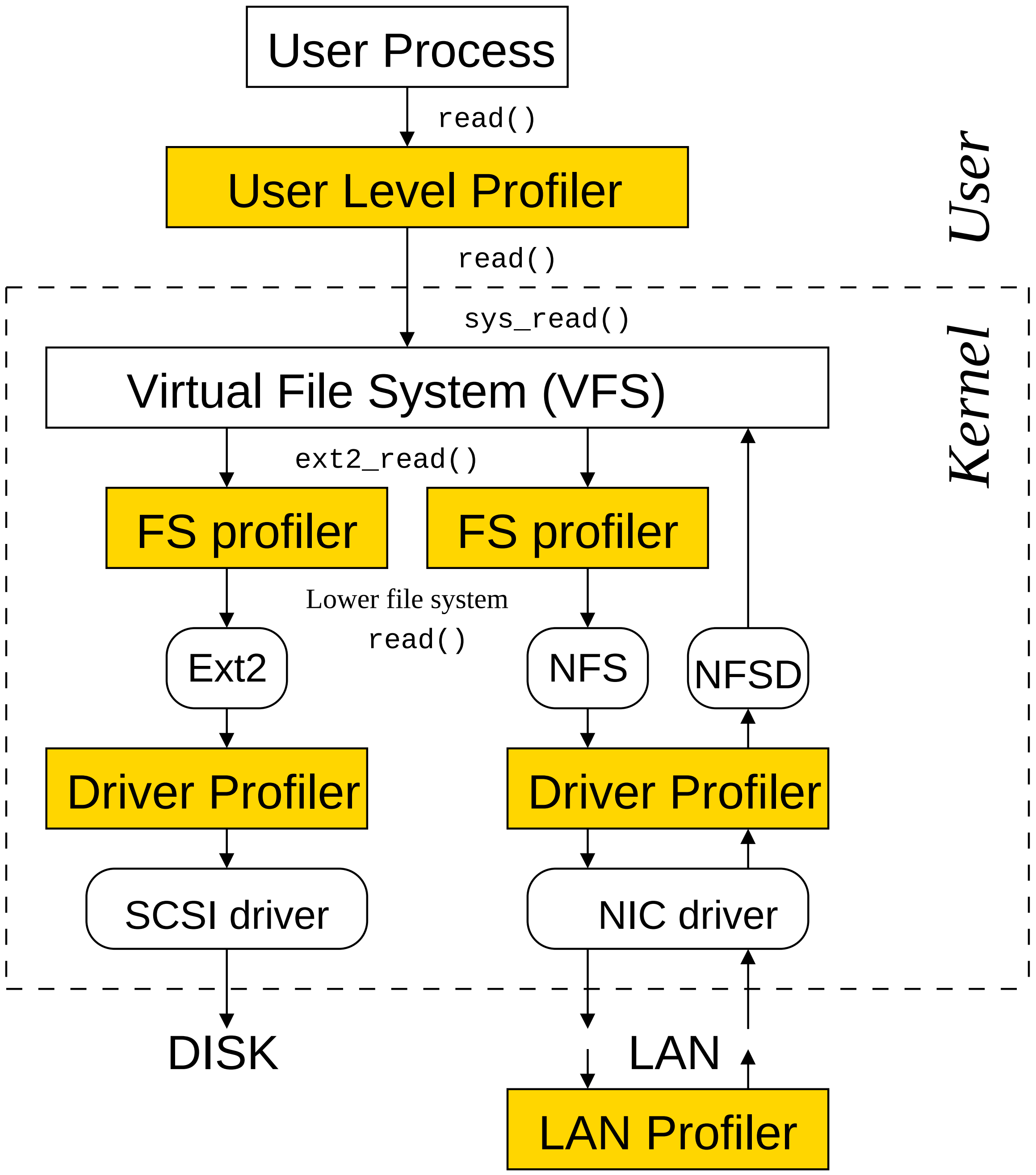


Figure 3.4: Our infrastructure allows profiling at the user, file system, driver, and network levels. Possible profiler locations are shown using the shaded boxes.

Figure 3.5 shows an example 3D view of the `lookup` operation on Ext2 captured while compiling a Linux kernel. The z axis contains the number of operations that fall within a given bucket (the x axis) within a given elapsed time interval (the y axis). Figure 3.6 shows the estimated delay for each bucket on the z axis, which is the number of operations in the b^{th} bucket multiplied by the average bucket latency $\frac{3}{2} \cdot 2^b = 3 \cdot 2^{b-1}$. A small number of invocations in buckets 22–25 (1 ms–30 ms) are responsible for a large portion of the operation's overall delay.

Profile sampling is useful to observe periodic interactions or analyze profiles generated by non-monotonic workload generators.

3.1.6 Direct Profiles and Values Correlation

If layered profiling is used, it is possible to correlate peaks on the profiles directly with the internal OS state. In particular, we first capture our standard latency profiles. Next, we sort OS requests based on the peak they belong to according to their measured latency. We then store logarithmic profiles of internal OS parameters in separate profiles for separate peaks. In many cases this allows us to correlate the values of internal OS variables directly with the different peaks and thus helps explain them.

We will illustrate all of the above profile analysis methods in Section 7.

3.2 Profiles Analysis Automation

A complete profile of file system activity (*e.g.*, as shown in Figure 3.2) may consist of dozens of profiles of individual operations. While analyzing these profiles we noticed that it is easy for people to spot interesting and unusual patterns. However, we also noticed that there are certain operations which can be automated. For example, it is often useful to select operations that contribute the most to the total latency under a given workload. Moreover, it is often desirable to compare two sets of profiles and select a smaller subset of operations with substantially different latency distributions. For example, a profile of one version of a file system or one type of workload may be compared with a profile of a different file system or the same file system under a different workload. Both of these operations can be performed automatically, leaving a much smaller and simpler set of profiles for manual analysis. We have designed a set of tests to compare profiles using their latencies, the counts of operations, and standard statistical independence tests to compare profiles and calculate their statistical significance.

3.2.1 Individual Profiles Comparison

There are several methods of comparing histograms where only bins with the same index are matched. Some examples are the chi-squared test, the Minkowski form distance [100], histogram intersection, and the Kullback-Leibler/Jeffrey divergence [65]. The drawback of these algorithms is that their results do not take factors such as distance into account because they report the differences between individual bins rather than looking at the overall

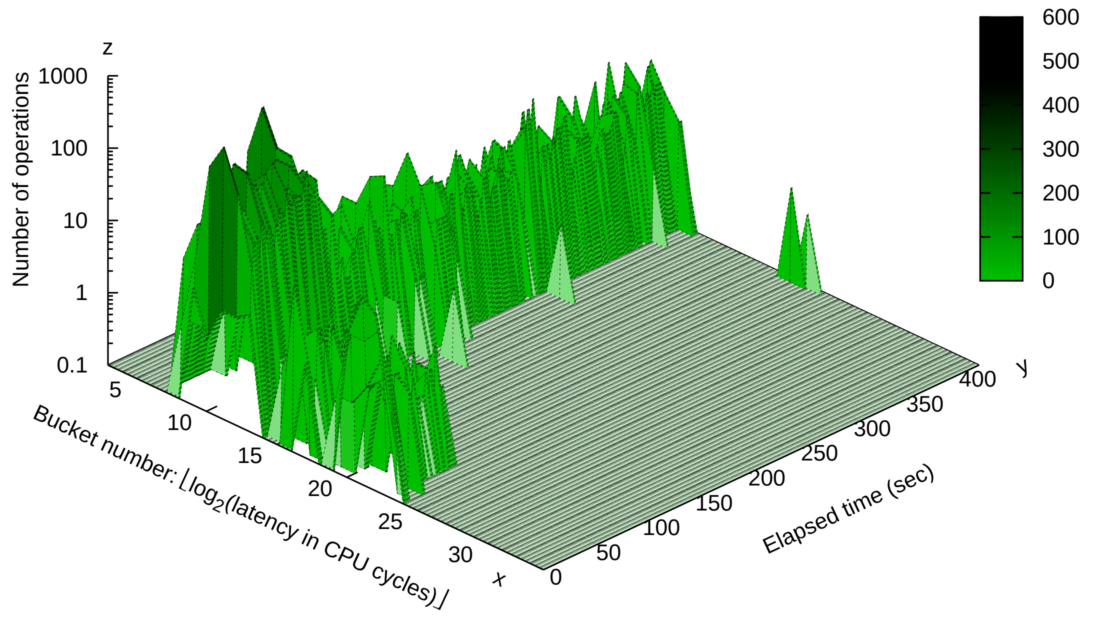


Figure 3.5: Three-dimensional profile of the Ext2 `lookup` operation under the kernel build workload.

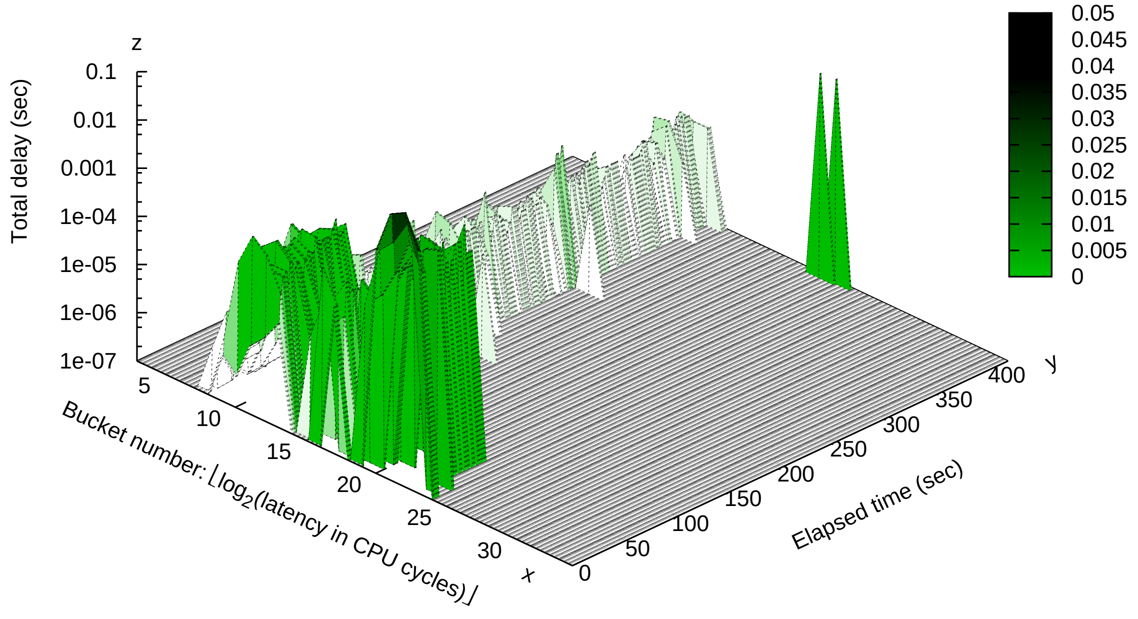


Figure 3.6: Three-dimensional profile of the Ext2 `lookup` operation under the kernel build workload. Buckets contain their expected total latency.

picture. For example, consider a histogram with items only in bucket 1. In a latency profile, shifting the contents of that bucket to the right by ten buckets would be much different than shifting by one (especially since the scale is logarithmic). These algorithms, however, would view both cases as simply removing some items from bucket 1, and adding some items to another bucket, so they would report the same difference for both. We implemented the chi-square test as a representative of this class of algorithms because it is “the accepted test for differences between binned distributions” [83].

Cross-bin comparison methods compare each bin in one histogram to every bin in the other histogram. These methods include the quadratic-form, match, and Kolmogorov-Smirnov distances [20]. Ideally, the algorithm we choose would compare bins of one histogram with only the relevant bins in the other. These algorithms do not make such a distinction, and the extra comparisons result in high false positives. We did not test the Kolmogorov-Smirnov distance because it applies only to continuous distributions.

The Earth Mover’s Distance (EMD) algorithm is a goodness-of-fit test commonly used in data visualization [89]. The idea is to view one histogram as a mass of earth, and the other as holes in the ground; the histograms are normalized so that we have exactly enough earth to fill the hole. The EMD value is the least amount of work needed to fill the holes with earth, where a unit of work is moving one unit by one bin. This algorithm does not suffer from the problems associated with the bin-by-bin and the cross-bin comparison methods, and is specifically designed for visualization. As we show in Section 6.3, EMD indeed outperformed the other algorithms.

3.2.2 Complete Profiles Comparison

We tried to combine some of the above techniques to automate the profiles-selection process even further. We developed an automated profiles analysis tool which performs the following steps:

1. sorts individual profiles of a complete profile according to their total latencies;
2. compares two profiles and calculates their degree of similarity; and
3. performs these steps on two complete sets of profiles to automatically select a small set of profiles for manual analysis.

The third step operates in three phases. First, it ignores any profile pairs that have very similar total latencies, or where the total latency or number of operations is very small, when compared to the rest of the profiles (this threshold is configurable). This step alone greatly reduces the number of profiles a person would need to analyze. In the second phase, our tool examines the changes between bins to identify individual peaks, and reports differences in the number of peaks and their locations. Third, we use one of several methods to rate the difference between the profiles. These included bin-by-bin and cross-bin comparison techniques, and the Earth Mover’s Distance algorithm [89]. We also used two simple comparison methods: the normalized difference of total operations and of total latency.

We will describe the implementation of these and other algorithms for profiles comparison in Section 5.7 and we evaluate them in Section 6.3.

3.3 Multi-Process Profiles

Capturing latency is simple and fast. However, early code-profiling tools rejected latency as a performance metric, because in multitasking OSs a program can be rescheduled at an arbitrary point in time, perturbing the results. We show here that rescheduling can reveal information about internal OS components such as the CPU scheduler, I/O scheduler, hardware interrupts, and periodic OS processes. Also we show conditions in which these components can be profiled or their influence ignored. All the profiles presented in this section were captured in user level (except the Linux part of Figure 3.8).

3.3.1 Forcible Preemption Effects

Execution in the kernel is different from execution in user space. Requests executed in the kernel usually perform limited amounts of CPU activity. Some kernels (***e.g.***, Linux 2.4 and FreeBSD 5.2) are non-preemptive and therefore a process in the kernel cannot be rescheduled, unless it voluntarily yields (gives up) the CPU—for example, during an I/O operation or while waiting on a semaphore. Let us consider a fully preemptive kernel where a process can be rescheduled at any point in time. Let n_b be b's bucket content without preemption enabled and m_b be the content of the same bucket with preemption enabled. Clearly, $\sum n_b = \sum m_b = N$, where N is the total number of profiled requests. A process can be preempted during the profiled time interval only during its t_{cpu} component. Let Q be the quantum of time that a process is allowed to run by the scheduler before it is preempted. A process is never forcibly preempted if it explicitly yields the CPU before running for the duration of Q. This is the case in most of the practical scenarios that involve I/O or waiting on semaphores (***i.e.***, yielding the CPU). Let Y be the probability that a process yields during a request. For example, $Y = 0.01$ if the lock contention on an involved semaphore is 1% or if data is not found in the file system cache 1 out of 100 times. The probability that a process does not yield the CPU during Q cycles is $(1-Y)^{(\frac{Q}{t_{period}})}$, where t_{period} is the average sum of user and system CPU times between requests. If during Q cycles, the process does not yield the CPU, then it will be preempted during the request with probability $\frac{t_{cpu}}{t_{period}}$ and otherwise it will be preempted in the user level. Therefore, the total probability that a process is forcibly preempted while being profiled is:

$$Pr(fp) = \frac{t_{cpu}}{t_{period}} \times (1-Y)^{(\frac{Q}{t_{period}})} \tag{3.5}$$

The expected value of preempted requests for N such Bernoulli trials is $N \times Pr(fp)$. We estimate that we can ignore the preemption effects if $N \times Pr(fp) < 1$. Differential analysis of Equation 3.5 shows that the function rapidly declines if $t_{period} \ll QY$ (we assume $-ln(1-Y) \approx Y$ for $Y < 0.5$). For example, the typical CPU times we observed are in the range of $2^5 - 2^{14}$ CPU cycles. The longest CPU time spent in the kernel that we observed was 2^{18} CPU cycles. (It was the time of the `CreateThread` function under Windows XP while creating a child process.) This is consistent with the earlier observation that file system operations tend to be around 2^{15} CPU cycles long [76]. The value of Q on modern OSs is usually on the order of 2^{26}. Plugging in our typical case numbers for times and 1% yield rate ($Y = 0.01, t_{cpu} = \frac{t_{period}}{2} = 2^{14}, Q = 2^{26}$) we get a small forced preemption probability: 10^{-9}. In the case of the unusually slow `CreateThread` function, most of the function code is semaphore-protected and the contention rate is 10%. Plugging in these numbers ($Y = 0.1, t_{cpu} = \frac{t_{period}}{2} = 2^{18}, Q = 2^{26}$) into Equation 3.5, we get a forced preemption probability of 0.7×10^{-6}. The forced preemption probability declines very rapidly for smaller t_{period} and higher yield rates. For example, it is as low as 10^{-280} if $t_{cpu} = \frac{t_{period}}{2} = 2^{10}$ even for $Y = 0.01$. As illustrated in Figure 3.7, $Pr(fp)$ declines less rapidly for $t_{period} \gg QY$ ($t_{period} - t_{cpu} = 2^{25}$ case). This happens, for example, if the CPU time spent between profiled requests is large.

Let us now consider a process that never yields the CPU ($Y = 0$). The probability for such a process to be preempted during the profiled time interval is $\frac{t_{cpu}}{Q}$. Therefore, the value in the original bucket n_b is decreased by $n_b \times \frac{t_{cpu}}{Q}$. These preempted requests show up in the bucket corresponding to $t_{wait} + t_{cpu} + (P-1) \times Q$, where P is the total number of processes running.

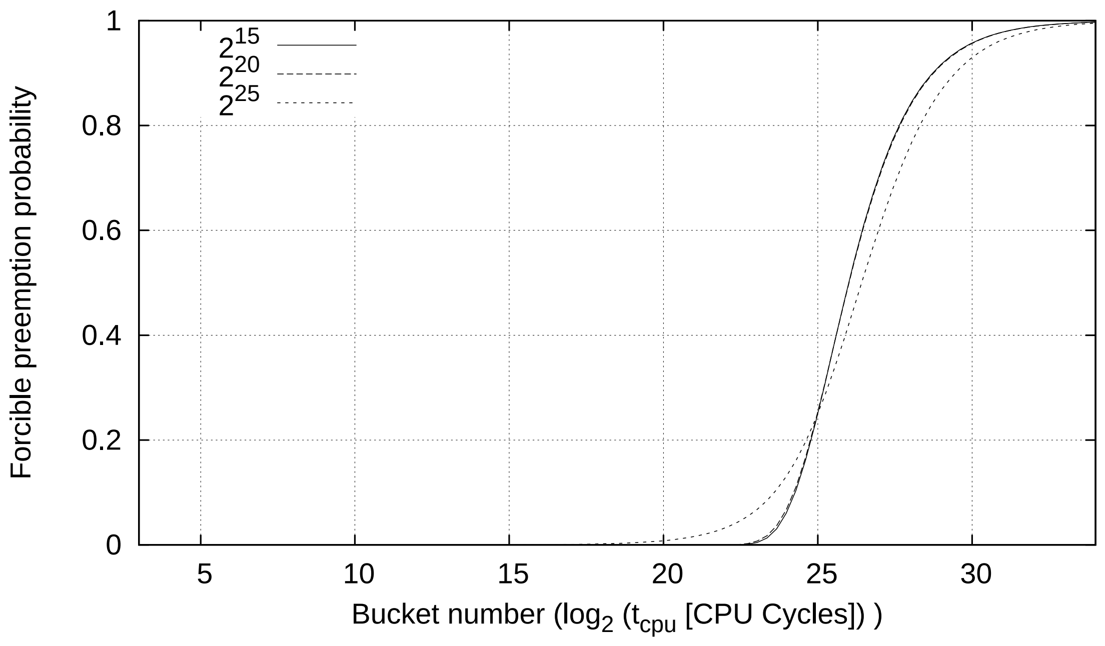
Figure 3.8 shows the profiles of two processes reading zero bytes of data from a file under the Linux 2.6.11 and Windows XP kernels. (Note that Linux profile is captured by the file system profiler so that we could profile the small peaks in the buckets 7–18. The Windows profile is captured in the user level because 0-byte read-requests cannot be profiled by the Windows file-system-level profiler.) Since both processes generate no disk requests, they are preempted after they run for the duration of the scheduler interval Q. The average latency value in bucket b is $\frac{2^b+2^{b+1}}{2} = \frac{3}{2}2^b$. Therefore,

$$m_b = n_b - n_b \frac{\frac{3}{2}2^b}{Q} + \sum_{k \in K(b)} n_k \frac{\frac{3}{2}2^k}{Q} \tag{3.6}$$

where

$$K(b) = \{k : 0 < k, b = \lfloor \log(\frac{3}{2}2^k + (P-1)Q) \rfloor\}$$

is the set of buckets such that the corresponding requests fall into the k^{th} bucket if not preempted and the b^{th} bucket if preempted. Thus, the sum adds up all the values from all the buckets k that go to the b^{th} bucket if the corresponding request execution is preempted. In particular, $\frac{3}{2}2^k + (P-1)Q$ is the average latency value for a preempted request whose latency without preemption corresponds to bucket k. $(P-1)Q$ is the added latency if each process is allowed to run for a scheduling quantum of time Q. The result is calculated with the $\pm 33\%$ precision because the bucket contents can be 33% different from our expected bucket's latency mean value. Using the numbers from the Linux profile shown in Figure 3.8 when captured without preemption, we estimate that $388 \pm 33\%$ (260–516) requests should be preempted and be moved to the 26^{th} bucket. We cannot turn off preemption in Windows, therefore we use the data from the preemptive profile, ignoring the peak in the 26^{th} bucket (that corresponds to Q). We estimate that $2,290 \pm 33\%$ (15,114–30,457) requests should appear in the 26^{th} bucket. The experimental profile with preemption as shown in Figure 3.8 confirmed our conclusions: the 26^{th} bucket contained 278 requests for the Linux profile and 2,337 requests for the Windows profile. Note that in order to measure these numbers we had to generate $N = 2 \times 10^8$ requests, which is many orders of magnitude higher than



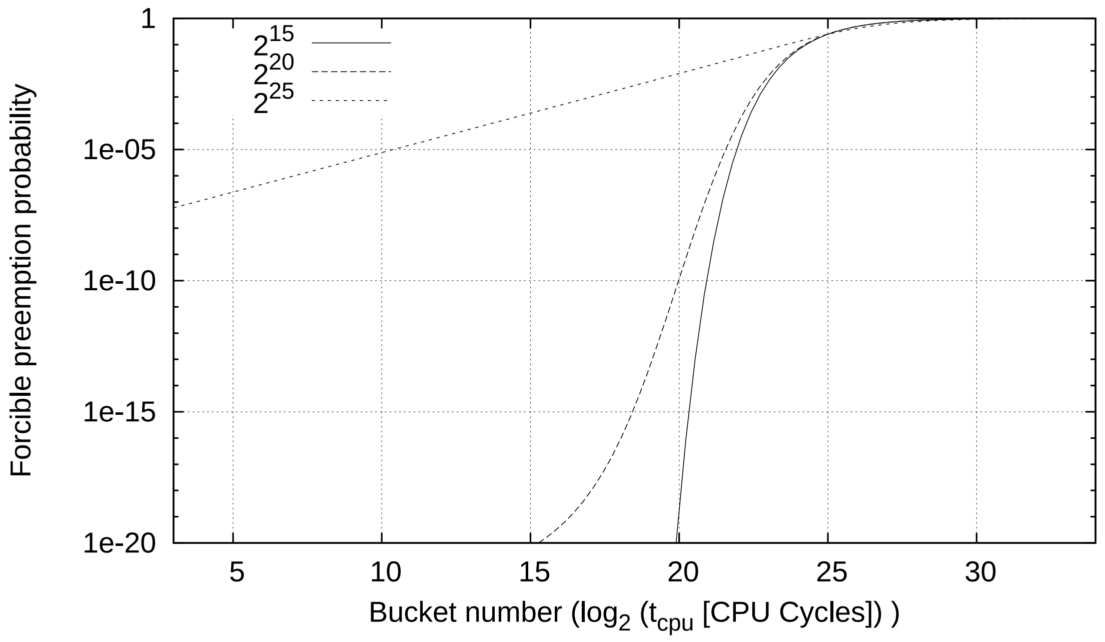


Figure 3.7: Forcible preemption probability as a function of the bucket number, for $t_{period} - t_{cpu}$ equal to 2^{15}, 2^{20}, and 2^{25} ($Y = 0.5$, $Q = 2^{26}$). Linear Y scale (top) and logarithmic scale (bottom). Lines for $t_{period} - t_{cpu}$ equal to 2^{15} and 2^{25} are indistinguishable on the top figure.

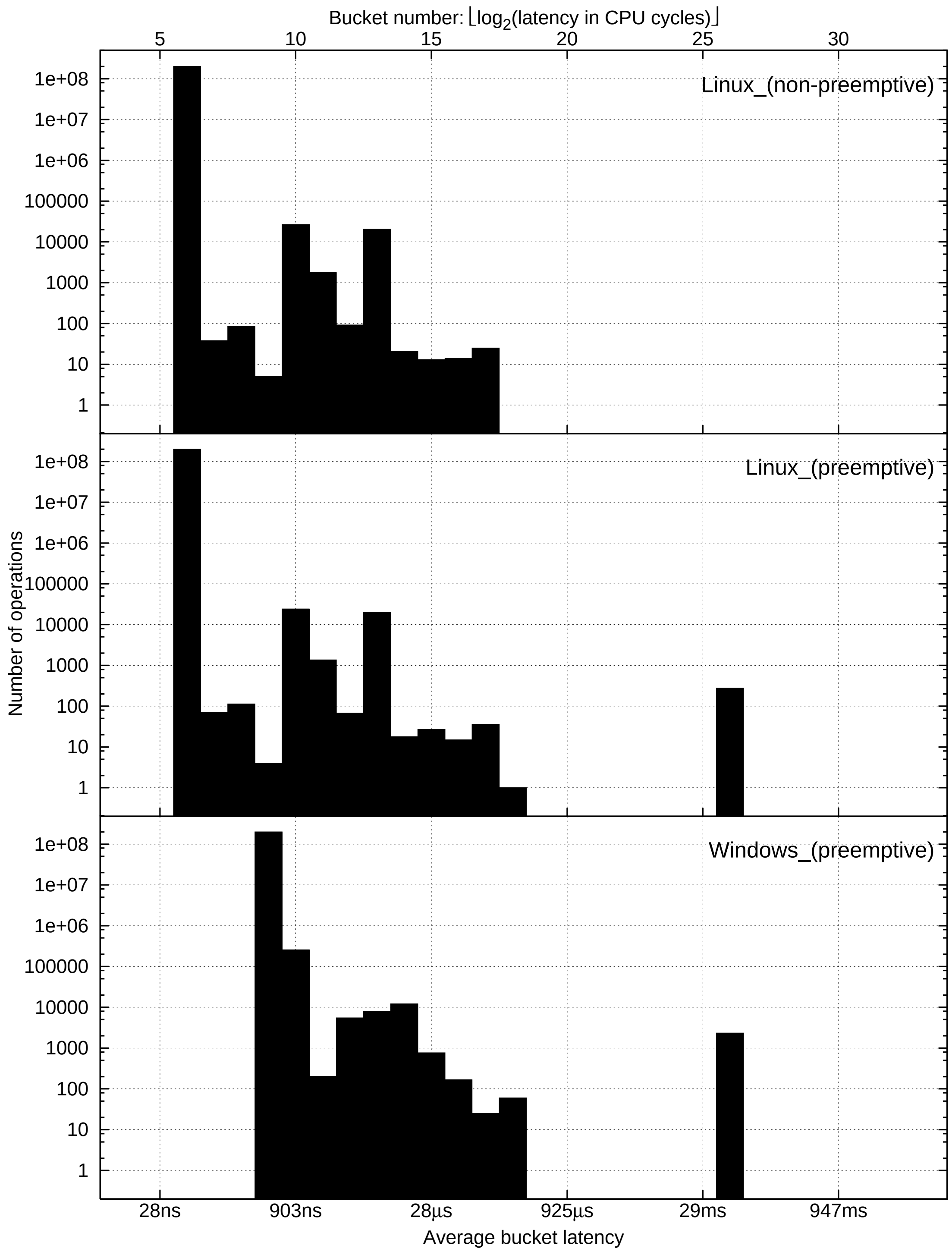


Figure 3.8: Profile of a read operation that reads zero bytes of data using a Linux 2.6.11 kernel compiled with in-kernel preemption enabled and the same kernel with preemption disabled and the same on Windows XP. Note the high number of operations (2×10^8) necessary to observe the preemption effects.

we used in all other profiles. This is because with smaller N, the expected number of preempted requests is much smaller than 1.

Let T_{cpu} be the total CPU time spent during the run. (The Sum of the total system and user times.) We call a workload *CPU-intensive* if $T_{cpu} \gg NQY$ and *yield-intensive* if $T_{cpu} \ll NQY$. All the values involved in this equation can be derived from the profile. Note that we assume that the OS is a gray box but the requests generator (a user-level program) is not; we assume that its user time component can be measured using a tool like `time` on Linux. Our analysis suggests that preemption effects can be completely ignored if the profiled portion of the CPU time and the total number of the profiled requests are small. In practice, this is usually the case. Even during our longest experiment (Linux kernel compilation), N was below 10^5. If the number of profiled requests or the profiled CPU times are large, then one needs to estimate the impact of preemption effects and either issue a smaller number of requests or analyze and possibly ignore the preemption effects.

On the other hand, one needs to generate $N \gg Pr(fb)$ requests (usually 10^8 or more) to measure the preemption effects like we did in Figure 3.8. Note that it is possible only for the CPU-intensive case because in the yield-intensive case, $Pr(fp)$ is astronomically small and it is not feasible to run the corresponding number of requests. Profiles that contain a large number of requests show information about low-frequency events such as hardware interrupts or background kernel threads even if these background events perform a minimal amount of activity. For example, on the Linux profile shown in Figure 3.8, the total duration of the profiling process divided by the number of elements in bucket 13 is equal to 4 ms. This suggests that this peak corresponds to the timer interrupt processing. Higher resolution profiles may help analyze these peaks.

To better observe background OS activity we ran a special workload: we ran one and two processes in a busy loop. Such workloads measure the latency of our profiler only and, sometimes, the latency of periodic OS processes. Figure 3.9 shows the profiles of 1 and 2 processes that run in a busy loop. The top profile was captured with our default resolution ($r = 1$) and the bottom one with double resolution ($r = 2$ in Equation 3.4). As we can see, the higher resolution allowed us to resolve peaks that were too close to each other. For example, we can see that a wide peak in buckets 15–17 of the top profile actually consists of two smaller peaks.

3.3.2 Wait Times at High CPU Loads

We normally assume that the wait time (t_{wait}) is defined by particular events such as I/O or a wait on a semaphore. However, if the CPU is still busy after the t_{wait} time, servicing another process, then the request's latency will be longer than the original latency of Equation 3.1. Such a profile will still be correct because it will contain information about the affected t_{wait}. However, it will be harder to analyze—it will be shifted to the right; because the buckets are logarithmic, multiple peaks can become indistinguishable. Fortunately, this can happen only if the sum of the CPU times of all other processes exceeds the profiled t_{wait}. The average CPU time between requests that have $t_{wait} > 0$ is $\frac{T_{cpu}}{NY}$. Therefore, if $t_{wait} \gg \frac{T_{cpu}}{NY}$ then there is no influence of other processes on the t_{wait} time of the profiled process.

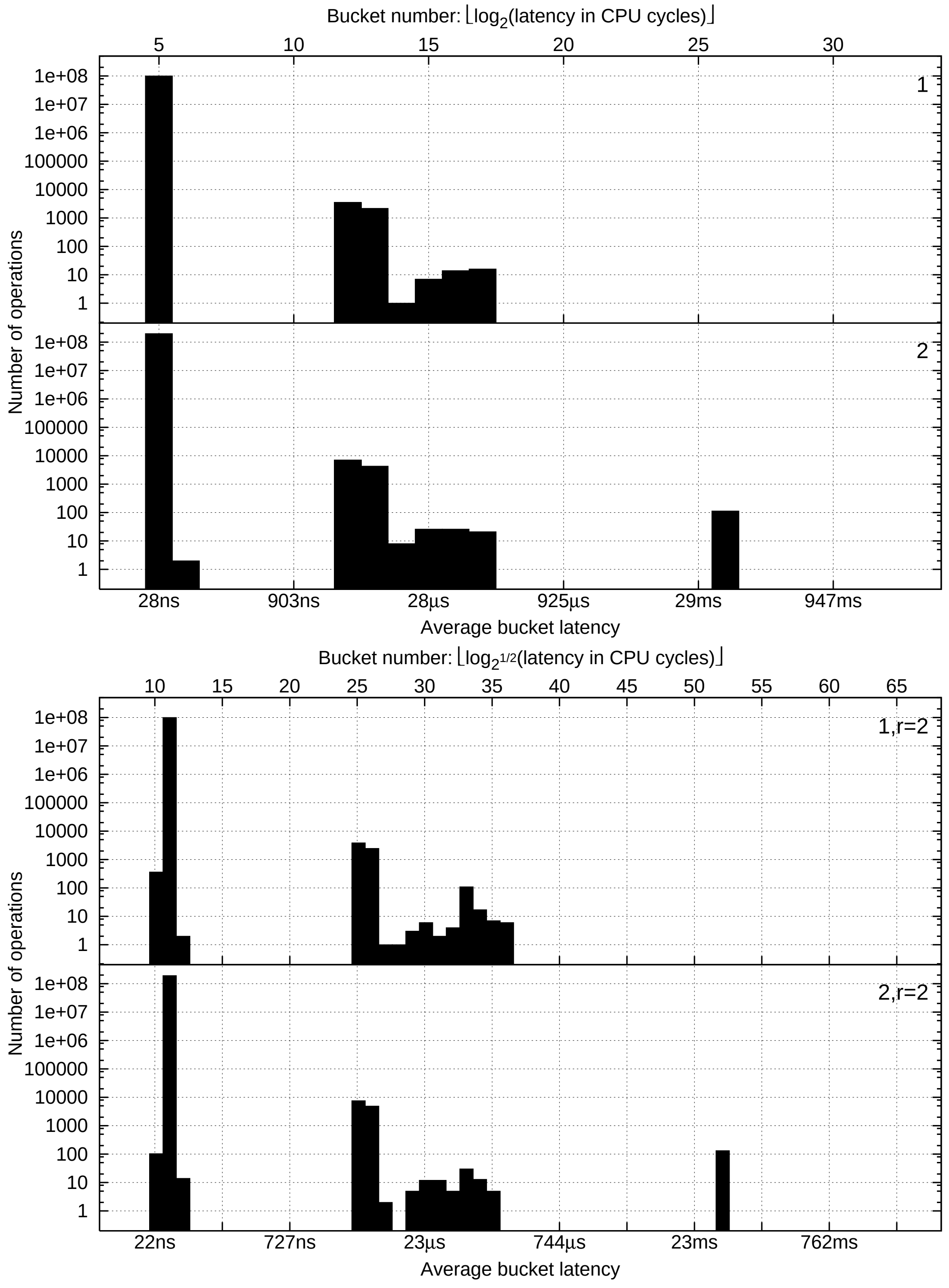


Figure 3.9: Profiles of 1 and 2 processes running in a busy loop and measuring profiler's latency captured with our default resolution ($r = 1$, top) and double resolution ($r = 2$, bottom).

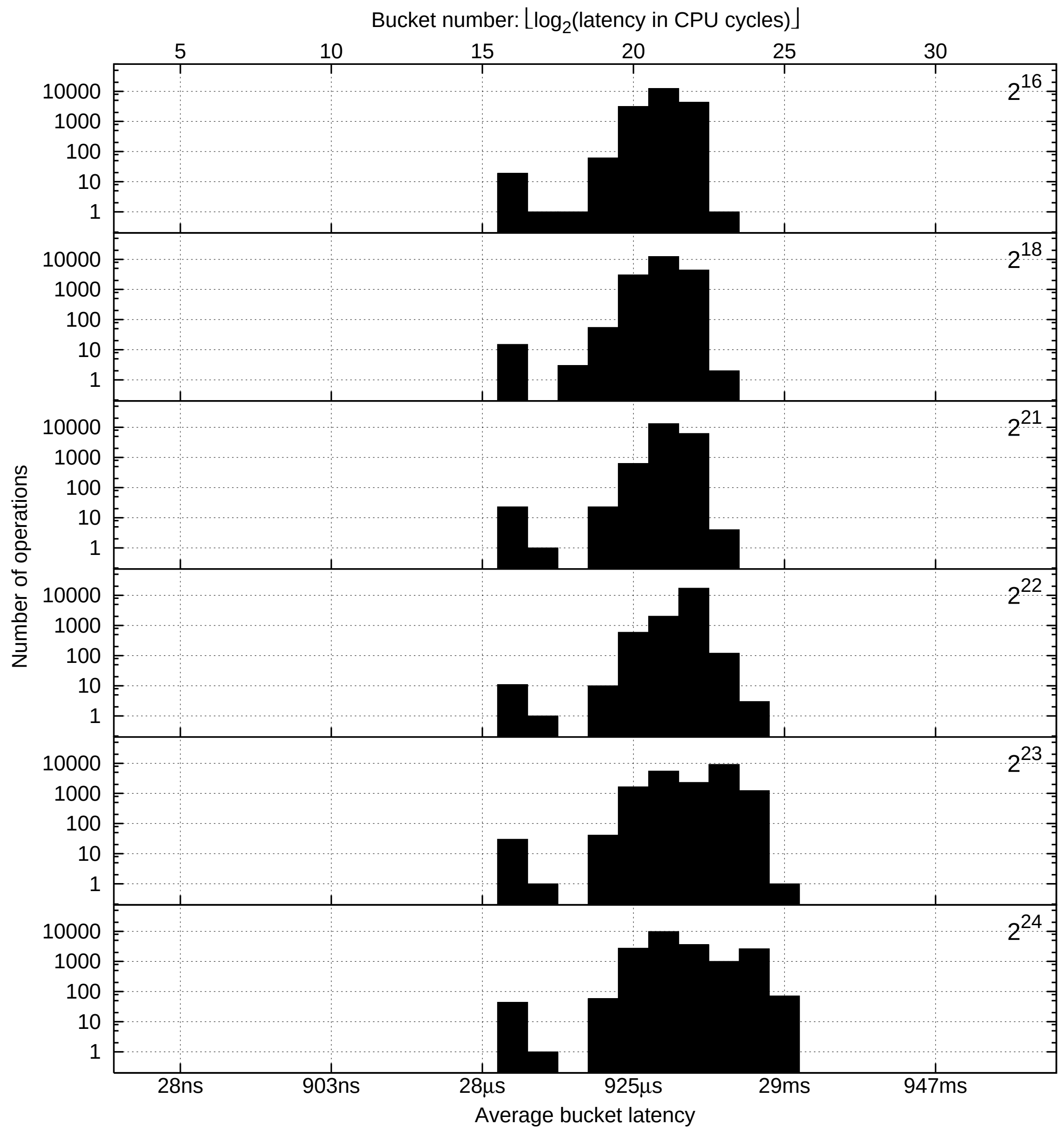


Figure 3.10: Profile of two processes that read files on separate hard drives using direct I/O under Windows XP with varying amount of CPU activity between reads (2^{16}, 2^{18}, 2^{21}, 2^{22}, 2^{23}, and 2^{24} CPU cycles).

Figure 3.10 shows a set of profiles captured under Windows XP. Two processes were randomly reading large files using direct I/O on separate hard drives and performed a different amount of CPU activity between reads. We used direct I/O to eliminate readahead effects and therefore simplify the profile analysis. Here, the total number of requests $N = 20,000$. The six profiles are captured at average CPU times between requests $\frac{T_{cpu}}{NY}$ equal to 2^{16}, 2^{18}, 2^{21}, 2^{22}, 2^{23}, and 2^{24}. As we can see, the profile is affected only if t_{wait} is less than $\frac{T_{cpu}}{NY}$. Otherwise, an extra peak appears on the profile that corresponds to $\frac{T_{cpu}}{NY}$ time. We can see on the profiles in Figure 3.10 how this extra peak first splits off from the peak in the 16^{th} bucket and later (as we increase the CPU time between reads) from the peak in the buckets 19–23.

This effect can be used to analyze workloads with overlapping peaks. Thus, it is possible to distinguish peaks without a t_{wait} component from the peaks that have a t_{wait} component by varying the user-level CPU time. Also, t_{wait} peaks can be moved to the right to avoid overlapping peaks. (In that case the peak latency should be calculated according to its original position.)

Influence of processes without wait times. The influence of CPU-only processes on processes with the wait times can be discarded or easily analyzed. Modern schedulers automatically decrease process priority if it consumes CPU cycles without making I/O requests, and therefore such processes are unlikely to significantly affect the profiles of I/O-active processes. Figure 3.11 shows the profiles of a Linux process that was sequentially reading a large file with and without a background CPU-intensive process running concurrently. We can see that the I/O-intensive process is barely affected by the CPU-intensive one. In particular, out of 10,000 I/O requests, only 3 were rescheduled under Linux and 33 under Windows. The formed peaks (buckets 24–26) have a well-defined location that corresponds to $(P - 1) \times Q$.

Interestingly, the Windows and the Linux schedulers exhibit different behavior: the Linux scheduler penalized the CPU-only process by decreasing its Q so that we see a peak in buckets 24–25 instead of 25–26.

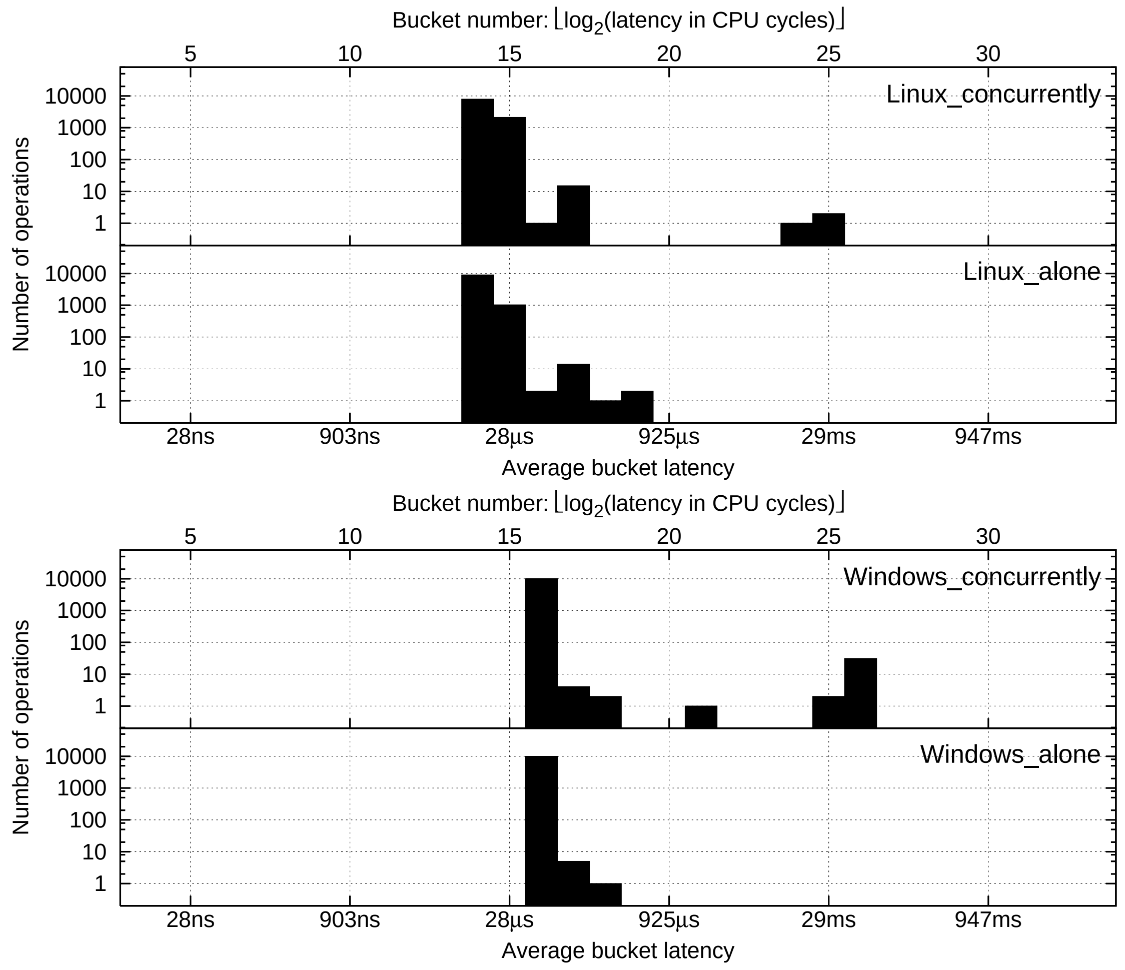


Figure 3.11: Profile of an I/O-active process that sequentially reads data using direct I/O, run alone and concurrently with another CPU-only process. Ext3 running on Linux 2.6.11 (top) and NTFS running on Windows XP (bottom).

3.4 Multi-CPU Profiles

Profiles captured on multi-CPU systems contain information that is even harder to capture using existing tools. For example, multi-CPU profiles include information about spinlocks. In general, latency profiling on multi-CPU systems is similar to profiling on the uniprocessor (UP) systems. However, there are several issues that require special attention.

3.4.1 Time Synchronization

We use CPU counters to measure latency. However, CPU counters on different CPUs have different values. Therefore, the measured latency may be equal to the true latency with addition of the CPU clock counters difference if a process is rescheduled and put on a different CPU while being profiled. (In that case we calculate the latency as the difference of clock counters on different CPUs.) As in the case of forcible preemption, rescheduling outside of the profiling interval does not affect our profiles. Modern schedulers attempt to schedule the same process on the same CPU it was running before, if possible, to avoid purging CPU caches. However, there is a more significant reason why we can ignore this problem.

We use logarithmic filtering of latency. Therefore, according to Equation 3.3 we can ignore the difference between CPU counters δ if $\delta \ll t_{max}$. Also, a process or a thread measures latency on different CPUs only if the process or the thread is rescheduled. Therefore, $t_{max} \geq t_{scheduling}$. Even for modern CPUs, $t_{scheduling}$ is typically at least several dozens of microseconds long. However, many CPUs initialize their clock counters to zero at their initialization time and are synchronized with nanoseconds-scale precision. Also, some OSs (*e.g.*, Linux) synchronize clock counters on multiple CPUs with a few microsecond precision. Figure 3.12 shows a special profile generated by two processes on the FreeBSD and Linux SMP systems with two 2.8 GHz Pentium IV CPUs: one of the processes was constantly reading the Pentium clock counter register (TSC) and storing the result in a shared variable; the second process was also reading TSC register, calculating its difference with

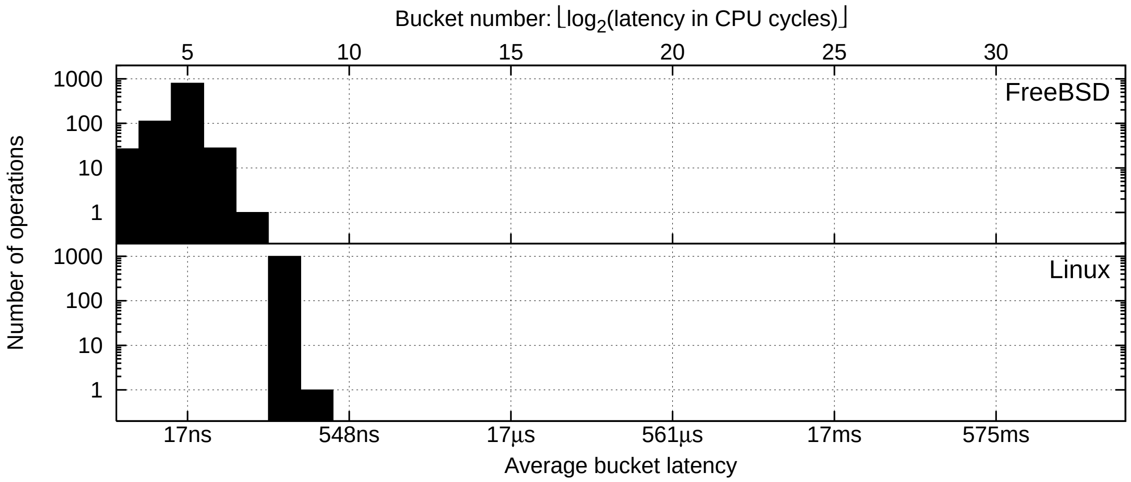


Figure 3.12: The difference in TSC register values between two CPUs under FreeBSD and Linux. (Both CPUs were running at 2.8 GHz.)

the shared variable and updating the corresponding bucket. The system was otherwise idle and the two processes were running on different CPUs. As we can see, in case of FreeBSD, the TSC registers are synchronized with $17ns$ precision. FreeBSD does not attempt to synchronize the clock counters. Linux, on the contrary, attempts to synchronize these counters with at least 4μs precision (2μs difference from the average value for all CPUs). Unfortunately, Linux does it unconditionally and, as we can see in Figure 3.12 (bottom), it actually makes the CPU synchronization worse. Fortunately, in both cases, $\delta \ll t_{scheduling}$ and we can ignore the problem due to the logarithmic filtering.

Now let us consider the case when the CPU counters are not initialized at the CPU initialization time and the OS does not do the synchronization either. First, it is trivial to synchronize the counters with microsecond-scale precision before performing profiling. Second, clock counters are usually 64-bit wide. Therefore, if they are left with random values at the CPU initialization time and no synchronization is performed, with high probability their difference will be on the order of billions of cycles. In that case, a small fraction of profiled events will show up in the most significant (rightmost) bucket on the profiles. (In our implementation, we store events with latencies higher than 2^{35} cycles in the bucket that corresponds to 2^{35} cycles.) Therefore, even in case of no CPU synchronization we can easily identify and discard errors caused by the CPU clock counter difference.

3.4.2 Shared Data Structures

Multiple threads and sometimes multiple processes share and update the same array of buckets in memory. Therefore, it seems that we need to protect this shared data structure with either a spinlock, a semaphore, or at least perform bucket updates atomically (*e.g.*, using the *lock* instruction prefix on Pentium CPUs). This is due to two reasons:

1. Several processes running on different CPUs may read the same bucket value, increment it, and all write the same new value back, thus losing some of the increments.

2. Some write operations are not atomic on some CPUs. For example, writes to the same non-memory–aligned 64-bit–wide variables by different CPUs may result in the situation that part of the variable is updated according to one write and another part is updated according to the other CPU's write.

However, all atomic operations are especially bad for CPU caches in multi-CPU environments and can significantly influence performance. Thus, on older CPUs, atomic operations locked the whole memory bus for all CPUs. Modern CPUs only purge one cache line from all CPUs except the current one. For us this means that if we increment buckets atomically, we will purge many if not all buckets from the CPU caches of all other CPUs upon every bucket update. Fortunately, because of the statistical nature of the latency profiling, we can perform all bucket updates non-atomically as follows:

1. The probability of the occurrence of the first problem described above can be estimated using the classical birthday paradox problem where a "birthday" is the event that two CPUs update the same bucket and the "year" is $\frac{t_{bucket}}{t_{update}}$ "days long". t_{update} is the time necessary to update the bucket (a memory write), and t_{bucket} is the average time between requests that result in writing to the *same* bucket. Thus, a coarse

approximation of the probability that two processes concurrently update the same bucket is:

$$1 - e^{-\frac{N_{cpu}^2}{2} \times \frac{t_{update}}{t_{bucket}}} \quad (3.7)$$

where N_{cpu} is the number of CPUs. This probability is illustrated in Figure 3.13 for 2, 4, 8, 16, 32, 64, 128, and 256 CPUs. t_{bucket} is at least 200 CPU cycles long due to our profiler overhead itself, whereas t_{update} is only several cycles long. Therefore, for a dual CPU system, the probability of two concurrent bucket updates is $<$ 1% even if our profiler does not measure any real latency and runs in a busy loop. Figure 3.14 shows such a profile captured on a dual Pentium IV machine running Linux. Two threads where updating the same set of buckets in a busy loop. Out of 10,000,000 total updates there are 95,116 total lost updates (0.95% $<$ 1%). For real workloads, t_{bucket} is much bigger. For example, $t_{bucket} \approx 2^{22}$ CPU cycles under the `grep -r` workload. Therefore, the probability of two concurrent writes to the same bucket under a grep-like multi-threaded workload is $<$ 1% even on a system with 256 CPUs. It is important to note here that lost updates are much less probable for the buckets located on the right side of the profiles—buckets we care about the most.

2. In general, the solution to the non-atomic–writes problem is CPU architecture-specific. However, aligned 32-bit–wide writes are atomic on most CPUs. We found 32-bit–wide buckets to be sufficient for all the experiments that we ran. If that is not enough, aligned 64-bit–wide writes are also atomic on some CPUs (including Pentiums). The most important observation, however, is that we always increment buckets by one. Therefore, even if two parts of the bucket can be updated inconsistently the probability of such event is very low.

One more possible complication may be a race condition if the profiles are still updated while reading out the accumulated profiles. To address this, we either read-out the accumulated profiles after the profiling is over, or the profiling time is substantially long. In the latter case, we can consider the read-out time small.

Given all the above, we do not use atomic operations at any time during profiling. This allowed us to decrease the CPU-time overheads several times and avoid CPU cache purging on multi-CPU systems. However, one has to be careful on systems with a large number of CPUs. On these systems each thread may be assigned a separate set of buckets to avoid any lost bucket updates. This increases memory overheads but eliminates all the aforementioned problems completely.

3.4.3 Profile Analysis

While profiling and analyzing collected profiles, it is necessary to understand that there is less contention for CPUs between different processes on multi-CPU systems. For example, forcible preemption and the wait time changes described in Section 3.3 almost never happen if there are only two processes that run on two CPUs.

Figures 3.15 and 3.16 show profiles captured on dual-CPU Linux and FreeBSD systems for 1, 2, and 4 concurrent processes that call the `clone` system call. (One may also compare these profiles with the profile shown in Figure 3.1.) In the FreeBSD case, the

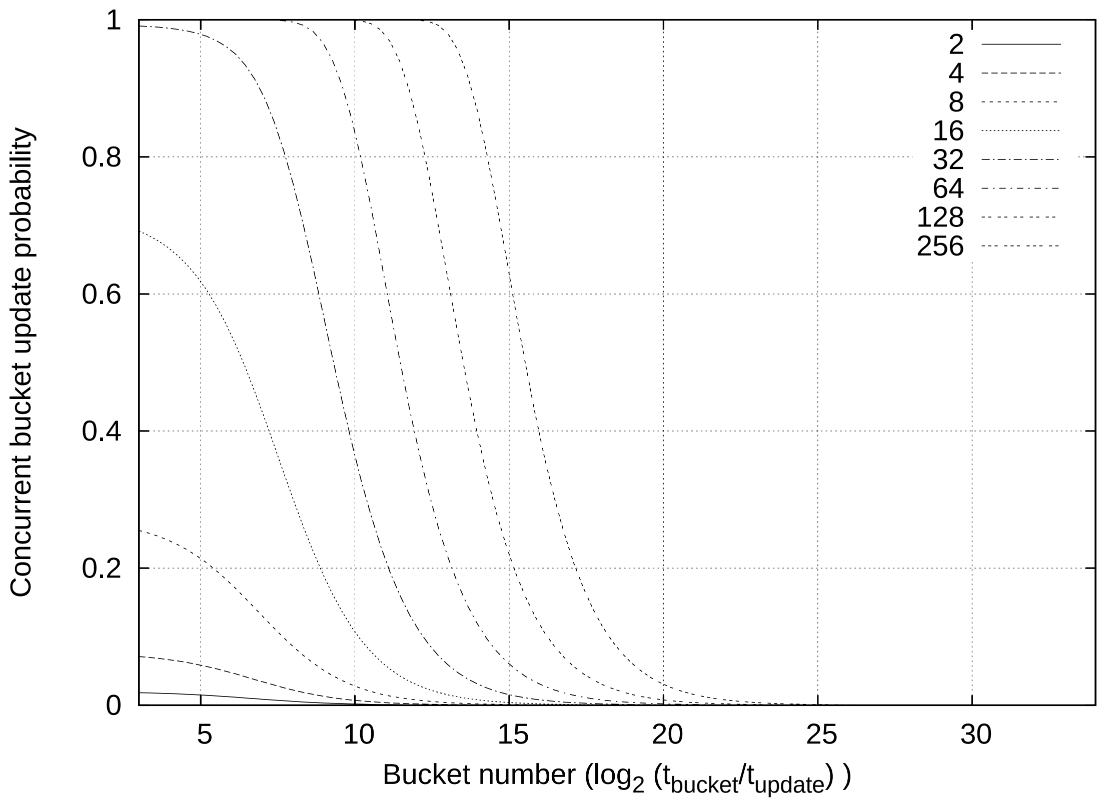


Figure 3.13: The probability of two concurrent writes to the same bucket, estimated using Equation 3.7, for systems with 2, 4, 8, 16, 32, 64, 128, and 256 CPUs. t_{bucket} is at least 200 CPU cycles long.

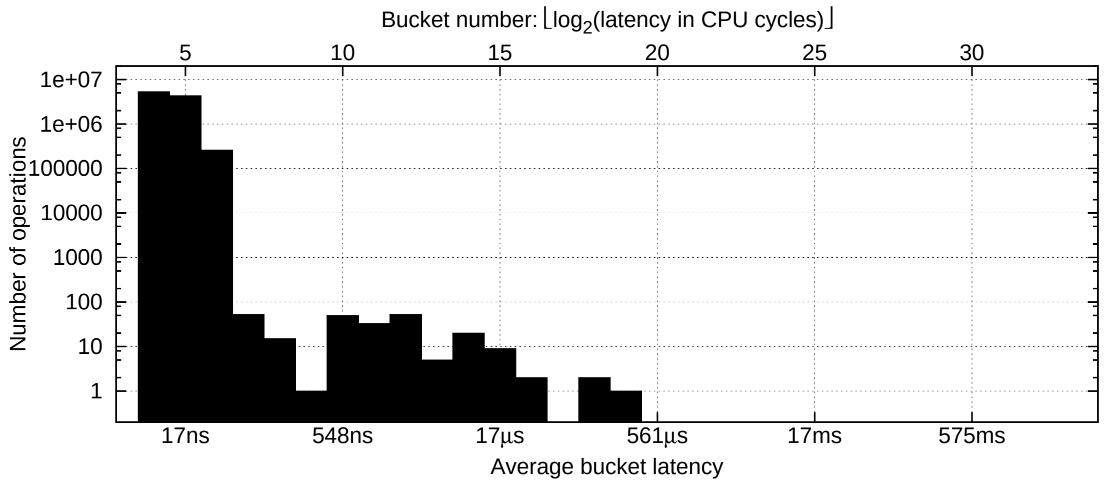
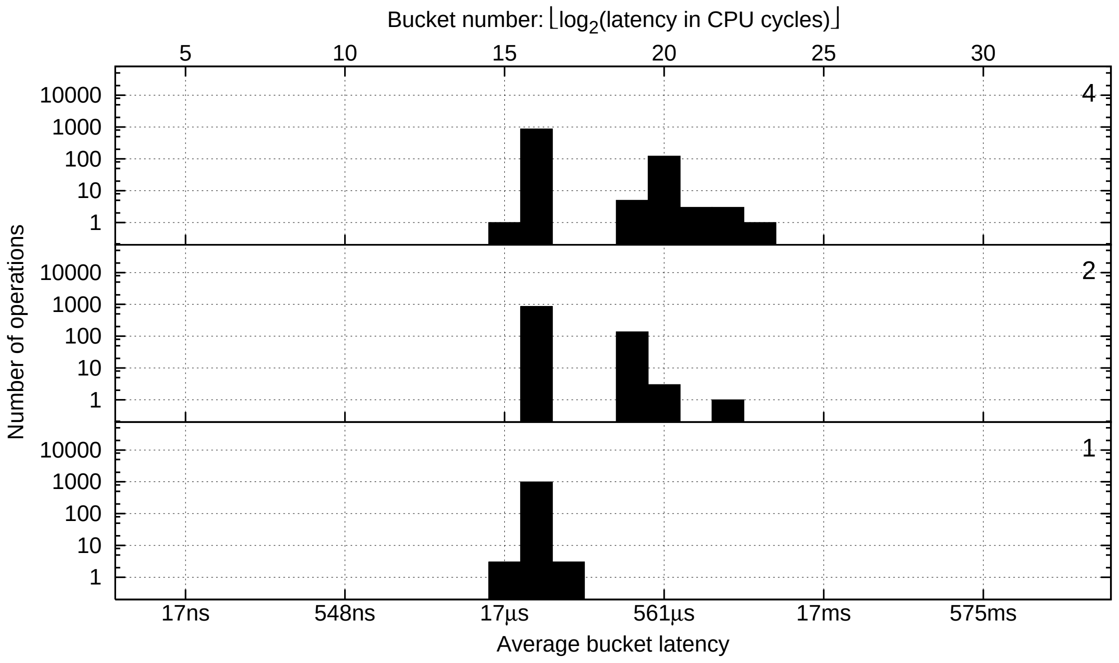


Figure 3.14: Two processes updating the same set of buckets in a loop on a dual-CPU Linux system. The two CPUs were running at a frequency of 2.8 GHz. Buckets 10 and above correspond to the background interrupt processing.



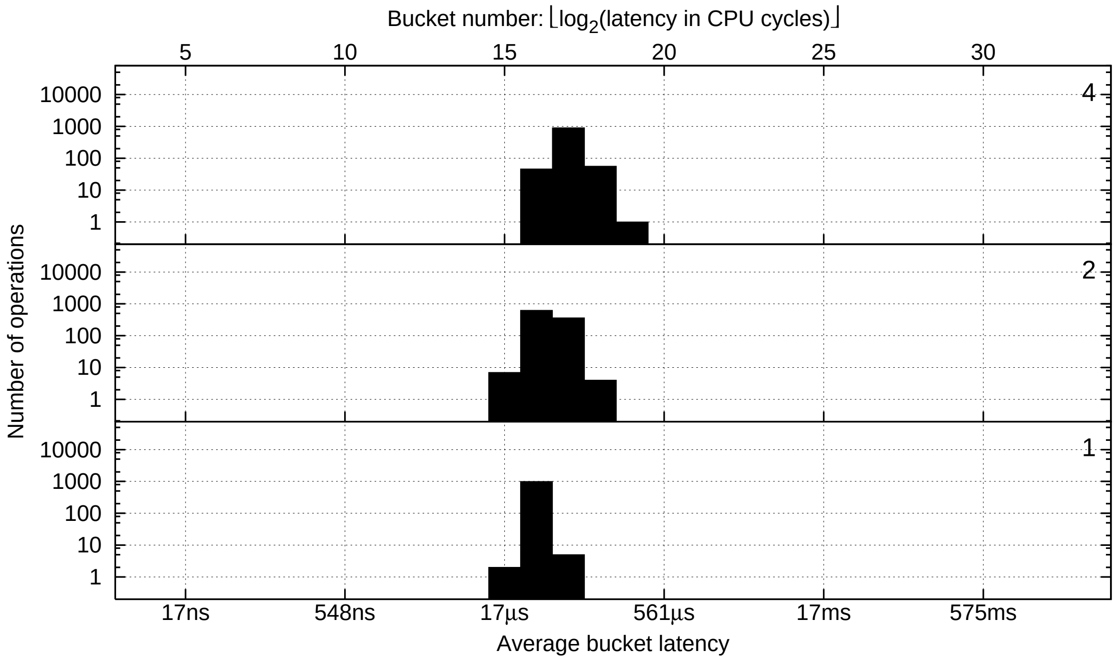


Figure 3.15: `clone` system call concurrently called by 1, 2, and 4 threads on the system running Linux 2.6.11.1 in uniprocessor (UP) mode (top) and SMP mode (bottom). The two CPUs were running at 2.8 GHz.

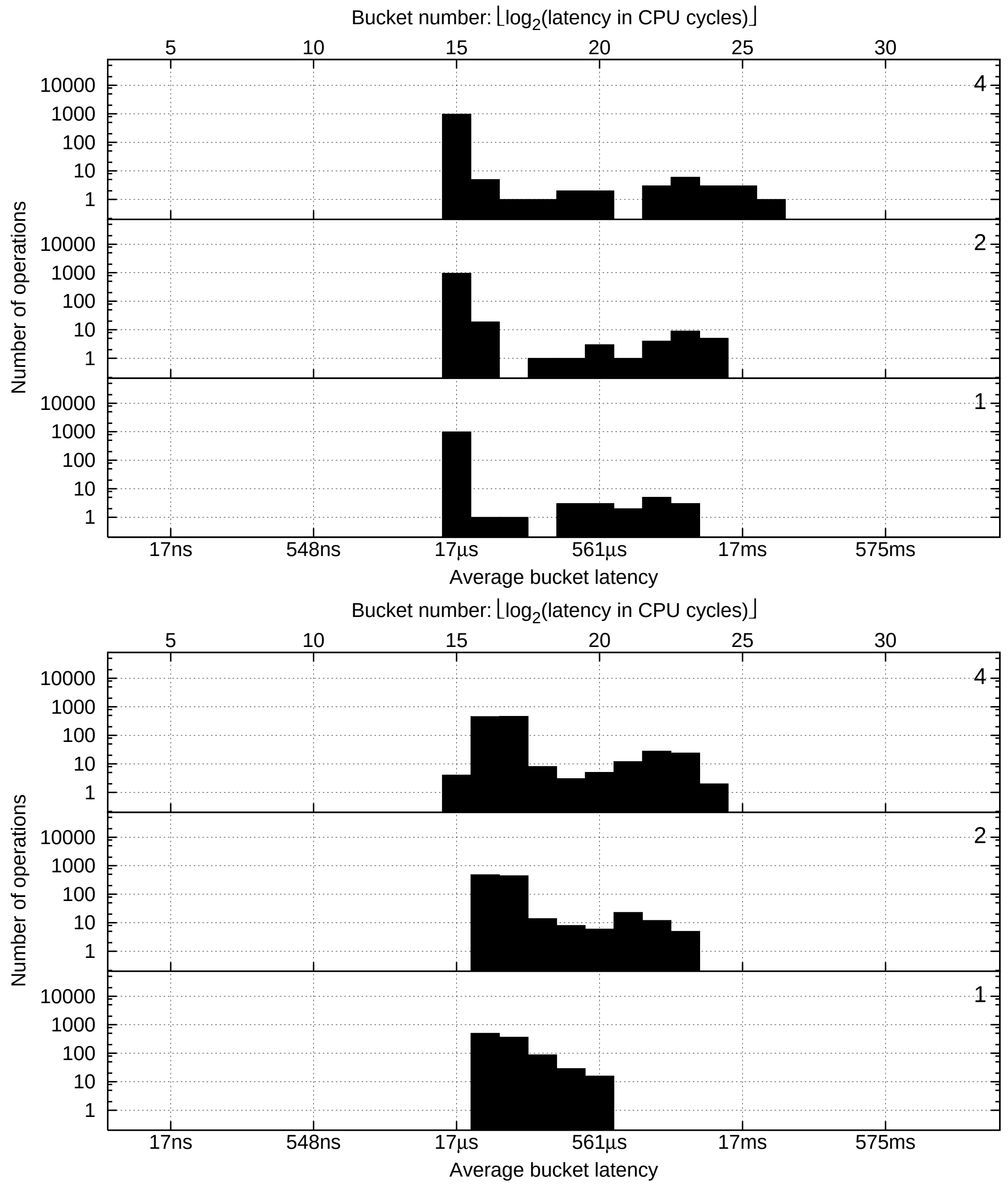


Figure 3.16: `clone` system call concurrently called by 1, 2, and 4 threads on the system running FreeBSD 6.0 in uniprocessor (UP) mode (top) and SMP mode (bottom). The two CPUs were running at 2.8 GHz.

profiles captured even with a single thread differ substantially for FreeBSD compiled with SMP support enabled and disabled. This is because OSs have different locking approaches in uni-CPU and multi-CPU configurations and even the CPU time alone is different due to the differences in the functions involved. For example, `clone` on SMP-enabled FreeBSD takes twice as long even if there is no process contention.

3.5 Profiling in Virtual Environments

Conceptually, OS profiling inside of Virtual Machines (VMs) is not different from ordinary profiling. However, it is important to understand that the host (underlying) OS and the VM itself affect the guest OS's behavior. Even if there is only one virtual machine running on the host, its guest OS's I/O requests will be serviced by the host operating system, which will affect their timing. In addition, the host OS will affect the caches of the shared CPU. Therefore, the benchmarking and profiling results collected in VMs do not necessarily represent the OS behavior running on a system directly. Other VMs running on the same system exacerbate the problem even more.

Nevertheless, there are two situations when profiling in virtual machines is necessary:

1. It is not always possible or safe to benchmark or profile on a real machine directly.

2. VM developers and developers of systems intended to run in or below virtual environments naturally benchmark and profile systems running in virtual environments.

These situations have contradicting requirements. In the first case, it is necessary to minimize the influence of virtualization on the guest OS. In the second case, it is necessary to profile the interactions between the virtual machines as well as their interactions with the host OS. Therefore, we use two different approaches to measure the latencies:

1. We use the guest OS's (apparent) time: for example, in VMware [58] we use the same CPU clock counter read instruction that we normally use. This allows us to minimize the influence of other VMs and the host OS on the profiled system. Unfortunately, this does not provide I/O isolation and depends on the quality of the clock counter virtualization.

2. We use the host OS's (wall clock) time, for example by specifying

   monitor_control.virtual_rdtsc = false

   in VMware's configuration file [103]. This allows us to capture all the mutual interactions between the host OS and the VMs.

Profiling in virtual environments is complicated by the fact that the complexity of the profiled systems increases more than two times. In particular, virtual machines add an extra layer between the host operating system and the guest operating system(s). Thanks to layered profiling it is possible to profile at the guest and host OSs concurrently. This allows us to simplify the profile analysis by attributing profile changes and peaks to the appropriate profiled layers.

Figure 3.17 shows user-mode profiles of the idle loop workload generated by one process and captured with four different configurations:

A. This profile was captured on the host OS running without any virtual machines. Note that this profile is different from the profile shown in Figure 3.8 on page 20. This is because the workload was generated by only one process and because the host hardware and OS were different.

B. This workload was executed and the profile was captured on the host OS. An idle virtual machine was running on the background. We can see that the guest OS influences the host operation even if the guest OS was mostly idle. However, the benchmark's elapsed time increased by merely 2%.

C. This profile was captured in the virtual machine using the guest OS's (apparent) time. We can clearly see three examples of how VMware poorly emulates the TSC CPU register. First, most of the time, the measured latency corresponded to the 9^{th} bucket instead of the original 3^{rd}. We measured the TSC synchronization error between the host CPUs and it corresponded to the same 9^{th} bucket. Therefore, we suspect that the peak in the 9^{th} bucket appears due to the fact that VMware uses different CPUs to read the TSC counter during its emulation. Second, the wide peak in buckets 13–21 is yet another significant artifact of the VMware TSC emulation. Third, the TSC register virtualization is costly and increases the running time of the whole benchmark by 16 times.

D. We again captured the profile in VMware but this time we used host TSC register to measure the latencies. The profile closely resembles the host profile (configuration A). The differences in the buckets 10–21 are due to the longer interrupts processing in VMware. The TSC register emulation in VMware is not just inaccurate but also adds significant overheads. In particular, in configuration D the benchmark's elapsed time was indistinguishable from the elapsed time of the same benchmark in configuration A.

We can conclude that virtual machines profiling is implementation-dependent. Nevertheless, our profiling method is suitable even for such systems.

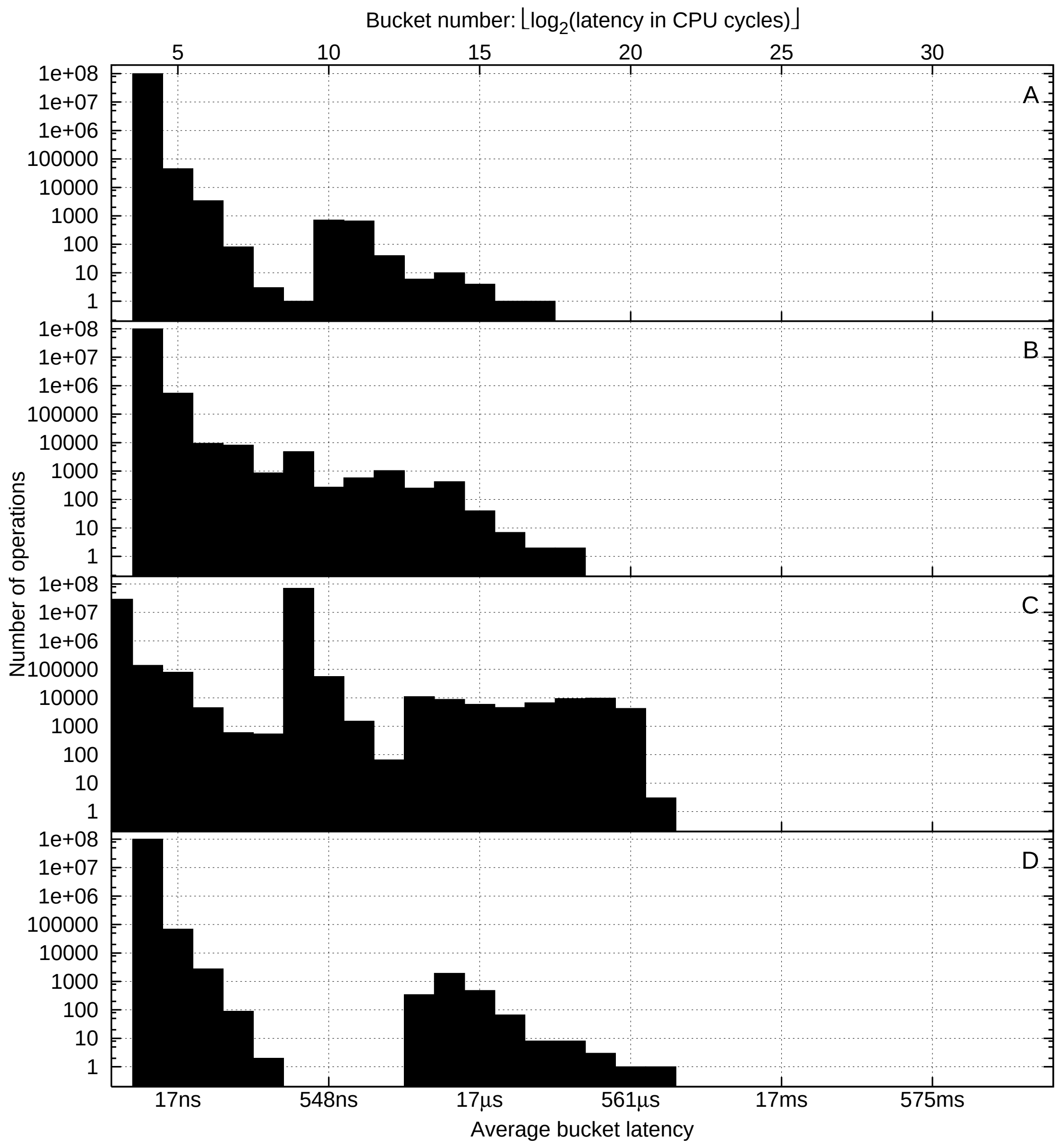


Figure 3.17: Idle loop profiles captured on the host running alone (A), on the host running with one idle VMware on the background (B), running in VMware captured using the guest OS' (apparent) time (C), and running in VMware captured using host OS's time (D). The host was a dual 2.8 GHz SMP system. We used VMware Workstation v.5.5.2. Both the host and the guest OSs were running Linux 2.6.17.

3.6 Method Summary

Our profiling method reveals useful information about many aspects of internal OS behavior. In general, profilers can be used to investigate known performance problems that are seen in benchmarks or during normal use, or to search for bottlenecks actively. We have used our profiler successfully in both ways.

When searching for potential performance issues, we found that a custom workload is useful to generate a profile that highlights an interesting behavior. In general, we start with useful but simple workloads and devise more specific, focused workloads as the need arises. The workload selection is a repetitive-refinement visualization process, but we found that a small number of profiles tended to be enough to reveal highly useful information.

We derived several formulas that allowed us to estimate the effects of preemption. We showed that for typical workloads (moderate CPU use and small number of system calls) preemption effects are negligible. Conversely, a different class of workloads (lots of CPU time and a large number of system calls) can expose preemption effects. This is useful to derive the characteristics of internal OS components such as the CPU scheduler, I/O scheduler, and background interrupts and processes. Such information cannot be easily collected using other methods. While creating the workloads, one should keep in mind that workloads generated by many active processes can have high CPU loads and split the latency peaks associated with the wait time. We have demonstrated that for SMP systems, time synchronization is not a problem and expensive locking is not necessary.

We do not require source code access, which enables us to use gray-box profiling. The resulting profiles show which process or operation causes contention with another operation. For example, the profiles do not show which particular lock or semaphore is causing a slow-down, because that information is specific to a particular OS and therefore conflicts with our portability goal. However, as we show in Section 7, the information we get is sufficient to find out which particular semaphore or lock is problematic if the source code is available.

Because our method can be used entirely outside of the kernel, it does not exact any overall overheads on the kernel; therefore, this results in minimal changes to the internal OS's behavior. Moreover, the small CPU-time overheads observed when profiling inside the kernel are added only on a per-request basis without adding any overhead for each internal event being profiled (*e.g.*, taking a semaphore).

Chapter 4

File System Instrumentation (FoSgen)

Looking at Figure 3.4 on page 13, one can notice that it is possible to use standard and portable API interfaces to intercept at the system call and networking layers. However, it is also desirable to measure the latency inside of the OS: above and below file systems. Manual instrumentation of the drivers below file systems is usually not difficult because there are only a few operation vectors. Unfortunately, manual instrumentation of file systems *is* difficult because the number of VFS operations is large. Also, file systems may significantly change from one version of the OS to another. This makes it necessary to redo the manual instrumentation work for every new OS release.

Incremental addition of code to file systems is a desirable feature not only for profiling but also for adding many other standard and custom features to existing file systems. Examples of such other features include tracing [6], compression [115], encryption [112], secure data deletion [57], data integrity checking [59], and many others.

Stackable file systems can incrementally add functionality to existing and even future file systems [115]. Figure 4.1 shows a *Base0fs* stackable file system that passes through all the file system operations from the *Virtual File System* (VFS) to the lower file system. Unfortunately, stackable file systems add overheads to all file system operations. Direct file system source code instrumentation produces file systems that run more efficiently, because only the necessary operations are instrumented and the compiler has the flexibility to optimize the code. Unfortunately, such instrumentation requires manual work for every file system and every OS version. We decided to combine the benefits of both approaches.

We have designed an automatic file-system instrumentation system we called *FoSgen*. It automatically instruments a subset of file system operations directly in the source code. If a file system's source code is unavailable, then FoSgen can instrument the Base0fs file system. Base0fs can then be mounted over a file system whose source code is not available, adding a small overhead but also providing the new functionality.

FoSgen was designed with two assumptions in mind: (1) At a high level of abstraction, all file systems under different OSs perform similar operations and deal with the same abstract objects: superblock, files, directories, and links; (2) Figure 4.2 shows that FoSgen processes both the target file system and the new extension (written in the FiST language [115]). Based on the information contained in both, FoSgen generates a new instrumented file system. The first assumption allows us to design file system extensions at a high level of abstraction and the second assumption allows us to adapt these abstract decisions to a particular file system.

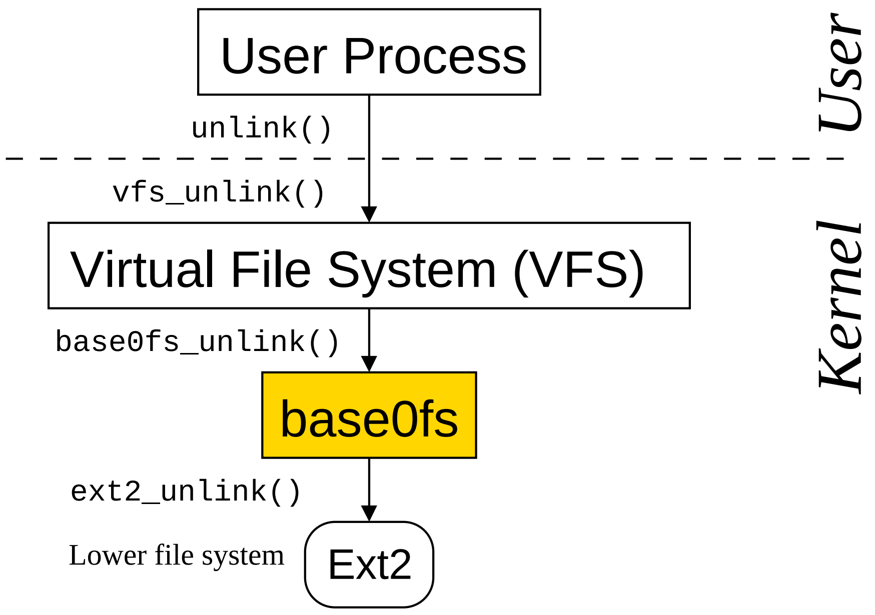


Figure 4.1: The Base0fs stackable file system mounted over Ext2.

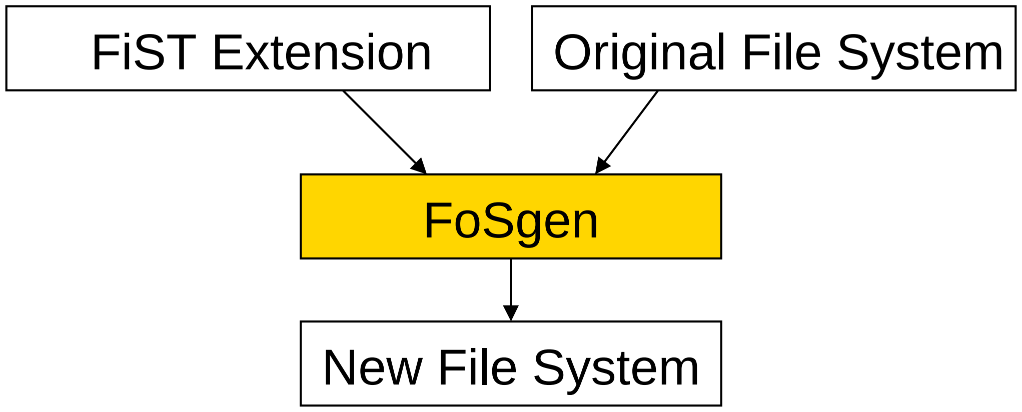


Figure 4.2: FoSgen script operation.

4.1 FiST

The FiST language was designed with similar assumptions as FoSgen [115]. Not surprisingly, we decided to use the FiST language to describe file system extensions. FiST files have a structure similar to the structure of the YACC file format [45]. Every FiST file consists of three sections separated with a `%%` line. The first section contains code and macros added to a generated header file. The middle section describes operations that require instrumentation. The last section describes routines for a separate generated source file. FiST is a C-based language. Because popular OSs are written in C, this allows direct insertion of C code from FiST files into the appropriate locations of file system code.

FoSgen can instrument individual file system operations and insert code at their beginning, before they return, or even replacing the original code. The syntax to specify the instrumentation target is the following:

```
%op:name:where {
    /* instrumentation code goes here */
}
```

where *`name`* is the operation name such as `unlink` or one of the special names: `all` to instrument all operations, `init` to instrument file system module initialization, and `exit` to instrument file system module resource deallocation. *`where`* can be one of the following: `precall`, `postcall`, and `call` to add instrumentation at the beginning, at the method return, and instead of method, respectively.

4.2 VFS Operation Interception

The source code instrumentation process itself is relatively simple and is based on the assumption that the file system's VFS operations are defined within fixed operation vectors. In particular, every VFS operation is a member of one of several data structures (*e.g.*, `struct inode_operations`). These data structures contain a list of operations and their associated functions. For example, Figure 4.3 shows the definition of Ext2's file operations for directories (top) and FreeBSD NFS vnode operations (bottom). The instrumentation script scans every file from the file system source directory for operation vectors, and stores the function names it discovers in a buffer. Next, the script scans the file system source files for the functions found during the previous phase.

The Microsoft Windows VFS is based on message passing. Therefore, most (and often all) file system operations are processed by a single function. As we can see on the top of Figure 4.4, such a function is assigned to members of the `MajorFunction` array and can be found by FoSgen. In turn, the function that is assigned to the `MajorFunction` array assigns an operation completion function by calling *IoSetCompletionRoutine*. FoSgen needs to discover both of these functions in order to measure the latency of requests. This is possible but it is more complicated than instrumenting Linux and FreeBSD file systems. In addition, many Windows file system operations, called `Fast I/O`, are defined similar to Linux and FreeBSD operations. For example, the bottom part of Figure 4.4 shows the declarations of Fast I/O operations for a Windows XP filter driver file system [72].

```
struct file_operations ext2_dir_operations = {
        .llseek =               generic_file_llseek,
        .read =                 generic_read_dir,
        .readdir =              ext2_readdir,
        .ioctl =                ext2_ioctl,
        .fsync =                ext2_sync_file,
};
```

```
struct vop_vector nfs_vnodeops = {
        .vop_default =          &default_vnodeops,
        .vop_access =           nfs_access,
        .vop_advlock =          nfs_advlock,
        .vop_close =            nfs_close,
        .vop_create =           nfs_create,
        .vop_fsync =            nfs_fsync,
        .vop_getattr =          nfs_getattr,
        .vop_getpages =         nfs_getpages,
        .vop_putpages =         nfs_putpages,
        .vop_inactive =         nfs_inactive,
        .vop_lease =            VOP_NULL,
        .vop_link =             nfs_link,
        .vop_lookup =           nfs_lookup,
        .vop_mkdir =            nfs_mkdir,
        .vop_mknod =            nfs_mknod,
        .vop_open =             nfs_open,
        .vop_print =            nfs_print,
        .vop_read =             nfs_read,
        .vop_readdir =          nfs_readdir,
        .vop_readlink =         nfs_readlink,
        .vop_reclaim =          nfs_reclaim,
        .vop_remove =           nfs_remove,
        .vop_rename =           nfs_rename,
        .vop_rmdir =            nfs_rmdir,
        .vop_setattr =          nfs_setattr,
        .vop_strategy =         nfs_strategy,
        .vop_symlink =          nfs_symlink,
        .vop_write =            nfs_write,
};
```

Figure 4.3: Linux Ext2 directory operations (top) and FreeBSD NFS vnode operations (bottom). The Linux kernel exports the `generic_file_llseek` and `generic_read_dir` functions for use by multiple file systems.

```
for( i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++ ) {
    DriverObject->MajorFunction[i] = FilemonDispatch;
}
```

```
FAST_IO_DISPATCH    FastIOHook = {
    sizeof(FAST_IO_DISPATCH),
    FilemonFastIoCheckifPossible,
    FilemonFastIoRead,
    FilemonFastIoWrite,
    FilemonFastIoQueryBasicInfo,
    FilemonFastIoQueryStandardInfo,
    FilemonFastIoLock,
    FilemonFastIoUnlockSingle,
    FilemonFastIoUnlockAll,
    FilemonFastIoUnlockAllByKey,
    FilemonFastIoDeviceControl,
    FilemonFastIoAcquireFile,
    FilemonFastIoReleaseFile,
    FilemonFastIoDetachDevice,
    FilemonFastIoQueryNetworkOpenInfo,
    FilemonFastIoAcquireForModWrite,
    FilemonFastIoMdlRead,
    FilemonFastIoMdlReadComplete,
    FilemonFastIoPrepareMdlWrite,
    FilemonFastIoMdlWriteComplete,
    FilemonFastIoReadCompressed,
    FilemonFastIoWriteCompressed,
    FilemonFastIoMdlReadCompleteCompressed,
    FilemonFastIoMdlWriteCompleteCompressed,
    FilemonFastIoQueryOpen,
    FilemonFastIoReleaseForModWrite,
    FilemonFastIoAcquireForCcFlush,
    FilemonFastIoReleaseForCcFlush
};
```

Figure 4.4: Declaration of normal file system operations (top) and `Fast I/O` operations (bottom) for a Windows XP filter driver file system.

Often, file systems use generic functions exported by the kernel. For example, Ext2 uses the `generic_read_dir` kernel function for its `read` operation and `generic_file_llseek` function for its `llseek` operation as shown in Figure 4.3. FoSgen cannot directly instrument external functions. Therefore, FoSgen creates wrapper functions directly in the file system source files. FoSgen inserts wrapper functions directly before the corresponding operation declarations (*e.g.*, as shown in Figure 4.3) and changes the operation declaration itself to refer to the wrapper function instead of the original one. For example, Figure 4.5 shows how FoSgen transforms the original Ext2 directory operations vector shown in Figure 4.3 (top) in order to instrument its `read` and `llseek` functions. (In this example, wrapper functions are not instrumented.) We use wrapper functions, not inline functions or macros, so that our function has an address for the operations vector to use. Our wrapper functions are later instrumented and are called instead of the original external functions.

By looking at the example shown in Figure 4.5, one can see that FoSgen needs information about the function types, and the number and types of the arguments, in order to create the wrapper functions. That information is available from the Linux header files. Fortunately, file system operation methods are part of the VFS API and do not change much from one Linux kernel version to another. Therefore, we adopted two solutions. First, FoSgen can extract all the necessary information from the kernel header files if they are available. Second, it has a built-in copy of such information for the newest OS versions.

4.3 FiST Support by FoSgen

FoSgen and the FiST language were designed with cross-OS compatibility in mind. Unfortunately, OSs are complex and the creation of a completely OS-independent language to describe OS components is a difficult task. Therefore, FoSgen is a compromise solution between complete OS-independence and implementation simplicity. FoSgen supports basic file system abstractions such as a *vnode*. FoSgen converts abstract vnode objects used in the FiST input file into objects used by target OSs and refers to file properties via a unified vnode object. For example, FoSgen understands that every vnode under Linux is represented by several objects: a file, a dentry, and an inode. Thus, FoSgen uses the dentry object to access the file name, and the inode object to access the file size. At the same time, developers do not need to know about it and can assume that these are the abstract vnode properties. Also, FoSgen supports common OS variables and functions. For example, it converts `fistMalloc`, `fistPrintf`, and several other functions into appropriate OS-specific functions. Similarly, FoSgen converts `fistLastErr`, `%pid`, and some other variables to their appropriate representations on the target OS. This functionality is enough to create many portable file system extensions entirely in the FiST language. However, OSs have more variables, functions, and abstractions than we are able to support. Therefore, we adopted the following model for large extensions: (1) we implement our extension as much as possible using the FiST language, and (2) we implement complex and OS-specific functionality in the form of separate OS-specific kernel modules. Usually, the FiST component describes file system instrumentation details. If we see that we repeatedly implement some functionality as a separate module, we then extend FiST and FoSgen to support it.

```
struct file_operations ext2_dir_operations = {
        .llseek =               generic_file_llseek,
        .read =                 generic_read_dir,
        .readdir =              ext2_readdir,
        .ioctl =                ext2_ioctl,
        .fsync =                ext2_sync_file,
};
```

```
loff_t fosgen_generic_file_llseek(
                  struct file *fosgen_fparam_0,
                  loff_t fosgen_fparam_1,
                  int fosgen_fparam_2)
{
        return generic_file_llseek(
                  fosgen_fparam_0,
                  fosgen_fparam_1,
                  fosgen_fparam_2);
}

ssize_t fosgen_generic_read_dir(
                  struct file *fosgen_fparam_0,
                  char __user *fosgen_fparam_1,
                  size_t fosgen_fparam_2,
                  loff_t *fosgen_fparam_3)
{
        return generic_read_dir(
                  fosgen_fparam_0,
                  fosgen_fparam_1,
                  fosgen_fparam_2,
                  fosgen_fparam_3);
}

struct file_operations ext2_dir_operations = {
        .llseek =               fosgen_generic_file_llseek,
        .read =                 fosgen_generic_read_dir,
        .readdir =              ext2_readdir,
        .ioctl =                ext2_ioctl,
        .fsync =                ext2_sync_file,
};
```

Figure 4.5: Original Ext2 directory operations vector (top) and its FoSgen-transformed version with the wrapper functions (bottom).

4.4 FSprof.fist

We call our file-system–level profiler FSprof. As shown in Table 4.1, FSprof consists of three components:

1. file system hooks to intercept file system operations;
2. the `aggregate_stats` library to measure latency and increase the corresponding bucket values; and
3. a user interface (usually an entry in the `/proc` file system) to read accumulated latency distributions by the users.

Component	**CPU-architecture portability**	**OS portability**
File system operations hooks	yes	yes
`aggregate_stats` library	no	yes
user interface	yes	no

Table 4.1: FSprof components and their portability.

The `aggregate_stats` library includes architecture-dependent code to read the CPU cycle counter. Nevertheless, the architecture-dependent code usually consists of just a single CPU instruction. User interface implementation is usually OS-specific. We have created three types of FSprof extensions with different portability:

- Figure 4.6 shows a minimal FiST file system extension necessary to measure the latency of all file system operations. It inserts function calls to measure the latency at the beginning and at the end of all file system operations. The functions themselves (`fsprof_pre` and `fsprof_post`) are implemented in a separate module. This FSprof extension is portable across all FoSgen-supported OSs.
- An extension common to all systems of the same CPU architecture (*e.g.*, ia64) consists of file system operation hooks and the `aggregate_stats` library. It relies on an external module for user interface functions.
- A complete FiST extension for Linux systems running on i386 and ia64 architectures that requires no extra modules is presented in Appendix A. It can measure the latency of file system operations and output the results via the `/proc` interface.

```
%{
  /*
   * fsprof.fist: collect latency distributions
   *              for all file system operations
   *
   * Copyright (c) 2006 Nikolai Joukov and Erez Zadok
   * Copyright (c) 2006 Stony Brook University
   */

unsigned long long fsprof_pre(int op);
void fsprof_post(int op, unsigned long long init_cycle);

%}

debug off;
license "GPL";

%%

%op:all:precall {
        unsigned long long fsprof_init_cycle =
                fsprof_pre(fistOP_%op);
}

%op:all:postcall {
        fsprof_post(fistOP_%op, fsprof_init_cycle);
}

%%
```

Figure 4.6: A minimal latency profiling FiST extension for FoSgen.

4.5 FoSgen Steps

Figure 4.7 shows an original and the generated `writepage` operation code for the Ext2 file system. We can see that FoSgen created a temporary variable to store and report the internal return value. Let us consider the steps performed by FoSgen during its run.

1. First, FoSgen parses the input FiST file.

2. It replaces simple OS-independent FiST constructions with the appropriate OS-specific functions or variables. For example, it replaces `fistPrintf` with `printk` if the target OS is Linux.

3. FoSgen scans the file system source files for operation declarations (*e.g.*, as shown in Figure 4.3).

4. FoSgen looks for an implementation of the methods that require instrumentation.

5. It adds wrapper functions for the methods that require instrumentation but cannot be found in the file system source files (as shown in Figure 4.5).

6. FoSgen adds the appropriate code at the beginning, at the end, or instead of the original methods, according to the FiST file specification. At this step FoSgen also performs method-specific code transformations. For example, it creates temporary variables like `fist_local_var` in Figure 4.7 and replaces `%op` with the appropriate operation name. Also, it binds method-specific variables with the appropriate method parameters (*e.g.*, it maps the vnode of the file to be unlinked with the `unlink` method's parameters).

7. FoSgen creates `fist.h` files according to the first section of the FiST extension. Also, FoSgen creates `#define` declarations for every operation that was instrumented and an array of strings with names of these operations. After that, FoSgen includes `fist.h` from all the file system source files.

8. Finally, FoSgen generates a `fist.c` file with the appropriate code from the FiST extension and adds the generated `fist.c` to the `Makefile`.

The FoSgen design allows file system developers to concentrate on their new concepts or features instead of the implementation for every file system and OS. Even better, if some OS property changes, developers may not even need to modify the FiST extension.

```
static int ext2_writepage(struct page *page,
                          struct writeback_control *wbc)
{
    return block_write_full_page(page,
                                 ext2_get_block,
                                 wbc);
}
```

```
static int ext2_writepage(struct page *page,
                          struct writeback_control *wbc)
{
    unsigned long long fsprof_init_cycle =
                fsprof_pre(fistOP_writepage);
    {
       int fist_local_var = block_write_full_page(page,
                                 ext2_get_block,
                                 wbc);
       fsprof_post(fistOP_writepage, fsprof_init_cycle);
       return fist_local_var;
    }
}
```

Figure 4.7: An original (top) and an FSprof-instrumented (bottom) `writepage` operation of the Linux Ext2 file system.

Chapter 5

Implementation

We designed a fast and portable `aggregate_stats` library that sorts and stores latency statistics in logarithmic buckets. Using that library, we created user-level, file-system-level, and driver-level profilers for Linux, FreeBSD, and Windows, as shown in Figure 3.4.

Instrumenting Linux and FreeBSD allowed us to capture their profiles with low overheads and verify some of the results by examining the source code. Instrumenting Windows XP allowed us to observe its internal behavior, which is not otherwise possible without access to the source code. We chose source code instrumentation techniques for the Linux and FreeBSD profilers for performance and portability reasons. We chose plug-in or binary rewriting instrumentation for the Windows profilers because source code is not available.

5.1 The aggregate_stats Library

This C library provides routines to allocate and free statistics buffers, store request start times in context variables, calculate request latencies, and store them in the appropriate buckets. We use the CPU cycle counter (TSC on x86) to measure time because it has a resolution of tens of nanoseconds, and querying it uses a single instruction. The TSC register is 64-bit wide and it only overflows once after counting 2^{64} CPU cycles (2^{32} seconds or more than hundred years for a CPU running at 4 GHz).

5.2 POSIX User-Level Profilers

We designed our user-level profiling mechanisms with portability in mind. We directly instrumented the source code of several programs used to generate test workloads in such a way that system calls are replaced with macros that wrap the system call with the appropriate profiling code. This way, the same programs can be recompiled for other POSIX-compliant OSs and immediately used for profiling. The collected profiles are printed to the standard output upon the program's exit. Alternatively, we can modify the libraries to intercept the calls at runtime, letting us profile programs for which source code is not available.

5.3 Windows User-level Profilers

To profile Windows under workloads generated by arbitrary non-open-sourced programs, we created a runtime system-call profiler. In Windows, system calls are implemented in a system-provided dynamic link library (DLL) called `kernel32.dll`. Our system call profiler is implemented as a DLL and uses the Detours library [44] to insert instrumentation functions for each system call of interest. Detours can insert new instrumentation into arbitrary Win32 functions even during program execution. It implements this by rewriting the target function images. To exercize a workload, we run a program that executes the test application and injects the system call profiler DLL into the test program's address space. On initialization, the profiler inserts instrumentation functions for the appropriate Windows system call. The Detours library implements the instrumentation by creating a *trampoline* function that invokes the actual system call being profiled for each instrumented function. The profiler's instrumentation functions call the corresponding trampoline function and capture the timing information.

5.4 Linux and FreeBSD File System Level Profilers

On Linux and FreeBSD source code is available for most file systems. Therefore, we decided to instrument file systems by directly instrumenting their source code. This allowed us to profile without perturbing the original file system operation. If the source code is not available, we can instrument and use stackable file systems [115]. Manual file system instrumentation is inconvenient because (1) the number of existing file systems is large, (2) each file system can support dozens of operations, and (3) most importantly, file systems can significantly change from one version of OS to another.

Before we started working on FoSgen, we created a simpler shell script that uses `sed` to add profiling code to Linux file systems. The script operates similarly to FoSgen as described in Section 4: it looks for particular patterns within file-system code. Despite its simplicity (the script consists of 184 `sed` expressions), the script successfully instrumented all the file systems we tried it on: Ext2 and Ext3 [19], Reiserfs 3.6 and 4.0 [84], NFS [78], NTFS [90], and Base0fs [115]—under both Linux kernel versions 2.4.24 and 2.6.11.

5.4.1 FoSgen and FSprof FiST extensions

FoSgen is a general file system instrumentation tool. It can add custom file system extensions under Linux and FreeBSD. More importantly, FoSgen can be extended to support more OSs in the future. FoSgen is written in *Perl* [104]. We believe Perl is an appropriate choice because most of the time the instrumentation process searches the code for matches of regular expressions—a task that Perl is especially suitable for. Aside from string matching and replacing, the instrumentation system parses only small portions of the C code. For example, it determines the types of functions and their arguments. We found that a simple top-down parser is sufficient for this purpose.

At the time of this writing, FoSgen can instrument Linux 2.4, Linux 2.6, and FreeBSD 5 file systems. It consists of 607 lines of Perl code. We have successfully applied and verified the functionality of several FiST extensions.

FSprof is our collection of FiST file system extensions that profile latency. We have created several such extensions with different portability and convenience characteristics. The smallest FSprof FiST extension is shown in Figure 4.6 on page 44. It describes appropriate function calls to be inserted at the beginning and at the return points of every file system operation. The function implementations must be provided in a separate OS-specific module. This extension is designed with maximum portability in mind—no extension changes should be necessary even if FoSgen is extended to support some new OSs. The FSprof extension described in Appendix A contains all the necessary functionality including the `/proc` interface implementation. This extension only works on Linux because of the Linux-specific `/proc` interface implementation. The FSprof extension with the `/proc` interface that we imported from a separate OS-specific module is portable across Linux and FreeBSD.

5.5 Windows File System Level Profilers

FoSgen can potentially be extended to support the instrumentation of Windows file systems. However, most Windows file systems' source code is unavailable or only partially available. Also, the Windows VFS API does not change a lot between Windows versions and there are usually only several functions to instrument due to the message-passing architecture of Windows file systems. Therefore, we decided to implement the Windows kernel mode profiler as a file system filter driver [72] that stacks on top of local or remote file systems.

In Windows, an OS component called the *I/O Manager* defines a standard framework for all drivers that handle I/O. The majority of I/O requests to file systems are represented by a structure called the *I/O Request Packet* (IRP) that are received via entry points provided by the file system. The type of an I/O request is identified by two IRP fields: MajorFunction and MinorFunction. In certain cases, such as when accessing cached data, the overhead associated with creating an IRP dominates the cost of the entire operation, and so Windows supports an alternative mechanism called Fast I/O. Our file system profiler intercepts all IRPs and Fast I/O traffic that is destined to local or remote file systems. Before passing the I/O request to the lower driver, our profiler begins measuring the latency; when the I/O request is completely processed, it ends the measurement of the I/O operation latency.

5.6 Driver Level Profilers

In Linux, file system writes and asynchronous I/O requests return immediately after scheduling the I/O request. Therefore, their latency contains no direct information about the associated I/O times. To detect this information, we instrumented a SCSI device driver by adding four calls to the `aggregate_stats` library. Windows provides a way to create stackable device drivers, but we did not create one because the file system layer profiler in Windows already captures latencies of writes and asynchronous requests.

5.7 Profile Analysis Automation

We have implemented several scripts that allow us to sort complete profiles by their characteristics. For example, most profiles shown in this dissertation are sorted based on the total latency of operations. Also, we have implemented several methods to compare pairs of individual profiles. In addition, we have combined these methods with several simple heuristics that allowed us to automate complete profiles analysis even more.

5.7.1 Individual Profiles Comparison

Automatic profiles independence tests are useful to select a small subset of operations for manual analysis from a large set of all operations. Also, such tests are useful to verify similarity of two profiles. Let us call the number of operations in the b^{th} bucket of one profile n_b, and the number of operations in the same bucket of the same operation in another profile m_b. Our goodness-of-fit tests return percent difference D between two profiles:

TOTOPS The degree of difference between the profiles is equal to the normalized difference of the total number of operations:

$$D = \frac{|\sum n_i - \sum m_i|}{\sum n_i} \times 100$$

TOTLAT The degree of difference between the profiles is equal to the normalized difference of the total latency of a given operation:

$$D = \frac{|\sum \frac{3}{2}2^{n_i} - \sum \frac{3}{2}2^{m_i}|}{\sum \frac{3}{2}2^{n_i}} \times 100 = \frac{|\sum 2^{n_i} - \sum 2^{m_i}|}{\sum 2^{n_i}} \times 100$$

CHISQUARE There are several methods of comparing histograms where only bins with the same index are matched. Some examples are the chi-squared test, the Minkowski form distance [100], histogram intersection, and the Kullback-Leibler/Jeffrey divergence [65]. The drawback of these algorithms is that their results do not match human perception. For example, if we had a histogram where only bins 1 and 5 were filled, and the contents were shifted to the right by one bin, we would not consider the difference to be too great. The bin-by-bin comparison methods, however, would consider this more different than if the contents of bin 5 were shifted to bin 2, which is perceptually more different. We have implemented the chi-square test as a representative of this class of algorithms because the chi-square test is “the accepted test for differences between binned distributions” [83]. It is defined for two histograms as follows:

$$\chi^2 = \sum \frac{(n_i - m_i)^2}{n_i + m_i}$$

The χ value can be mapped to the probability value P between 0 and 1, where a small value indicates a significant difference between the distributions. To match the semantics and scale of the previous two tests, we present $D = (1 - P) \times 100$. We utilized the standard **Statistics::Distributions** Perl library [64] in the implementation.

EARTHMOVER The Earth Mover's Distance (EMD) algorithm is a goodness-of-fit test commonly used in data visualization [89]. The idea is to view one histogram as a mass of earth, and the other as holes in the ground. The EMD is the least amount of work needed to fill the holes with earth, where a unit of work is moving one unit by one bin. This algorithm does not suffer from the problems associated with bin-by-bin and cross-bin comparison methods, and is specifically designed for visualization. Since the number of operations in the profiles are not necessarily equal, the histograms were normalized. We implemented the calculation as a greedy algorithm.

5.7.2 Combined Profile Comparison Methods

We have created several profile comparison methods that combine simple techniques that we use when manually comparing profiles. We first use the total number of operations and the total latency to determine if the profiles are very similar, very different, or insignificant in the context of the set of profiles. In these cases, the analysis is done, and the profiles are either the same or different (they receive a score of 0 or 100). Otherwise, we distinguish the peaks on the profiles using derivatives. If the number of peaks differs between the profiles, or their locations are not similar, the profiles are considered to be different (score of 100). As we will see in Section 6.3, these preparation steps alone significantly decrease the number of incorrectly classified profiles. If after these preparation steps the profile analysis is not over we can perform further comparisons based on the previously described algorithms (`TOTOPS`, `TOTLAT`, `CHISQUARE`, and `EARTHMOVER`). This way, we have implemented the following two methods:

GROUPOPS If the peaks in the profiles are similar, the score is the normalized difference of operations for individual peaks.

GROUPLAT This method is same as `GROUPOPS`, except that we calculate latency differences for individual peaks.

We will evaluate different properties of the implemented methods in Section 6.3.

5.8 Representing Results

All our profilers output their collected results to the user in the form of plain text. Plaintext is more convenient than binary data, because it is directly human-readable and powerful text processing utilities can be used to analyze it. The overhead associated with generating the plaintext profile is small, because the results are generally small and reading the profile is a rare operation. We denote profiles with all zeros using the minus character. Shown below is a sample profiler output, which consists of the operation name, the number of operation invocations, and the total operation latency, followed by a timeline of buckets (sampled profile):

```
OP_WRITE_SUPER 14 213428976
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
-
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
-
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
-
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
OP_LOOKUP ...
```

User-mode profilers print results directly to the `stderr` output. Our Linux and FreeBSD file system and driver profilers output results via their entries in the `/proc` interface. Writing to the per-file-system `/proc` entry resets the profile's counters.

We wrote several scripts to generate formatted text views and Gnuplot scripts to produce 2D and 3D plots. All the figures representing profiles in this dissertation were automatically generated. We found the following two data views for a particular VFS-operation especially useful:

- The total number of invocations with a given latency (*e.g.,* shown in Figure 3.2).
- The number of invocations with a given latency within each elapsed-time interval (*e.g.,* shown in Figure 3.5).

More views can be obtained from the original 4D profiles (the four dimensions are 1) operation, 2) latency, 3) number of operations with this latency, and 4) elapsed time interval) by summing up the values in one or more of the dimensions.

During profiling, the total number of operations is updated before entering the profiled code and the buckets are updated after returning from it. This allows our data-processing script to check the profile for consistency. In particular, for every operation, results in all of the buckets are summed and then compared with the total number of operations. This way, our data-processing script verification can catch many potential code instrumentation errors.

5.9 Portability

Each of our instrumentation systems consists of three parts: (1) instrumentation hooks; (2) the aggregate statistics library, which is common to all of our profiling systems; and (3) a reporting infrastructure to retrieve buckets and counts.

- The aggregate statistics library is 141 lines of C code, and requires no changes for different platforms. *(Shared among all OSs)*
- For POSIX user-space applications, our instrumentation and reporting interface used 68 lines. *(Works on most OSs).*
- For file systems, we wrote an automatic instrumentation script in `bash` and `sed` that can instrument any file system under 2.4 or 2.6. The shell script is 307 lines and contains 184 distinct `sed` expressions. We also wrote 221 lines of C code for the generic-function wrappers. *(Linux-specific)*
- FoSgen consists of 607 lines of Perl code. *(Linux and FreeBSD)*
- In the Linux kernel, we used the `/proc` interface for reporting, which consists of 163 lines. *(Linux-specific).*
- The instrumentation hooks for our Linux device driver used 10 lines. *(Linux-specific).*
- We used the Detours library [44] for the Windows user-space profiling tool. We added 457 lines of C code to intercept 112 functions, of which 337 lines are repetitive pre-operation and post-operation hooks. *(Windows-specific)*
- Our Windows filter driver was based on FileMon [101] and totaled 5,262 lines, of which 273 were our own C code and 63 of which were string constants. We also wrote a 229-line user application to retrieve the profile from the kernel. *(Windows-specific)*

In sum, our instrumentation system is fairly portable. The aggregate statistics library runs without changes in several different environments: Unix applications, Windows applications, and the Linux, FreeBSD, and Windows kernels. POSIX-compliant user-mode applications are portable across most OSs. *Optional* file system and driver instrumentation infrastructure is fairly portable and simple. More code is required to implement the profile reporting infrastructure from the kernel to the user mode. This is partially because we used a plain text representation of the results. Note that this code is required for any in-kernel profiler in order to report collected results.

Chapter 6

Evaluation

We evaluated the overhead of our Linux 2.6.11 Ext2 file system profiler with respect to memory usage, CPU cache usage, the latency added to each profiled operation, and the overall execution time. We chose to instrument a file system, instead of a program, because a file system receives a larger number of requests (due to the VFS calling multiple operations for some system calls) and it demonstrates higher overheads. Moreover, user-level profilers primarily add overheads to user time. We conducted all our experiments on a 1.7 GHz Pentium 4 machine with 256KB of cache and 1GB of RAM. It uses an IDE system disk, but the benchmarks ran on a dedicated Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disk with an Adaptec 29160 SCSI controller. We used the Auto-pilot benchmarking suite [111] to unmount and remount all tested file systems before each benchmark run. We also ran a program we wrote called *chill* that forces the OS to evict unused objects from its caches by allocating and dirtying as much memory as possible. We ran each test at least ten times and used the Student-t distribution to compute the 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time and consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean.

6.1 Memory Usage and Caches

We evaluated the memory and CPU cache overheads of the file system profiler. The memory overhead consists of three parts. First, there is some fixed overhead for the aggregation functions. The initialization functions are seldom used, so the only functions that affect caches are the instrumentation and sorting functions which use 231 bytes. This is below 1% of cache size for all modern CPUs. Second, each VFS operation has code added at its entry and exit points. For all of the file systems we tested, the code-size overhead was less than 9KB, which is much smaller than the average memory size of modern computers. The third memory overhead comes from storing profiling results in memory. A profile occupies a fixed memory area. Its size depends on the number of implemented file system operations and is usually less than 4KB.

6.2 CPU Time Overheads

To measure the CPU-time overheads, we ran Postmark v1.5 [60] on an unmodified and on an instrumented Ext2. Postmark simulates the operation of electronic mail servers. It performs a series of file system operations such as create, delete, append, and read. As shown in Figure 6.1, we configured Postmark to use the default parameters, but we increased the defaults to 20,000 files and 200,000 transactions so that the working set is larger then OS caches and so that I/O requests will reach the disk. This configuration runs long enough to reach a steady-state and it sufficiently stresses the system.

Overall, the benchmarks showed that wait and user times are not affected by the added code. The unmodified Ext2 used 18.3 seconds of system time, or 16.8% of elapsed time. The instrumentation code increased system time by 0.73 seconds (4.0%). As seen in Figure 6.2, there are three additional components added: making function calls, reading the TSC register, and storing the results in the correct buckets. To understand the details of this per-operation overheads, we created two additional file systems. The first contains only empty profiling function bodies, so that the only overhead is calling the profiling functions. Here, the system time increase over Ext2 was 0.28 seconds (1.5%). The second file system read the TSC register, but did not include code to sort the information or store it into buckets. Here, the system time increased by 0.36 seconds over Ext2 (2.0%). Therefore, 1.5% of system time overheads were due to calling profiling functions, 0.5% were due to reading the TSC, and 2.0% were due to sorting and storing profile information.

Not all of the overhead is included within the profile results. Only the portion between the TSC register reads is included in the profile, and therefore it defines the minimum value possible to record in the buckets. Assuming that an equal fraction of the TSC is read before and after the operation is counted, the delay between the two reads is approximately equal to half of the overhead imposed by the file system that only reads the TSC register. We computed the average overhead to be 40 cycles per operation. The 40-cycle overhead is well below most operation latencies, and can influence only the fastest of VFS operations that perform very little work. For example, `sync_page` is called to write a dirty page to disk, but it returns immediately if the page is not dirty. In the latter case its latency is at least 80 cycles long.

```
set size 512 10240
set number 20000
set seed 42
set transactions 200000
set location /n/test/fsprof
set subdirectories 600
set read 4096
set write 4096
set buffering false
set bias read 5
set bias create 5
```

Figure 6.1: Postmark configuration that we used for FSprof benchmarking.

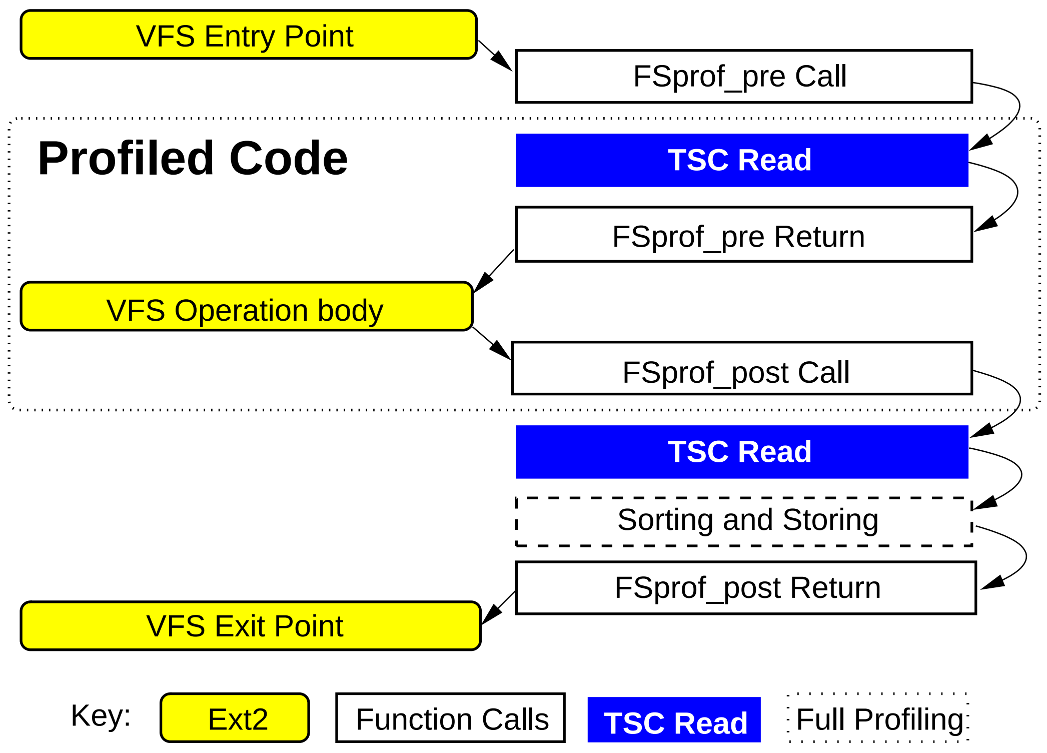


Figure 6.2: Profiled function components.

6.3 Profile Analysis Automation

To evaluate our profile analysis automation methods we compared the results of the automatic profile comparison with manual profile comparison. In particular, we analyzed 150 profiles of individual operations from these that we will describe in Chapter 7. We manually classified these profiles into “different” and “same” categories. A false positive (or a type I error) is an error when two profiles are reported different whereas they are same according to the manual analysis. A false negative (or a type II error) is an error when two profiles are reported same whereas they are different according to the manual analysis. Our tests return the profile’s difference value. A difference threshold is the value that delimits decisions of our binary classification based on the test’s return values.

Figures 6.3–6.8 show the dependencies of the number of false positives and false negatives on the normalized difference of the two profiles calculated by the six profile comparison methods that we have implemented. As we can see, EMD algorithm had a threshold region with the smallest error rates of both types. However, both our custom-made methods `GROUPOPS` and `GROUPLAT` have a wide range of difference thresholds where both errors are below 5%. This means that these methods can produce reliable and stable results for a wide range of profiles.

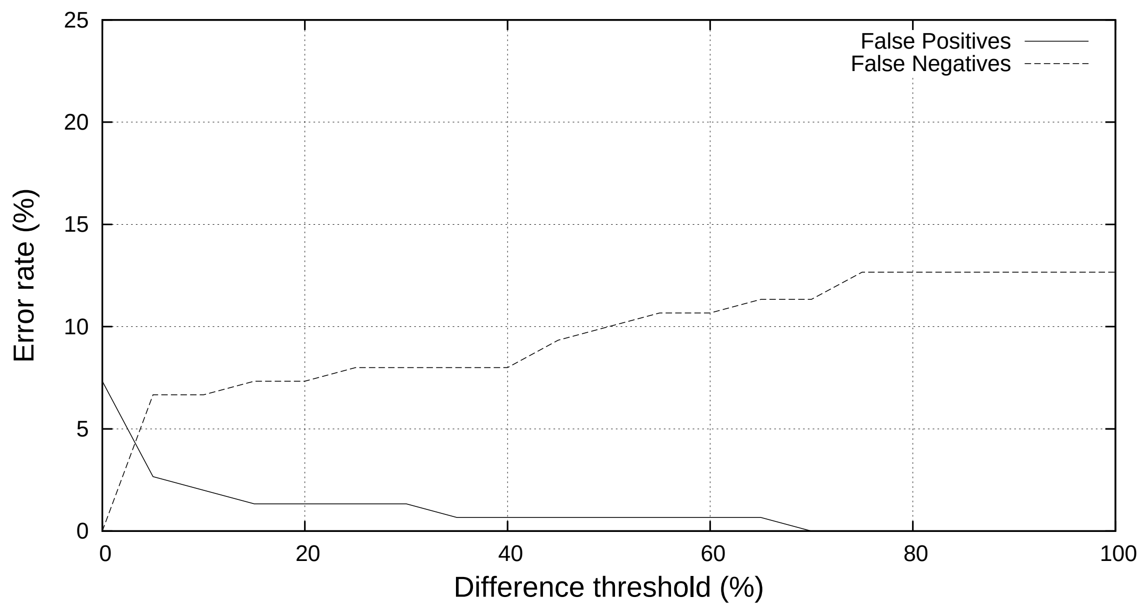


Figure 6.3: `TOTOPS` test results compared with manual profiles analysis.

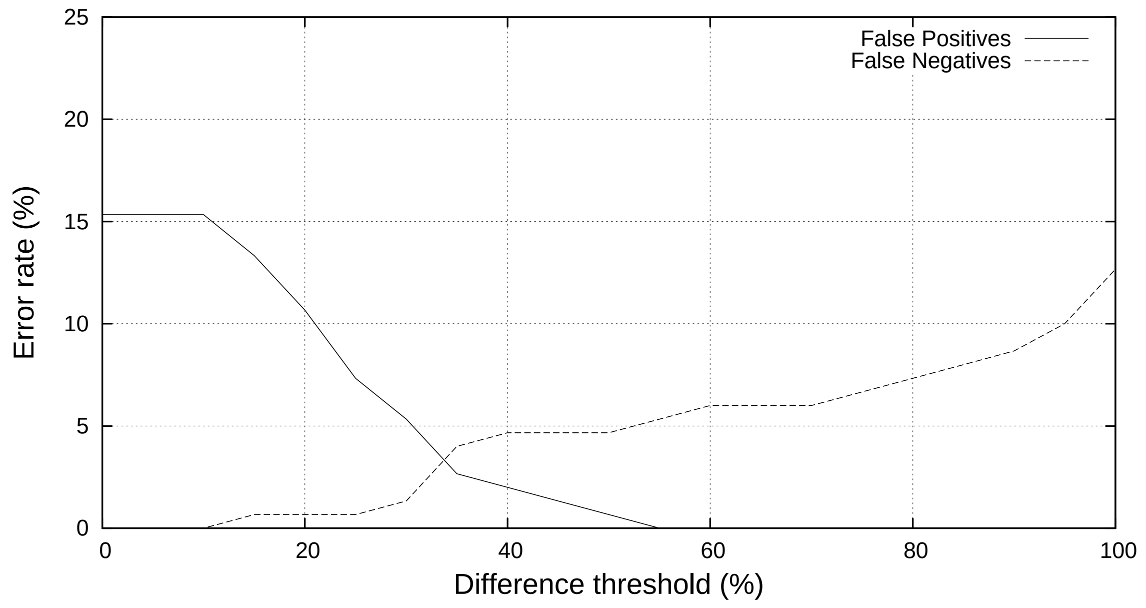


Figure 6.4: `TOTLAT` test results compared with manual profiles analysis.

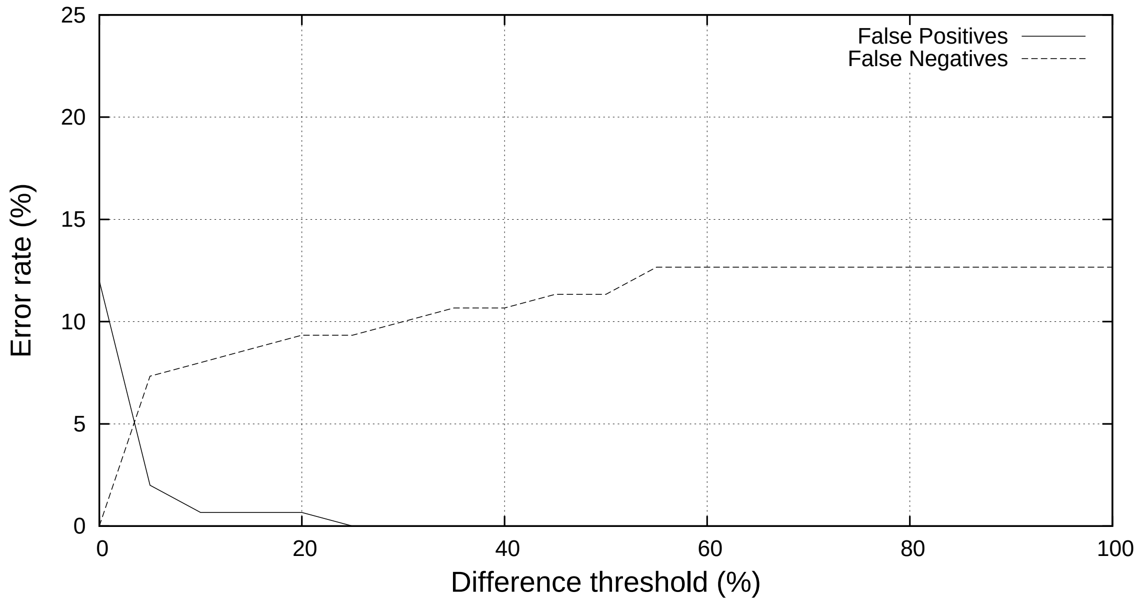


Figure 6.5: `CHISQUARE` test results compared with manual profiles analysis.

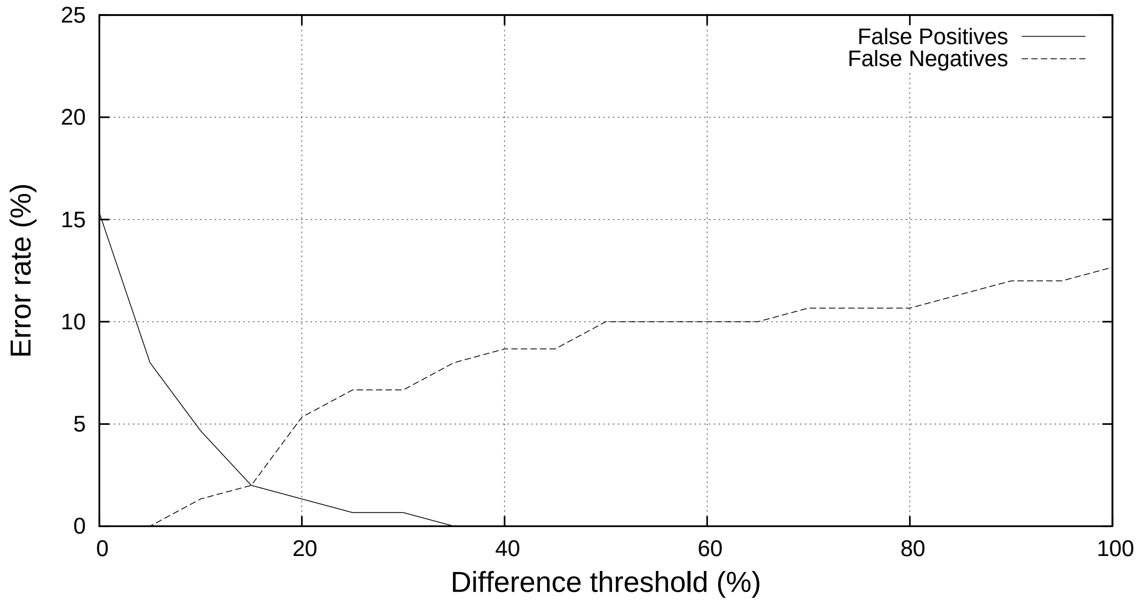


Figure 6.6: `EARTHMOVER` test results compared with manual profiles analysis.

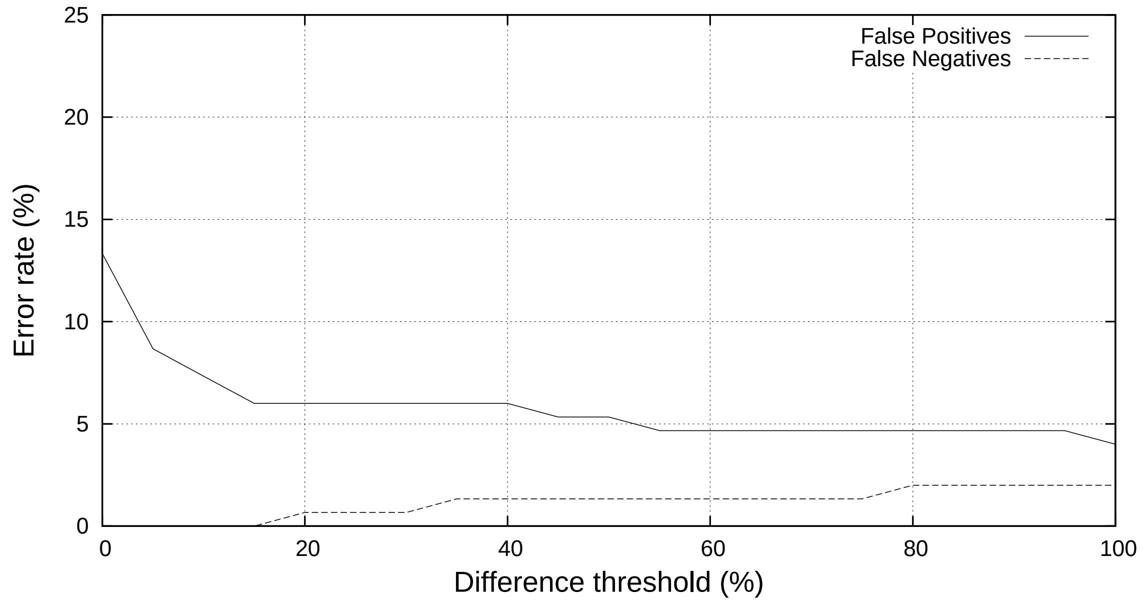


Figure 6.7: `GROUPOPS` test results compared with manual profiles analysis.

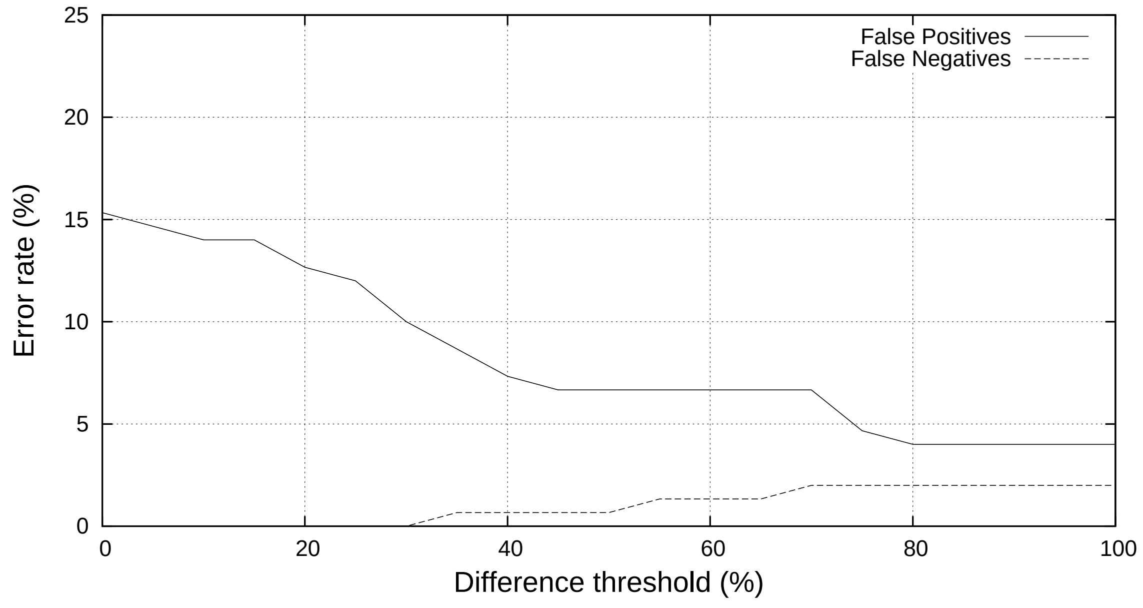


Figure 6.8: `GROUPLAT` test results compared with manual profiles analysis.

6.4 FoSgen

Finally, we evaluate the efficiency of our FoSgen prototype. Its efficiency is less important than the runtime profiling overheads. However, short instrumentation times are more desirable for file system developers. At first glance it may seem that the process of file system generation can take a considerable amount of time. Indeed, the generation process requires at least two scans of the file system source code. Table 6.1 shows the times necessary to add (1) secure deletion [57] and (2) FSprof latency profiling extensions to several popular Linux 2.6.16 and FreeBSD 5.3 file systems and their compilation times. Note that the overheads of compilation times were indistinguishable. As we can see, in all the cases except Base0fs, the instrumentation times were smaller than the compilation times of these file systems. This is because Base0fs explicitly implements most existing file system operations for extensibility reasons. Fortunately, its absolute compilation time is small compared with other file systems. Also, we can see that FreeBSD instrumentation is faster than instrumentation of Linux file systems. This is because FreeBSD has a simpler VFS API.

File system	**OS**	**SecDel addition time (seconds)**	**FSprof addition time (seconds)**	**Compilation time (seconds)**
Ext2	Linux	0.3	9.4	14.1
Ext3		0.4	16.8	25.0
vfat		0.2	0.3	2.5
NFS		0.5	20.4	30.2
CIFS		0.6	21.7	28.3
ramfs		0.1	0.6	2.4
Reiserfs		0.6	29.7	33.1
Base0fs		0.3	6.4	5.8
NFS	FreeBSD	0.5	7.3	20.2
msdosfs		0.4	3.4	15.3
nullfs		0.1	0.8	6.5

Table 6.1: FoSgen instrumentation times of several Linux 2.6.16 and FreeBSD 5.3 file systems with secure deletion (SecDel) and latency profiling (FSprof) extensions. The rightmost column shows the original compilation times for these file systems. We performed the tests with warm file system caches.

Chapter 7

Example File System Profiles

In this chapter we describe a few interesting examples that illustrate our method of analyzing file system behavior. We concentrated on profiles of popular disk-based and network file systems. Such profiles tend to contain a wider spectrum of events. We conducted all experiments on the same hardware setup as described in Section 6. Unless noted otherwise, we profiled a vanilla Linux 2.6.11 kernel and Windows XP SP2. All profiles presented in this section are from the file-system level except Figure 7.15.

We ran two workloads to capture the example profiles: a *grep -r* and a *random-read* on a number of file systems. The *grep -r* workload was generated by the `grep` utility that was recursively reading through all of the files in the Linux 2.6.11 kernel source tree. We have chosen the *grep -r* workload because it is simple but at the same time it triggers many different file system operations. The *random-read* workload was generated by two processes that were randomly reading the same file using direct I/O mode. In particular, these processes were changing the file pointer position to a random value and reading 512 bytes of data at that position. Of note is that we did not have to use many workloads to reveal a lot of new and useful information. After capturing just a few profiles, we were able to spot problematic patterns by simply looking at the profiles of different operations.

7.1 Analyzing Disk Seeks

In Linux, the current file pointer position is stored in the `file` data structure, which usually belongs to a single process. Information such as the file size is shared between processes and is stored in the `inode` data structure which is unique for any given file. Therefore, one would expect that a change of the file pointer position would not cause contention between processes, because only the per-process data structures are updated. The profiles shown in Figure 7.1 were captured by applying the *random-read* workload. The profile shows that the `llseek` operation of one process competes with the `read` operation of another process. We see this because the `llseek` operation profile partially resembles the `read` profile and this behavior does not occur with only one process running. (On average, `llseek` waits for half of the duration that the semaphore is held by `read`. Therefore, the corresponding `llseek` buckets are shifted by one to the left.) This means that a corresponding number of `llseek` operations were waiting for the `read` operation to release

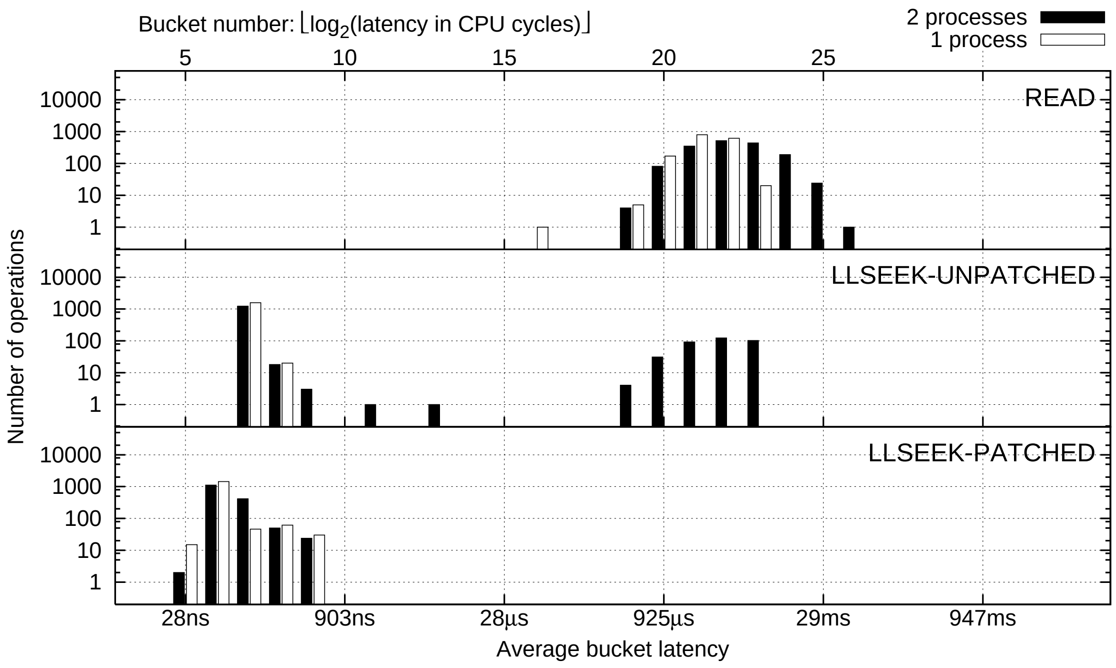


Figure 7.1: The `llseek` operation under random reads.

some lock or a semaphore.

Upon investigation of the source code, we verified that the delays were indeed caused by the `i_sem` semaphore in the Linux-provided method `generic_file_llseek`—a method which is used by most of the Linux file systems including Ext2 and Ext3. We observed that this contention happens 25% of the time, even with just two processes randomly reading the same file. We modified the kernel code to resolve this issue as follows. In particular, we observed that to be consistent with the semantics of other Linux VFS methods, no semaphore protection is necessary for file objects and it is necessary only for directory objects. The `llseek` profile captured on the modified kernel is shown at the bottom of Figure 7.1. As we can see, our fix reduced the average time of the `llseek` from 400 cycles to 120 cycles (a 70% reduction). The improvement is compounded by the fact that all semaphore and lock-related operations impose high overheads even without contention, because the semaphore function is called twice and its size is comparable to `llseek`. Moreover, semaphore and lock accesses require either locking the whole memory bus or at least purging the same cache line from all other processors, which can hurt performance on SMP systems.

We submitted a small patch which fixes this problem and its description to the core Linux VFS developers, who agreed with our conclusions [47]. We ran the same workload on a Windows NTFS file system, and found no lock contention. This is because on Windows, keeping the current file position consistent is left up to user-level applications.

7.2 Analyzing File System Read Patterns

We now show how we analyzed various file system I/O patterns under the `grep -r` workload. In this workload, we use the `grep` utility to search recursively through the Linux kernel sources for a nonexistent string.

7.2.1 Ext2

Peaks shown in the top profile of Figure 7.2 are common for many file system operations that require hard disk accesses. Here we refer to the `readdir` operation peaks by their order from left to right: first (buckets 6 and 7), second (9–14), third (16 and 17), and fourth (18–23). A complete profile of Linux 2.6.11 Ext2 under the `grep -r` workload is shown in Figure 7.3. (Notice the differences with a Linux 2.4.24 profile in Figure 3.2 on page 10.)

First peak (buckets 6–7) From the profile of Figure 3.8 we already know that on Linux, the peak in the 6^{th} bucket corresponds to a read of zero bytes of data or any other similar operation that returns right away. The `readdir` function returns directory entries in a buffer beginning from the current position in the directory. This position is automatically updated when reading and modifying the directory and can also be set manually. If the current position is past the end of the directory, `readdir` returns immediately (this happens when a program repeatedly calls `readdir` until no more entries are returned). Therefore, it seems likely that the first peak corresponds to the reads past the end of directory. One way to verify our hypothesis would be to profile a workload that issues `readdir` calls only after there are no more directory entries to read and then compare the resulting profiles (differential analysis). However, we can demonstrate our other method of profile analysis by directly correlating peaks and variables.

To do so, we slightly modified our profiling macros: instead of storing the latency in the buckets we (1) calculated a `readdir_past_EOF` value for every `readdir` call (`readdir_past_EOF` = 1 if the file pointer position is greater or equal to the directory

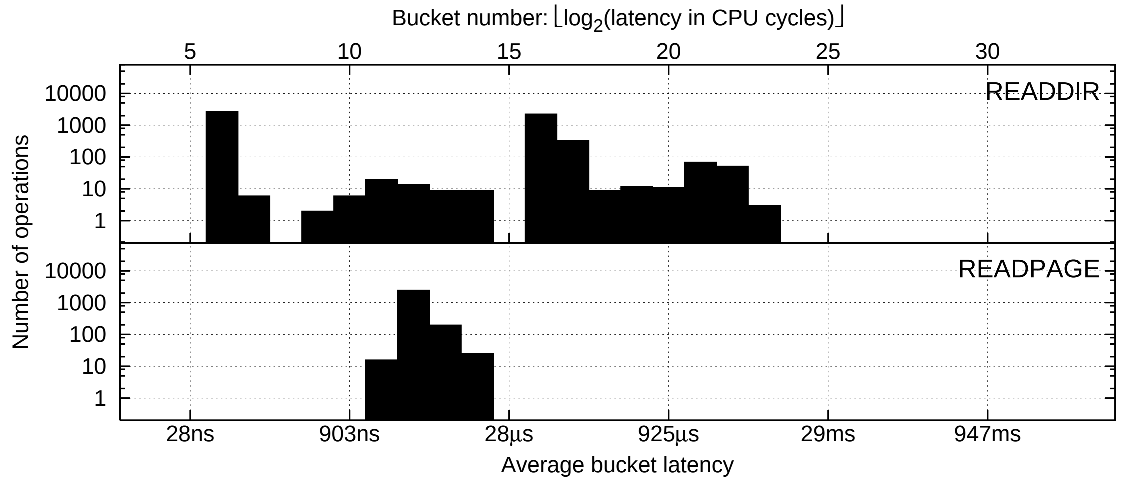


Figure 7.2: Profiles of Linux 2.6.11 Ext2 `readdir` (top) and `readpage` (bottom) operations captured for a single run of `grep -r` on a Linux source tree.

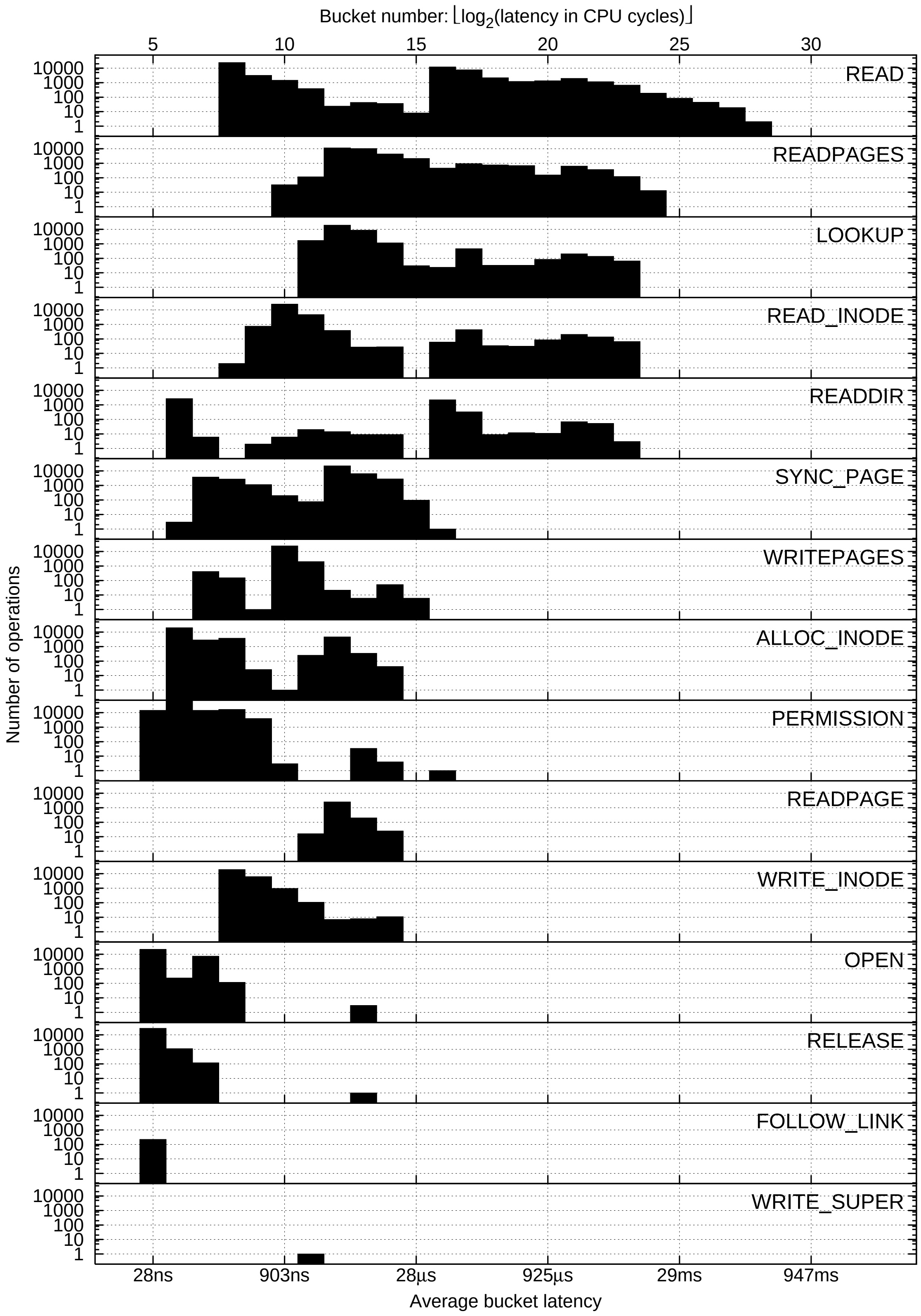


Figure 7.3: Profile of Linux 2.6.11 Ext2 under the `grep -r` workload.

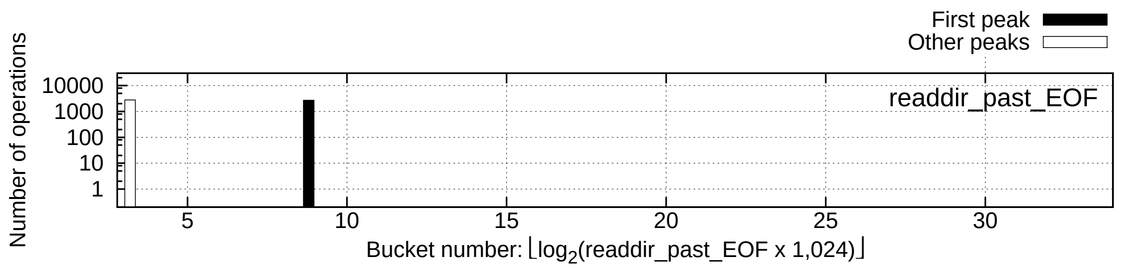


Figure 7.4: Correlation of the `readdir_past_EOF` $\times 1,024$ and the peaks in Figure 7.2.

buffer size and is 0 otherwise); (2) if the latency of the current function execution fell within the range of the first peak, a value of the bucket corresponding to `readdir_past_EOF` $\times 1,024$ was incremented in one profile and in another profile otherwise. The resulting profiles are shown in Figure 7.4 and prove our hypothesis.

Second peak (buckets 9–14) The `readdir` operation calls the `readpage` operation for pages not found in the cache. The `readpage` profile is a part of the complete `grep -r` workload profile and is shown on the bottom in Figure 7.2. During the complete profile preprocessing phase, our automatic profiles analysis tool discovered that the number of elements in the third and fourth peaks is exactly equal to the number of elements in the `readpage` profile. This immediately suggests that the second peak corresponds to `readdir` requests that were satisfied from the cache. Note that the latency of `readpage` requests is small compared to related `readdir` requests. That is because `readpage` just initiates the I/O and does not wait for its completion.

Third peak (buckets 16–17) The third and the fourth peaks of the `readdir` operation correspond to disk I/O requests. The third peak corresponds to the fastest I/O requests possible. It does not correspond to I/O requests that are still being read from the disk and thus may require disk rotations or even seeks. This is because the shape of the third peak is sharp (recall that the Y scale is logarithmic). Partially read data requests would have to wait for partial disk rotations and thus would spread to gradually merge with the fourth peak. Therefore, the third peak corresponds to I/O requests satisfied from the disk cache due to internal disk readahead.

Fourth peak (buckets 18–23) The fourth peak corresponds to requests that may require seeking with a disk head (track-to-track seek time for our hard drive is 0.3 ms; full stroke seek time is 8 ms) and waiting for the disk platter to rotate (full disk rotation time is 4 ms).

7.2.2 NTFS and Ext3

It is well known that disk accesses that require disk head seeks take considerably more time than those that do not; that is why Ext2 and Ext3 use an FFS-like allocation scheme [19]. Inodes in the same directory are stored in the same cylinder group (so they have closer block numbers).

On NTFS, all data and metadata is stored as regular files [96]. NTFS has a special file called a *master file table* (MFT), located near the beginning of the partition as a contiguous disk area (on unfragmented partitions). This file contains information about every file on the partition. Regular data is stored on the remainder of the partition. To minimize disk head seeks, actual file and directory data are put directly into the MFT if the corresponding data size is smaller than 1.5KB. Figure 7.5 shows the `read` operation on NTFS running on Windows and Ext3 running on Linux and the total impact of every bucket. Here, NTFS uses a normal `read` operation the first time the file is read and uses `FastIO` for additional reads. We combined these two operations in the figure for the purpose of comparing the two OSs more fairly, and the `read` operation for NTFS should be understood in this context for the remainder of the dissertation.

We can see how the on-disk structures for these two file systems affect these benchmarks. Both graphs have three distinct groups. NTFS is generally divided into the first two buckets (9–10), the next five (11–15), and the remainder (16–23). Ext3 is generally divided into the first two buckets (8–9), the next six (10–15), and the remainder (16–22). Based on the latencies, the first group represents data to be read from memory, the second group is when the data is in transit from the disk due to readahead, and the last group is when the disk needs to be accessed. Both file systems have a similar number of operations in their first groups, which is reasonable because we used the same data set for both file systems and the machines had the same amount of memory. NTFS has more than three times as many operations in its second group as Ext3 does. From this we can infer that NTFS has a more aggressive readahead policy. Finally, Ext3 has 12% more operations in its third group, but it still has lower latencies than NTFS's third group. This tells us that although we go to disk more often on Ext3, the seeks are shorter on average, making the total delays for going to disk on Ext3 less than NTFS. Going back to Figure 7.1 we can see that disk head seek times can differ by several orders of magnitude. Therefore, it is sometimes more important to avoid *long* disk head seeks than just any seeks, a fact that is often overlooked by file-system designers.

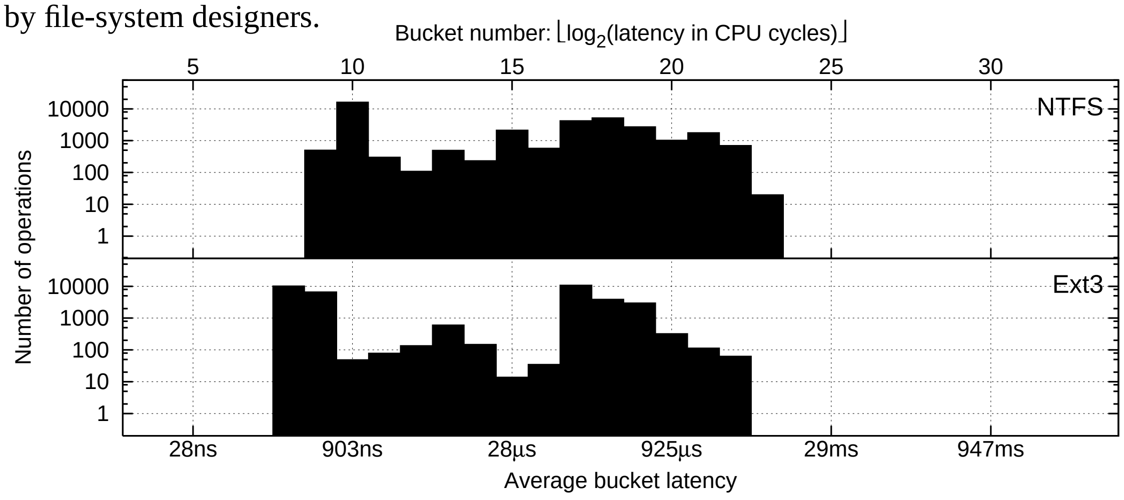


Figure 7.5: Profile of `read` operations under the *grep -r* workload for Ext3 on Linux and NTFS on Windows.

7.3 Handling Access Time Updates

Access times record when a file was last read. Maintaining them requires disk writes, even for a read-only workload. This interaction of reads and writes is interesting.

7.3.1 NTFS

NTFS updates a file's `atime` by writing asynchronously to the file's attribute, which is part of the MFT. The `atime` is also written to a directory-entry file stored in the same directory as the file. The writes to the MFT cause long seeks between the data on the disk, the journal, and the MFT.

The top graph of Figure 7.6 shows the latency distributions for the `atime` update operation. Because all metadata on NTFS is stored in files, the updates show up as normal `write` operations. One can see that the `write` operation spans 9 buckets. Therefore, the longest write takes 512 times longer than the fastest write that also requires a disk-head seek. The bottom graph of Figure 7.6 shows the `read` operation with and without `atime` updates. Reading has become slower due to seeking between the data being read and the MFT. Updating the `atime`s for this workload yielded a 16% average slowdown in elapsed time. Because of this performance issue, Windows does not keep the `atime` value current: NTFS updates it on disk only if the value in memory is at least one hour greater than what is currently on disk [25]. In addition, there is a registry key that allows users to disable all `atime` updates. Fortunately, however, `atime`s can be read from the local directory entry, so performance is only affected when they are updated.

7.3.2 Reiserfs and Profiles Sampling

By default, Reiserfs enables *tail merging* which combines small files and the ends of files into a single disk block. Figures 7.7–7.9 show the profiles of the Reiser 3.6 file system

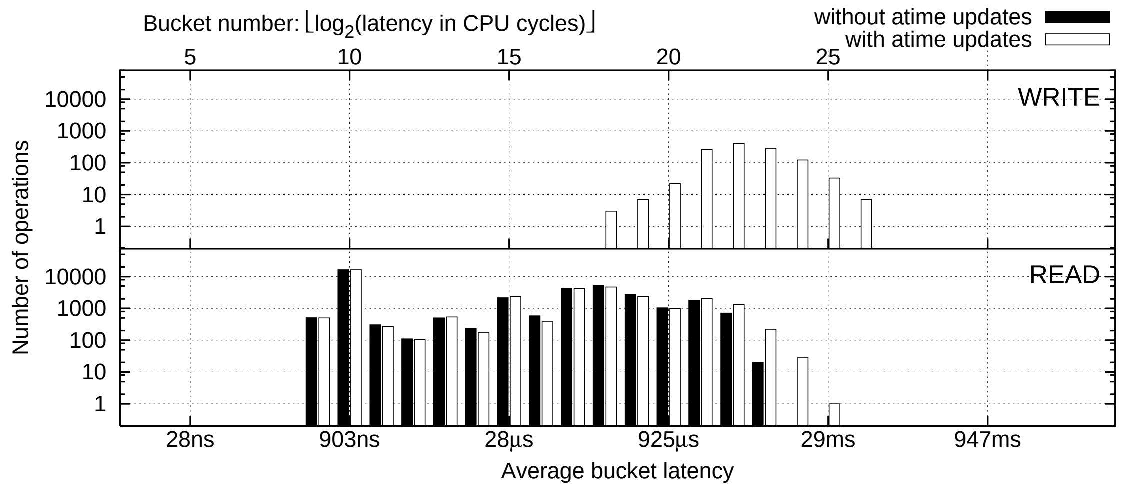


Figure 7.6: Effects of `atime` updates on NTFS operations.

in the default configuration, with the *notail* option, which disables tail merging, and Reiserfs 4.0 respectively.

The first interesting observation we can make is that the `write_super` operation on Reiser 3.6 takes longer than most other operations. The second observation is that there is a clear correlation between the longest `dirty_inode`, `read`, and `write_super` operations.

We use *profiles sampling* to analyze this behavior further. A sampled profile is similar to our standard profile, but instead of adding up all of the operations for a given workload, we divide the profile into fixed-time segments and save each segment separately. We show two such profiles for the `write_super` and `read` operations on Reiserfs 3.6 on Linux 2.4.24 in Figures 7.10 and 7.11, respectively. The Y axis shows the elapsed time in CPU cycles. To compare profiles with vastly different maximal values in the buckets, and to allow direct timeline comparisons, we have also shown the same profiles from a different angle (top view) on Figure 7.12. The three vertical black stripes on the `read` profile in that figure correspond to those peaks already shown in Figure 3.3: cached reads, disk-cached reads, and reads with a disk-head seek or a platter rotation.

We can see that the long operations are executed every 5 seconds, which suggests that they are invoked by the `bdflush` kernel thread to update access time information of the accessed inodes. On Linux, `atime` updates are handled by the Linux buffer flushing daemon, `bdflush`. This daemon writes data out to disk only after a certain amount of time has passed since the buffer was released; the default is thirty seconds for data and five seconds for metadata. This means that every five and thirty seconds, file system behavior may change due to the influence of `bdflush`. Updating `atime` causes journal writes, so `write_super` is called to flush the journal. The correlation between several operations is caused by the `write_super` operation, which always takes the Big Kernel Lock (BKL), a global kernel lock in Linux. The other operations must wait for the `write_super` operation to finish. This observation is especially important because it shows that Reiserfs 3.6 blocks not only its own operations, but those of other file systems and also many other kernel functions, for significant periods of time. In Figure 7.12, we can see that when the `write_super` operation is called there are also long-running `read` operations since the rightmost points on both graphs coincide (buckets 25–28). This is because in Reiserfs 3.6, the whole file system is locked for the duration of `write_super`, and therefore any process that attempts to read data has to wait on a semaphore until the write completes. We can see that this can stop all file-system-related requests for as long as 0.15 seconds (2^{28} CPU cycles on our 1.7 GHz CPU). Thus, even profiles collected over relatively long periods of time can reveal correlations between a test process and a periodic thread running in the kernel.

The results presented in Figure 7.9 demonstrate that Reiserfs 4.0's behavior is very different from that of Reiserfs 3.6. According to the general Linux development trend, Reiserfs 4.0 never takes the BKL. That is why Reiserfs 4.0 does not use the `write_super` operation—because it is called with the BKL held. This tends to reduce lock contention considerably and improves Reiserfs 4.0's performance overall. However, inode access time updates are still the longest individual operations in Reiserfs 4.0.

We also noticed that the `readdir` operation takes longer on Reiserfs 4.0 than 3.6. Upon inspection of the Reiserfs 4.0 code, we found out that its `readdir` operation also

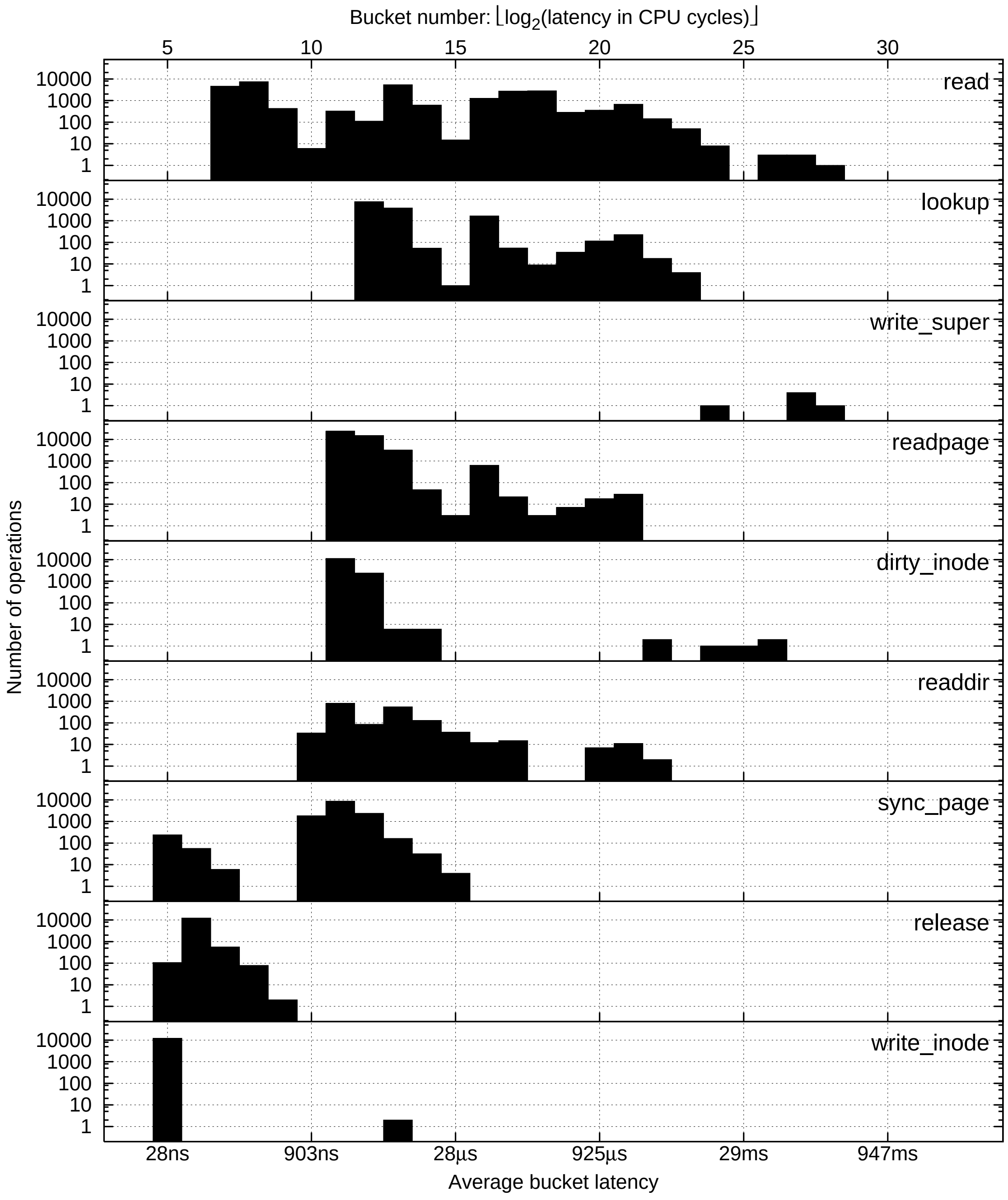


Figure 7.7: Profile of Reiserfs 3.6 (default configuration) under the *grep -r* workload.

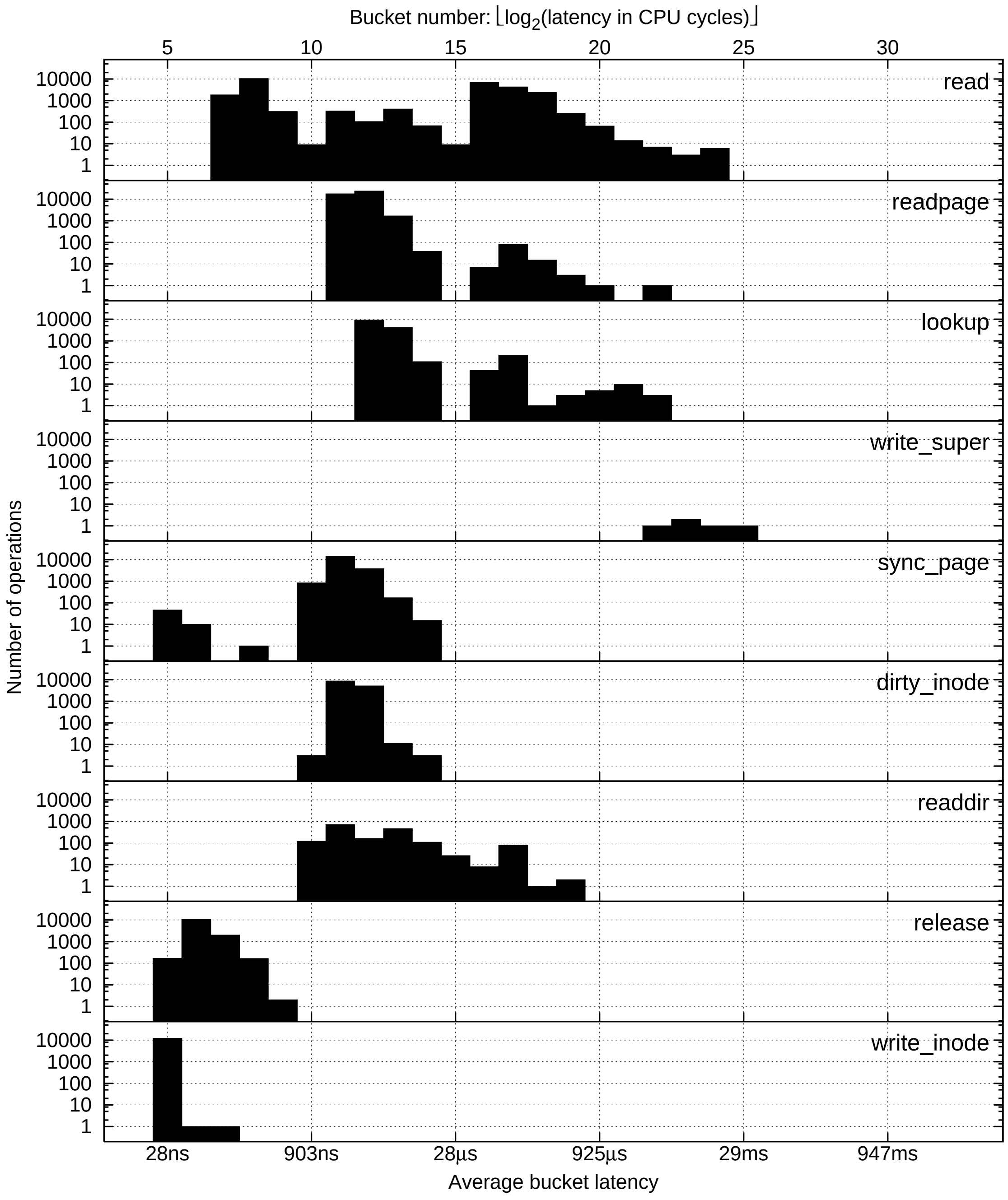


Figure 7.8: Profile of Reiserfs 3.6 (with *notail*) under the *grep -r* workload.

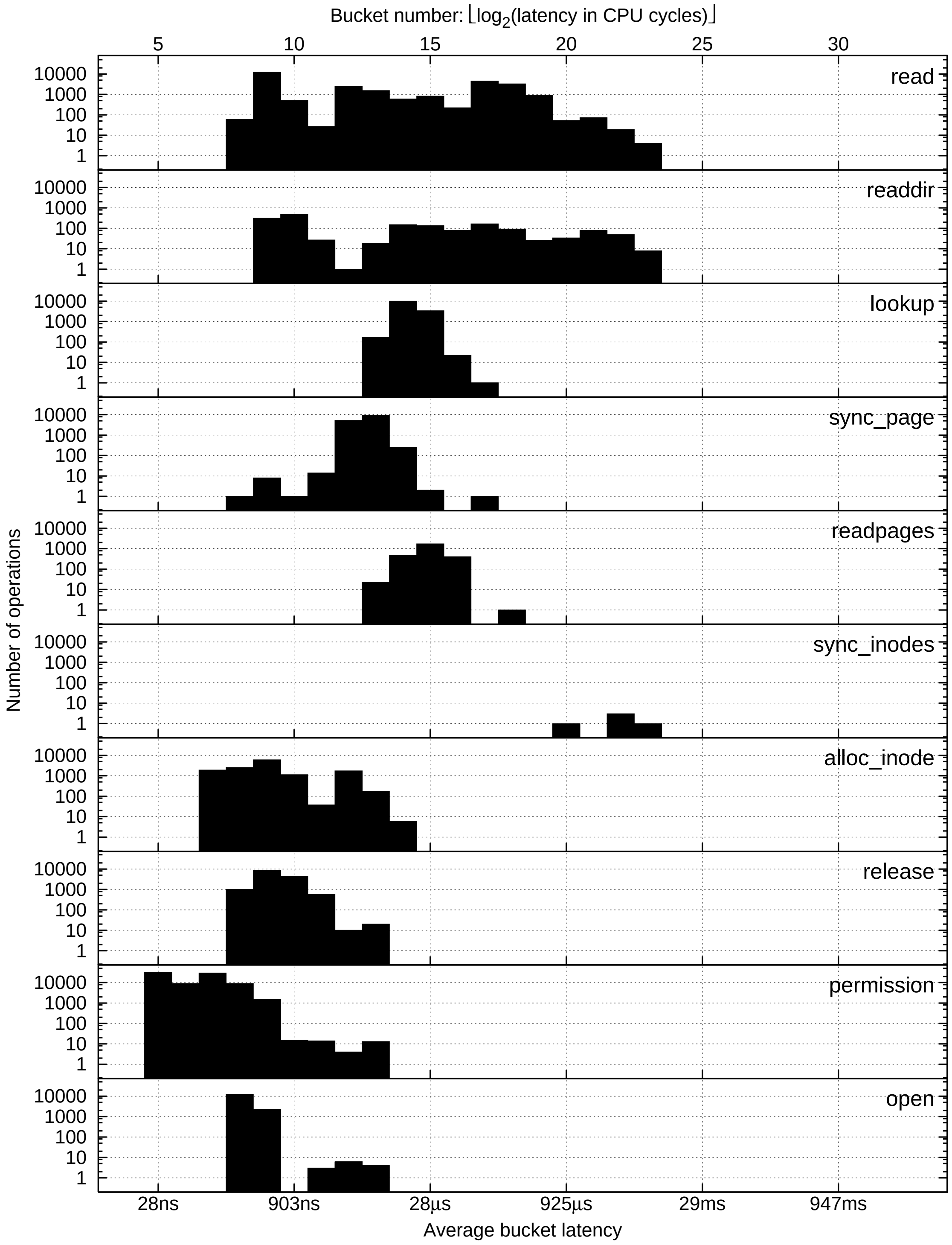


Figure 7.9: Profile of the Reiserfs 4.0 file system under the *grep -r* workload.

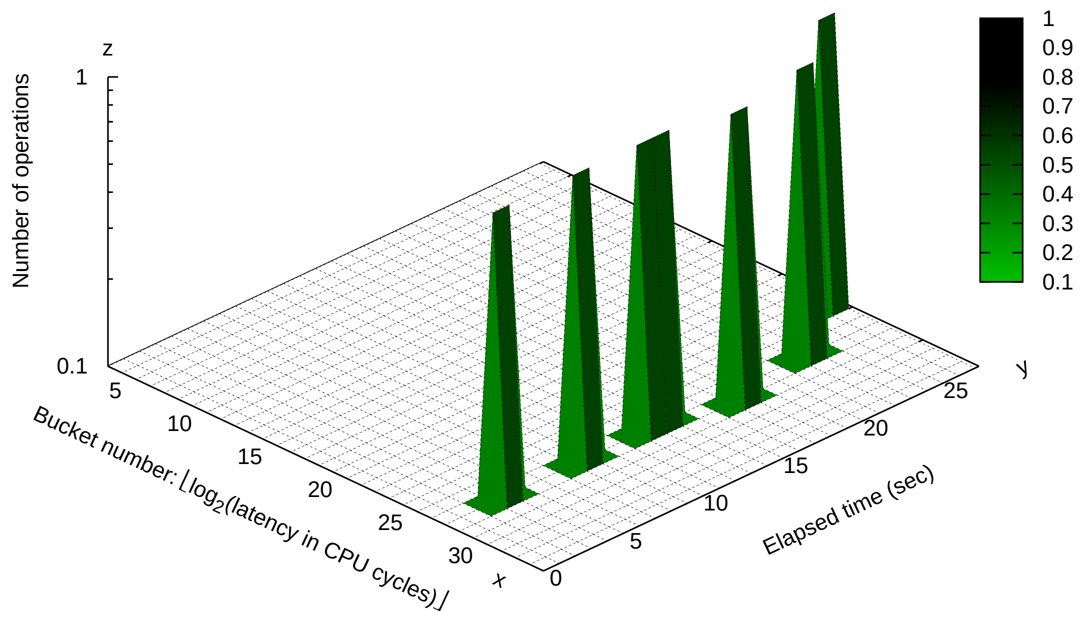


Figure 7.10: Linux 2.4.24 Reiserfs 3.6 (default configuration) `write_super` operation sampled profile under the *grep -r* workload.

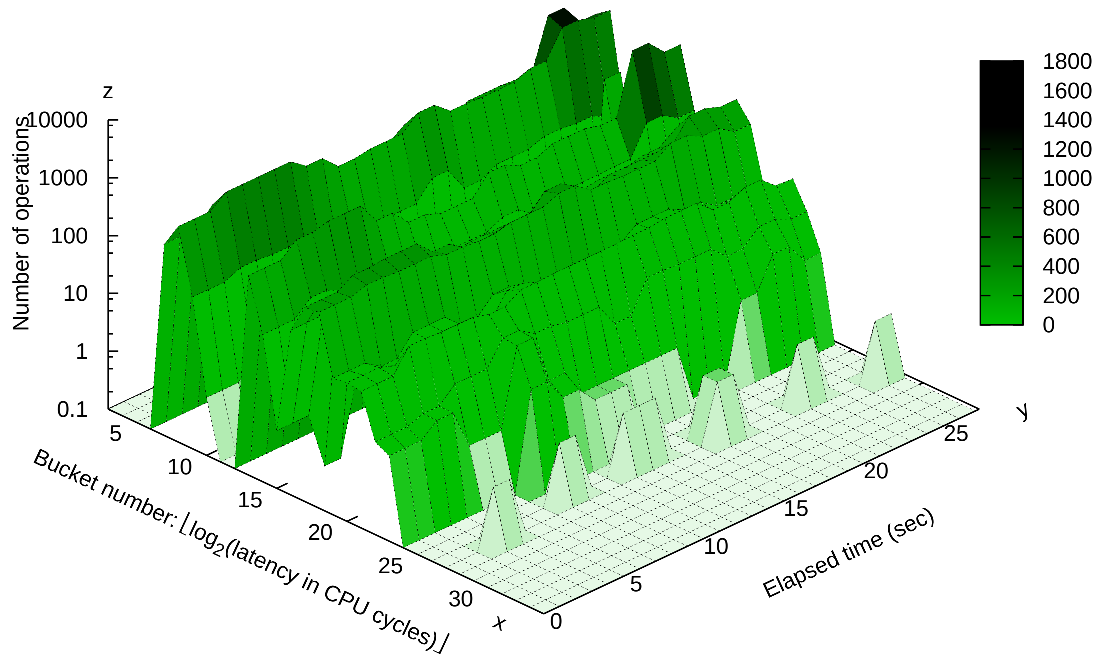


Figure 7.11: Linux 2.4.24 Reiserfs 3.6 (default configuration) `read` operation sampled profile under the *grep -r* workload.

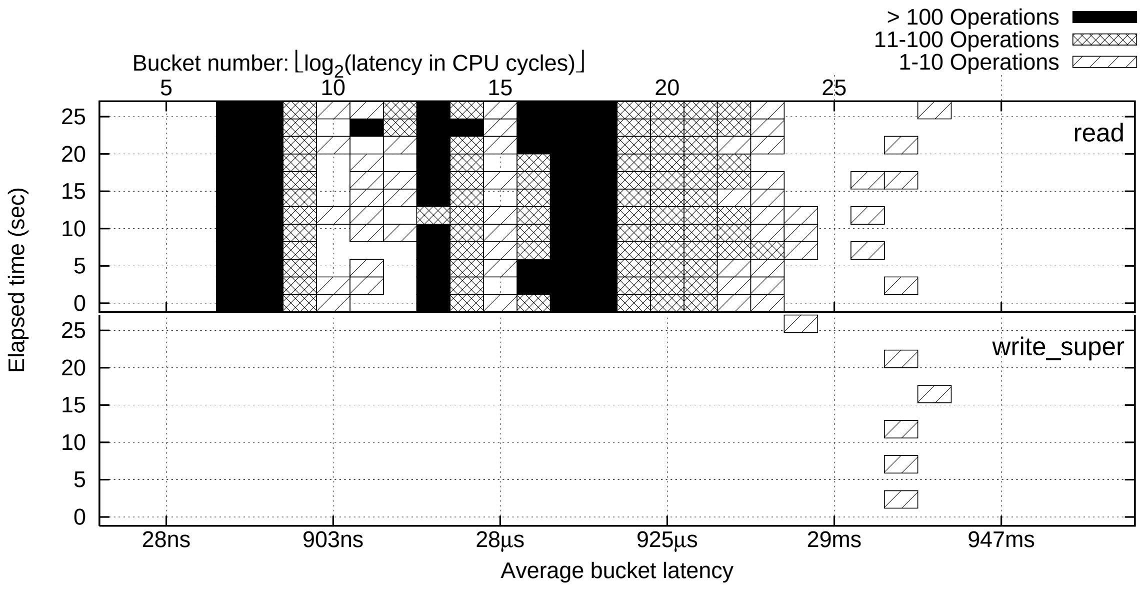


Figure 7.12: Linux 2.4.24 Reiserfs 3.6 file-system profiles sampled at 2.5 second intervals.

schedules read-aheads for the inodes of the directory entries being read. This is an optimization which was previously noted by NFSv3 developers—that `readdir` operations are often followed by `stat(2)` operations (often the result of users running `ls -l`); that is why NFSv3 implements a special protocol message called READDIRPLUS which combines directory reading with stat information [17]. Consequently, Reiserfs 4.0 does more work in `readdir`, but this initial effort improves subsequent `lookup` operations. Overall, this is a good trade-off for this workload: Reiserfs 4.0 used 60.6% less system time and I/O time than 3.6.

7.3.3 Ext3

Ext3 and most other file systems update their `atime` asynchronously, and release the locks right after initiating the write operation. The two graphs in Figure 7.13 show the `read` operation with and without `atime` updates. We can see that the profile for the `read` oper-

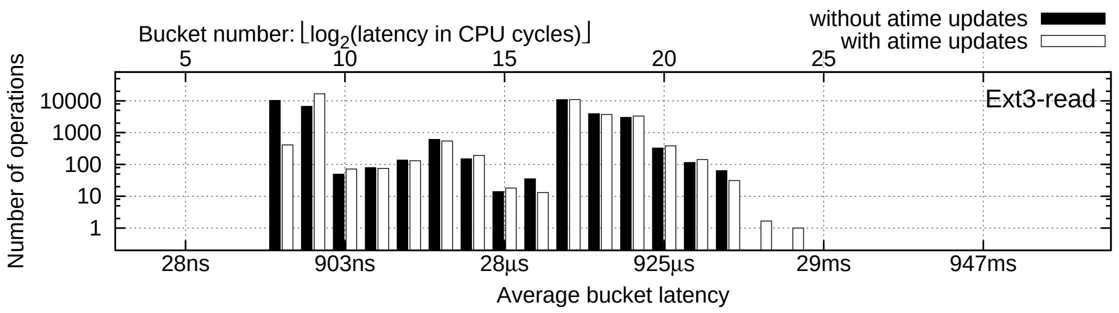


Figure 7.13: Effects of `atime` updates on Ext3.

ation with `atime` updates enabled on Ext3 is farther to the left than that of NTFS (bottom graph in Figure 7.6). This is because Ext3 keeps metadata close to its accompanying data, and so seeks are fairly short. Because of Ext3's improved locking policies and on-disk layout, its elapsed time overhead for updating the access times is only 2.8%.

7.4 Analyzing Network File Systems

We connected two identical machines (described in Section 6) with a 100Mbps Ethernet link and ran our *grep -r* workload on Windows with an NTFS drive shared over CIFS. Figure 7.14 shows a complete profile of CIFS under the *grep -r* workload. We found that the `findfirst` and `findnext` operations on the client had peaks that were farther to the right than any other operation (buckets 26–30 in the top two graphs of Figure 7.15). These two peaks alone account for 12% of the elapsed time, which was 170 seconds in total. `findfirst` searches for file names with a given pattern and returns all matching file names along with their associated metadata information. It also returns a cookie, which allows the caller to continue receiving matches by passing it to `findnext`.

By examining the peaks in other operations on the client and the corresponding requests on the server, we found that instances of an operation which fall into bucket 18 and higher (greater than 168 μs) involve interaction with the server, whereas buckets to the left of it were local to the client. All of the `findfirst` operations and the two rightmost peaks of the `findnext` operation here go through the server. Since CIFS is a modified version of the SMB protocol, we tried the same *grep -r* workload with the Windows server and a Linux client over SMB. The fact that in these cases we did not observe similar peaks, suggests that the high latencies were attributed to CIFS. Once we determined that the problem was due to some CIFS client-server interaction, we ran a packet sniffer on the network to investigate this further.

A timeline for a typical `findfirst` transaction between a Windows client and a Windows server explains the source of the problems, and is shown on the left-hand side of Figure 7.16. The client begins by sending a `findfirst` request containing the desired pattern to search for (*e.g.*, `C:\linux-2.6.11\*`). The server replies with file names that match this pattern and their associated metadata. Since the reply is too large for one TCP packet, it is split into three packets ("`FIND_FIRST` reply," "reply continuation 1," and "reply continuation 2"). The acknowledgment (ACK) for "reply continuation 1" is sent immediately, but the ACK for "reply continuation 2" is sent only after approximately 200 ms. This behavior is a *delayed ACK*: because TCP can send an ACK in the same packet as other data, it delays the ACK in the hope that it will soon need to send another packet to the same destination. Most implementations wait 200 ms for other data to be sent before sending an ACK on its own. Delaying an ACK is considered to be good behavior, but the Windows server does not continue to send data until it has received an ACK for everything until that point. This unnecessary synchronous behavior is what causes poor performance for the `findfirst` and `findnext` operations. After the server receives this ACK, it sends the client a "transact continuation" SMB packet, indicating that more data is arriving. This is followed by more pairs of TCP replies and ACKs, with similar delays.

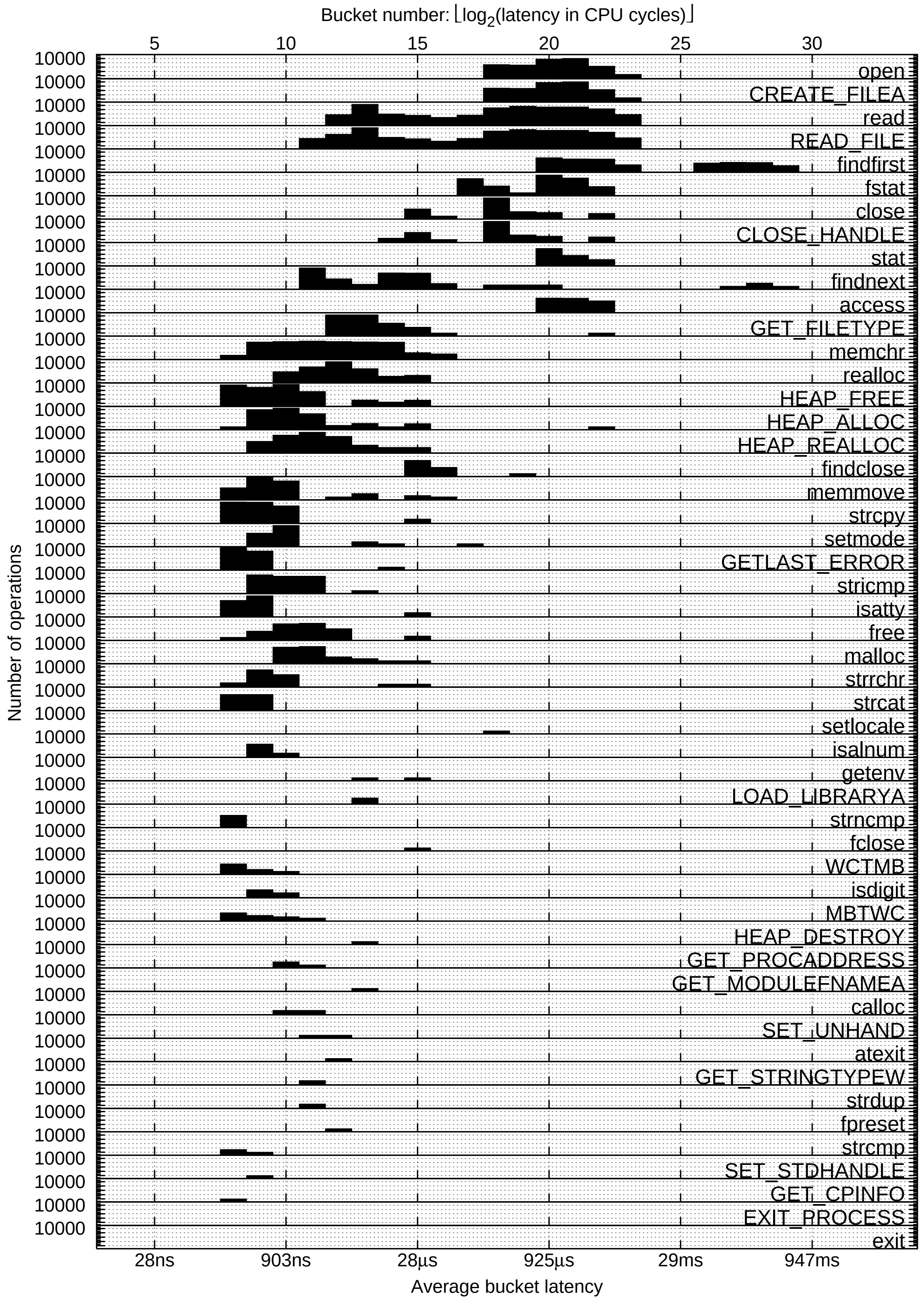


Figure 7.14: A complete profile of the *grep -r* workload on the Windows client over CIFS.

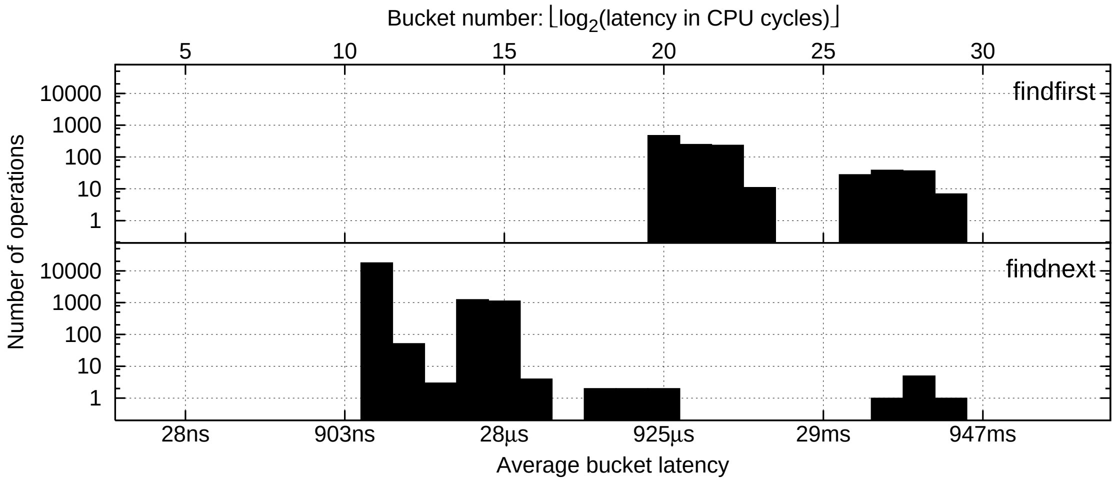


Figure 7.15: Windows CIFS client operations that sometimes take unusually long time to complete under the *grep -r* workload.

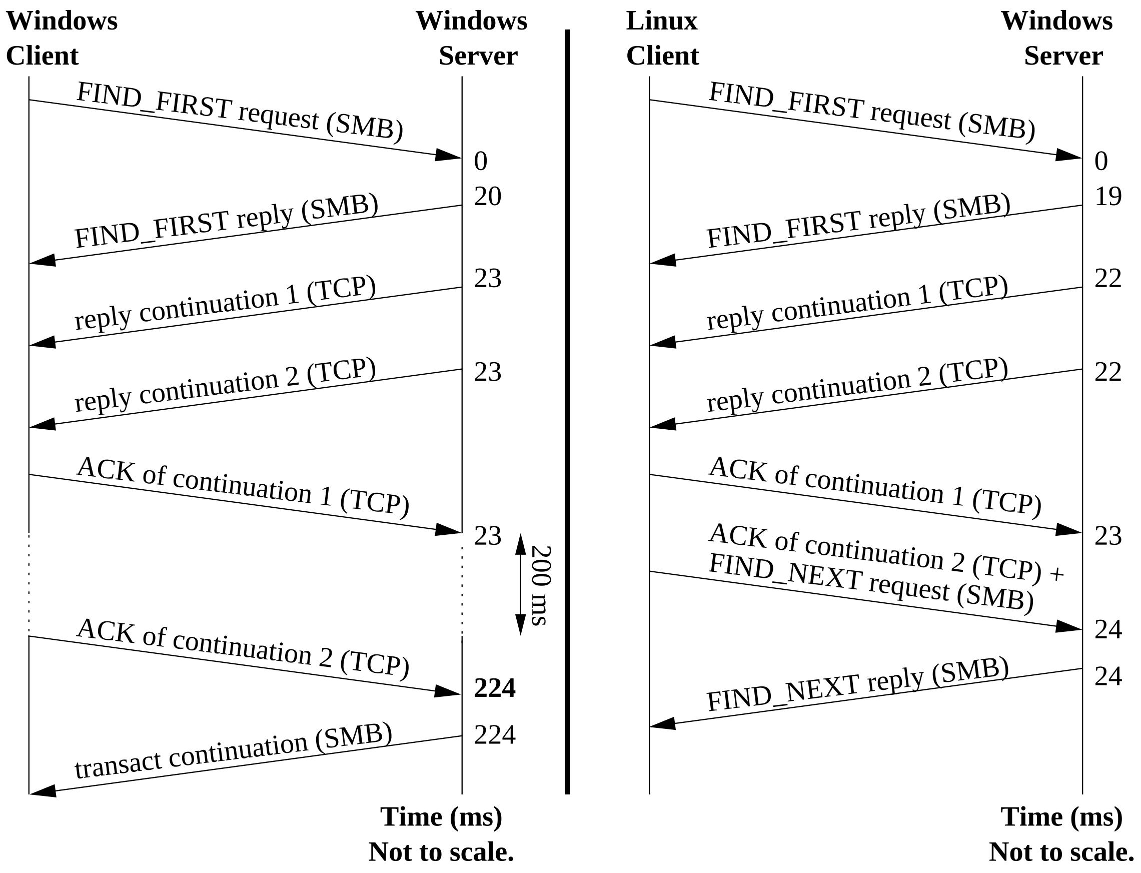

Windows Client
Windows Server
Linux Client
Windows Server
FIND_FIRST request (SMB)
0
FIND_FIRST reply (SMB)
20
reply continuation 1 (TCP)
23
reply continuation 2 (TCP)
23
ACK of continuation 1 (TCP)
23
200 ms
ACK of continuation 2 (TCP)
224
transact continuation (SMB)
224
FIND_FIRST request (SMB)
0
FIND_FIRST reply (SMB)
19
reply continuation 1 (TCP)
22
reply continuation 2 (TCP)
22
ACK of continuation 1 (TCP)
23
ACK of continuation 2 (TCP) + FIND_NEXT request (SMB)
24
FIND_NEXT reply (SMB)
24
Time (ms) Not to scale.
Time (ms) Not to scale.

Figure 7.16: Timelines depicting the messages involved in the handling of a `findfirst` request between Windows client and server over CIFS (left) and between a Linux client and a Windows server over SMB (right). Times are in milliseconds and are representative of typical latencies (not drawn to scale). Protocols are shown in parentheses.

The right-hand side of Figure 7.16 shows a similar timeline for a Linux client interacting with a Windows server over SMB. The behavior is identical, except that instead of sending a delayed ACK for “reply continuation 2,” Linux sends a `findnext` request immediately that also contains an ACK. This causes the server to return more entries immediately. We modified a Windows registry key to turn off delayed ACKs, and found that it improved elapsed time by 20%. This is not a solution to the problem, but a way to approximate potential performance benefits without waiting on ACKs.

7.5 Influence of Stackable File Systems

We used our method to evaluate the impact of file system stacking on the captured profile. Figure 7.17 shows the latency distribution of Base0fs, a thin passthrough stackable file system mounted over Ext2, and a vanilla Ext2 file system, both evaluated with the *grep -r* workload.

The stacking interface has a relatively small CPU overhead, which affects only the fastest buckets. Unfortunately, the overheads are different for different VFS operations. This can be explained by the differences in the way these operations are handled in stackable file systems. In particular, some operations are passed through with minimal changes, whereas others require the allocation of VFS objects such as inodes, dentries (directory entries), or memory pages. As we can see in Figure 7.17, Base0fs’s peaks are generally shifted to the right of Ext2’s peaks, demonstrating an overall overhead. The overheads of `open` and `lookup` exceed 4K CPU cycles, whereas `readdir` has an overhead below 1K CPU cycles.

VFS objects have different properties on the lower-level and the stackable file systems. For example, an encryption file system maintains cleartext names and data, but the lower file system maintains encrypted names and data [115]. Therefore, stackable file systems create copies of each lower-level object they encounter.

This behavior of stackable file systems adds overheads associated with data copying and causes distortions in the latencies of their read and write operations. For example, `read_page` is only invoked by the `read` operation if the page is not found in the cache. Therefore, only `read_page` operations that require disk accesses are captured and passed down. The `sync_page` operation is never invoked because pages associated with Base0fs inodes are never marked dirty.

Most importantly, duplicate copies of data pages effectively reduce the page cache size in half. This can result in serious performance overheads when a workload fits into the page cache but not into less than 50% of the page cache. Unfortunately, in Linux, each page cache object is linked with the corresponding inode and therefore the double representation of inodes implies double caching of data pages.

We found a relatively simple solution to the problem, which is not optimal [48] but allows us to profile the behavior of stackable file systems without the double caching of data. We use data pages associated with the upper inode for both the lower and upper file system layers. In particular, the data pages belong to the upper inode but are assigned to lower-level inodes for the short duration of the lower-level page-based operations. Here is an example of the modified `readpage` operation:

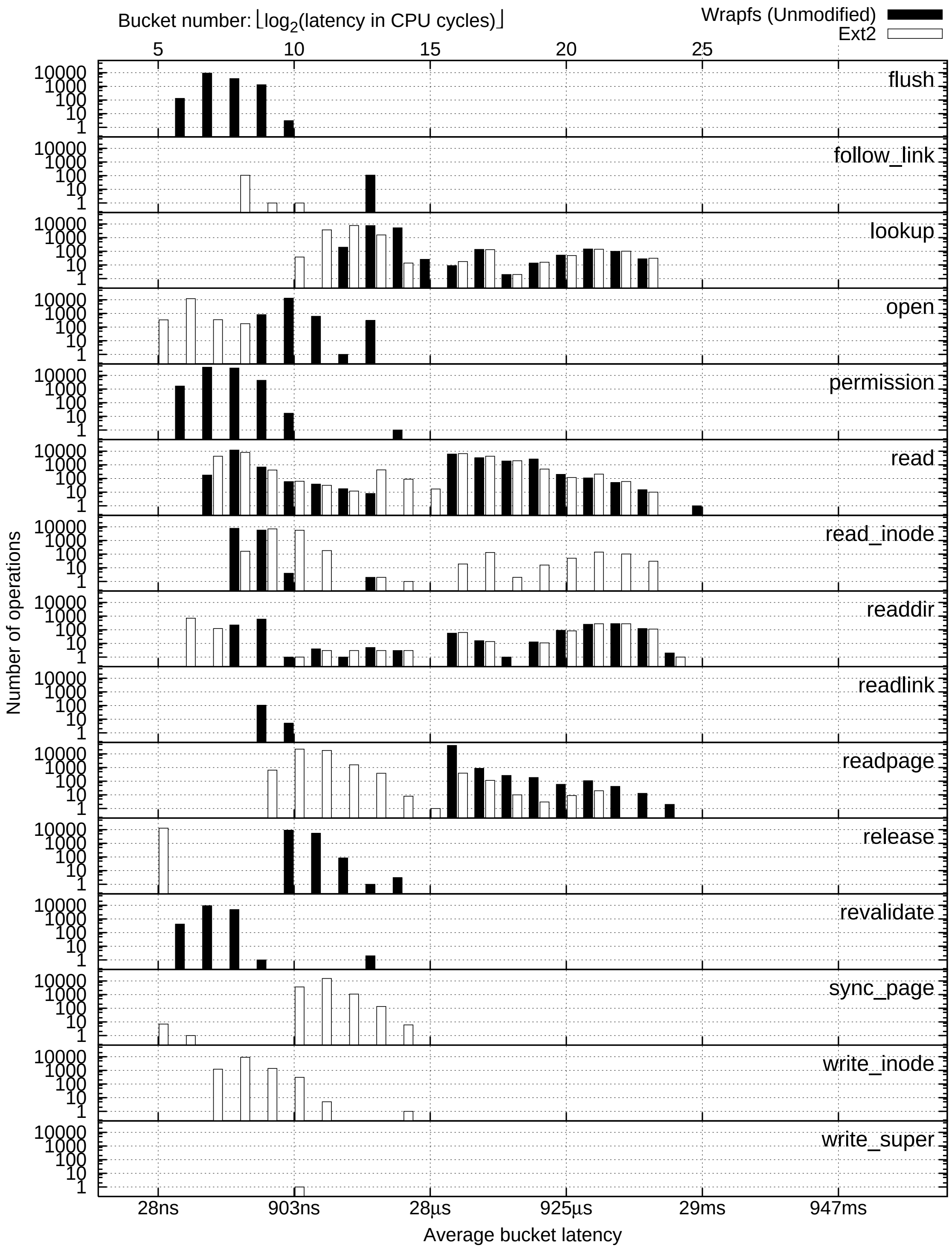


Figure 7.17: The distribution of operation latencies for an unmodified Base0fs mounted over Ext2, and for the Ext2 file system, under a *grep -r* workload. Profiles are sorted by operation names.

```
page->mapping = lower_inode->i_mapping;
err = lower_inode->i_mapping->a_ops->readpage(lower_file, page);
page->mapping = upper_inode->i_mapping;
```

The resulting code allows profiling of page-based operations, and also eliminates data copying and double caching. We analyzed the Linux kernel functions that directly or indirectly use inode and cache page connectivity and found that in all these cases, the above modification works correctly. We tested the resulting stackable file system on a single-CPU and multi-CPU machines under the compile and Postmark workloads.

As can be seen in Figure 7.18, the no-double-caching patch described above decreases the system time compared to the original Base0fs, has a cache size that is the same as Ext2, and also prevents double caching from influencing the cache-page related operations. In particular, the profile of the modified file system has virtually no difference from the plain Ext2 file system for the `read`, `read_page`, and `sync_page` operations.

Overall, a stackable file system does influence the profile of the lower-level file system, but it still can be used to profile a subset of VFS operations when the source code is not available. Even for operations whose latency values are affected by the stackable file system, the peaks and overall structure of the profile usually remain the same. Therefore, key file system and workload characteristics can be collected.

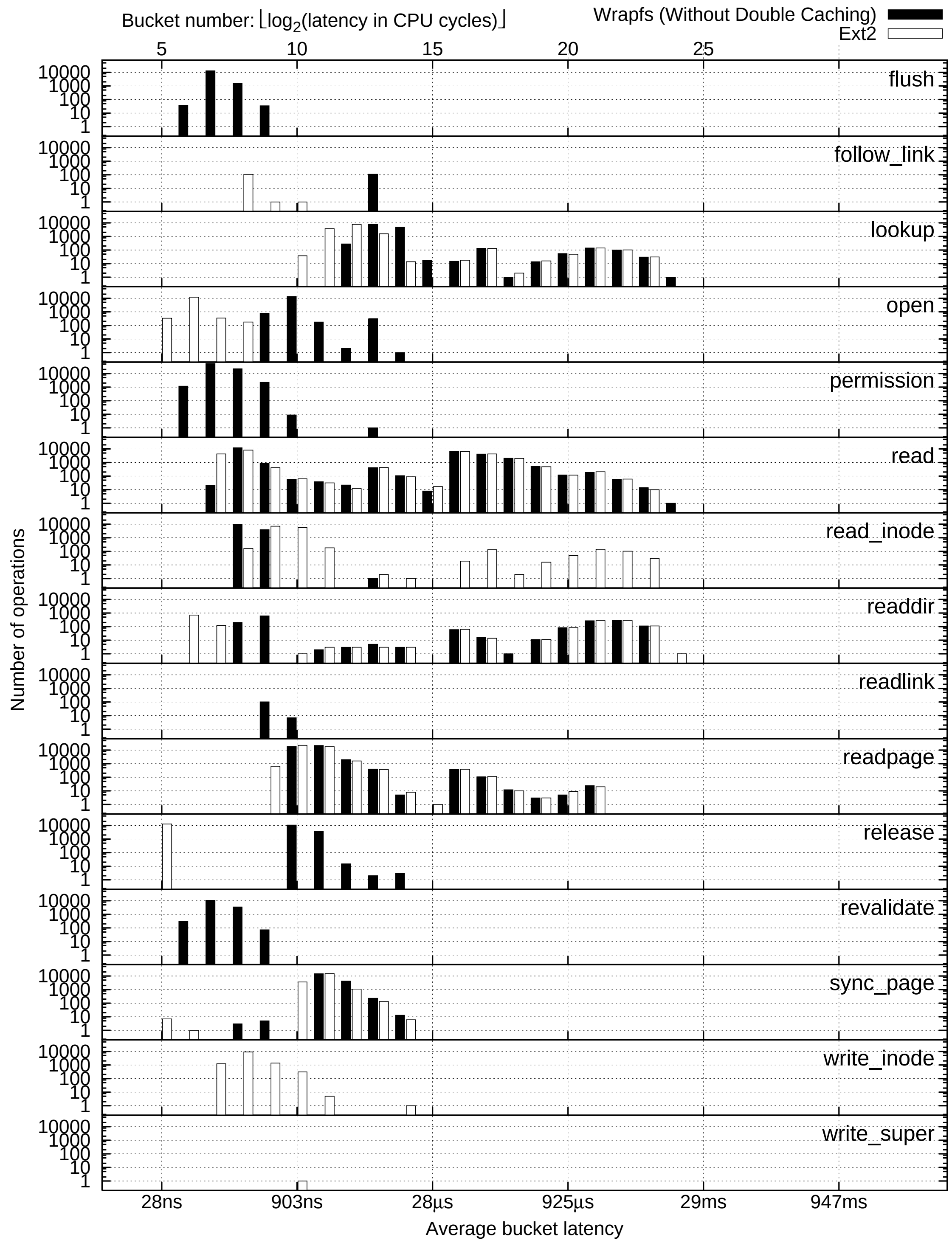


Figure 7.18: The distribution of operation latencies for Base0fs without double caching, mounted over Ext2, and for the Ext2 file system, under a *grep -r* workload. Profiles are sorted by operation names.

Chapter 8

Using FSprof without Buckets

In Chapter 7 we have considered several examples of file system profiling. However, FSprof also allows to collect simpler aggregate statistics about the file system behavior. Below we will consider two examples: (1) we will characterize several compilation processes by looking at the aggregate number of file system operations and their aggregate latency; (2) we will use sampling of the number of operation invocations to characterize the quality of the trace replaying.

8.1 Workload Characterization

Compile benchmarks are often used to evaluate file system behavior [54]. We show that even seemingly similar mixes of source files generate considerably different VFS operation mixes. Therefore, results obtained during different compile benchmarks cannot be fairly compared with each other.

We profiled the build process of three packages commonly used as compile benchmarks: (1) SSH 2.1.0, (2) Am-utils 6.1b3, and (3) the Linux 2.4.20 kernel with the default configuration. Table 8.1 shows the general characteristics of the packages. The build process of these packages consists of a preparation and a compilation phase. The preparation phase consists of running GNU `configure` scripts for SSH and Am-utils, and running "`make defconfig dep`" for the Linux kernel. We analyzed the preparation and compilation phases separately, as well as together (which we call a "whole build"). Before the preparation and compilation phases, we unmounted the file system in question, purged the caches using our custom *chill* program, and finally remounted the tested file systems. For the full build, we performed this cache-purging sequence only before the preparation phase. This means that the number of invocations of every operation in the case of full build is the sum of the invocations of the same operation during the preparation and compilation stages. However, the full-build delays are not the sum of the preparation and compilation delays, because we did not purge the caches between phases for the full build. This way it was possible to compare the compilation profiles separately. The delays of the compilation phase, as a part of the build process, can be obtained by subtracting the preparation phase delays from the full build delays.

Figure 8.1 shows the distribution of the total number of invocations and the total delay of all the Ext2 VFS operations used during the build process of SSH, Am-utils, and the Linux kernel. Note that each of the three graphs uses different scales for the number of operations and the total delay.

Figures 8.1(a) and 8.1(b) show that even though the SSH and Am-utils build process sequence, source-file structure, and total sizes appear to be similar, their operation mixes are quite different; moreover, the fact that SSH has nearly three times the lines of code of Am-utils is also not apparent from analyzing the figures. In particular, the preparation phase dominates in the case of Am-utils whereas the compilation phase dominates the SSH build. More importantly, an Am-utils build writes more than it reads, whereas the SSH build reads more than it writes: the ratio of the number of reads to the number of writes is $\frac{26{,}458}{35{,}060} = 0.75$ for Am-utils and $\frac{42{,}381}{33{,}108} = 1.28$ for SSH. This can result in performance differences for read-oriented or write-oriented file systems.

Not surprisingly, the kernel build process's profile differs from both SSH and Am-utils. As can be seen in Figure 8.1(c), both of the kernel build phases are strongly read biased. Another interesting observation is that the kernel build phase populates the cache with most of the meta-data and data early on. Figures 3.5 and 3.6 on page 15 show the profile of the `lookup` operation during the kernel build process, where we see that the preparation phase causes the vast majority of `lookup`s that incur disk I/O.

Table 8.2 shows the `lookup` operation's latency peaks for different build processes. We can see that the Am-utils build process has the least cache misses. Therefore, it has the minimal average `lookup` operation delay (the only metric measurable by some other kernel profilers such as kernprof [94]). SSH's average `lookup` delay is only slightly higher because the higher percentage of misses is compensated by the high fraction of disk oper-

	Am-utils	**SSH**	**Linux Kernel**
Directories	25	54	608
Files	430	637	11,352
Lines of Code	61,513	170,239	4,490,349
Code Size (Bytes)	1,691,153	5,313,257	126,735,431
Total Size (Bytes)	8,441,856	9,068,544	174,755,840

Table 8.1: Compile benchmarks' characteristics.

	Am-utils	**SSH**	**Linux Kernel**
Fastest peak	1,817	2,848	10,423
Middle peak	9	48	79
Slowest peak	25	32	227
Page cache misses (%)	1.9	2.7	2.9
Average delay	83,022	95,697	186,672

Table 8.2: Distribution of the Ext2 `lookup` operations among the three peaks shown in Figure 3.5 on page 15 representing a page cache hit (buckets 10–15), a disk buffer cache hit (buckets 15–19), and a long disk rotation or a head seek (buckets 20 and above). The page cache miss ratio is calculated as the sum of the operations in the middle and slowest peaks over the total number of operations.

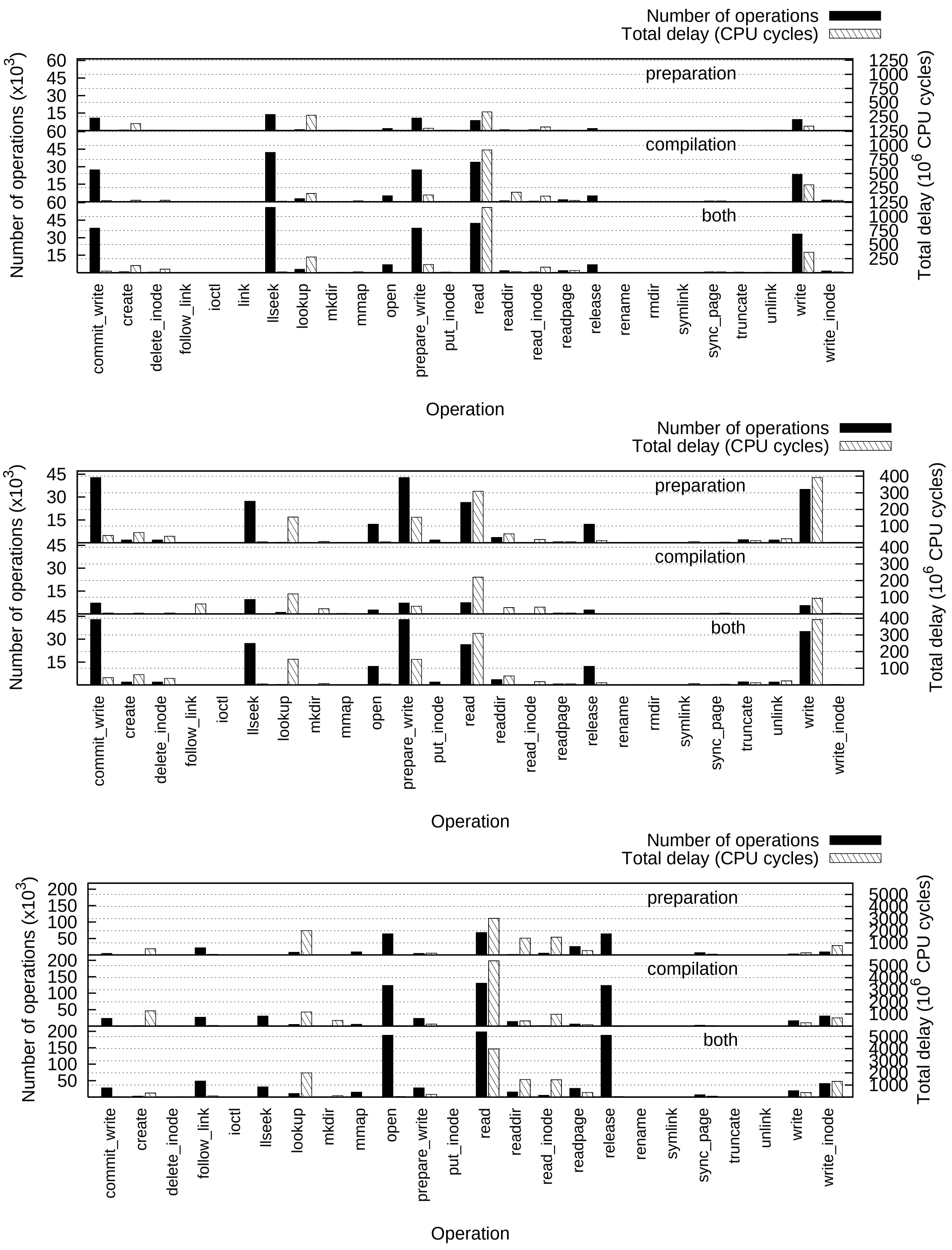


Figure 8.1: Operation mixes during a compilation as seen by the Ext2 file system. From top to bottom: (a) SSH 2.1.0, (b) Am-utils 6.1b3, and (c) Linux 2.4.20. Note that each plot uses a different non-logarithmic scale.

ations that do not require long disk-head seeks. The Linux kernel build process incurs a higher proportion of buffer cache misses and at the same time has a high proportion of the long disk requests. Therefore, its average `lookup` delay is the highest.

We see that not only can we not directly compare different compile benchmarks, but we can also not extrapolate results based on summary information about the source files such as the package size, number of lines of code, etc. The order and type of file-system operations can seriously change the delay of VFS operations, and hence the benchmark's CPU and I/O times.

8.2 Quality of Replaying Analysis

Replayfs is a VFS-level file system trace replayer that we developed [55]. We used FoSgen and FiST latency profiling extension for its evaluation. In particular, we collected sampled profiles during trace capture and trace replaying directly at the file system level (albeit we did not use the latency values in the profiles that we collected). We captured profiles to calculate the timing error of our trace replaying as a function of the elapsed time. Figure 8.2 shows the timing error dependence while replaying the Am-utils package [80] compilation trace. The corresponding VFS operation invocation rates are shown in Figure 8.3.

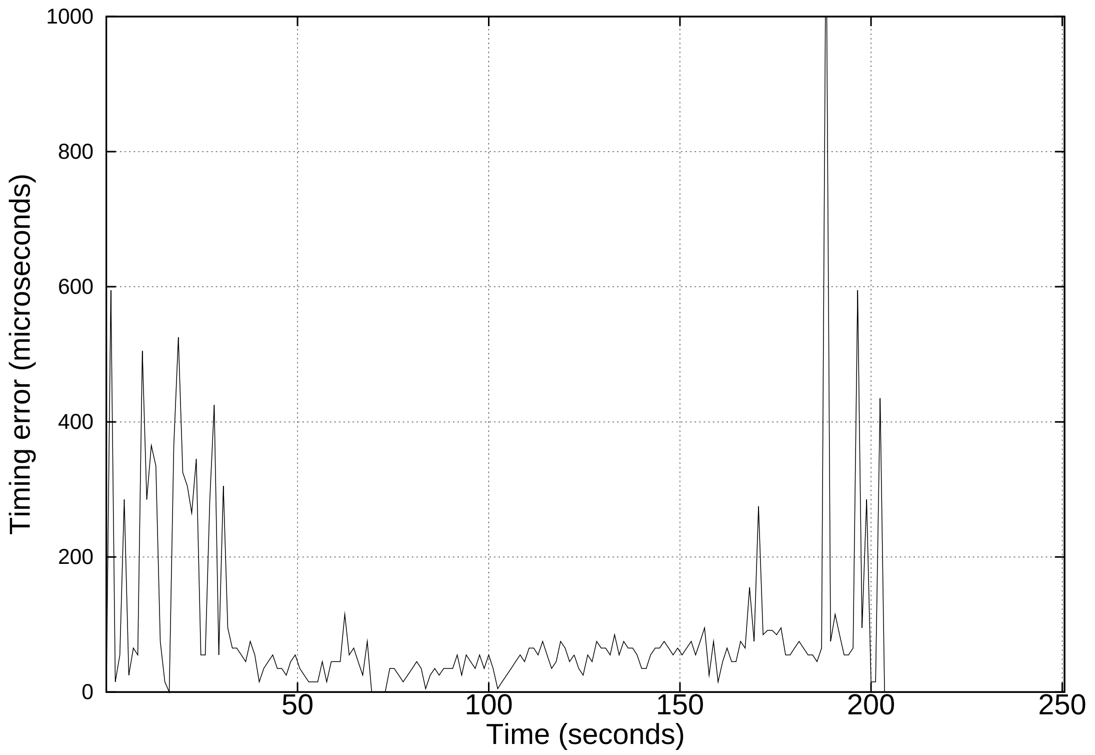


Figure 8.2: The difference between tracing and replaying rates.

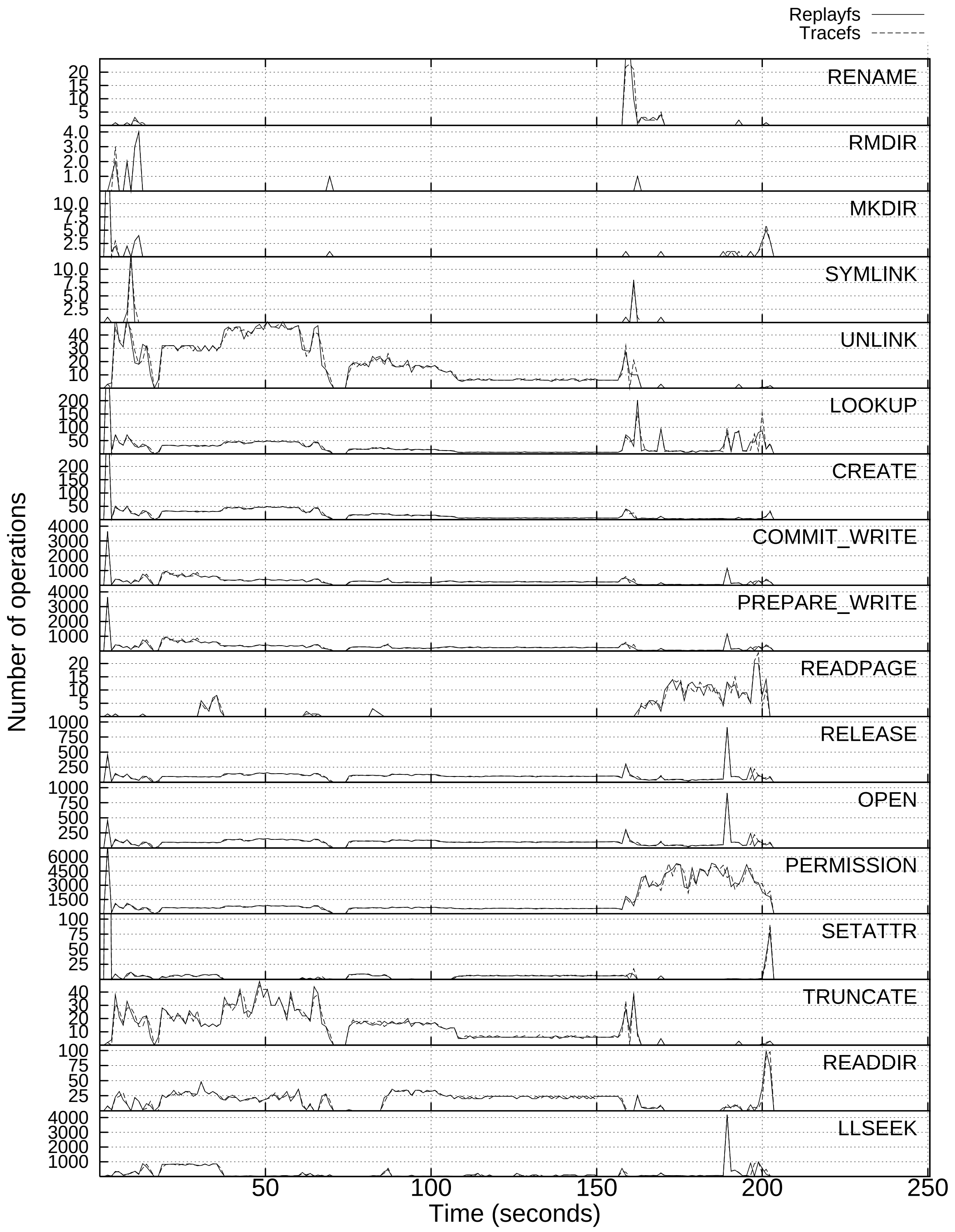


Figure 8.3: A comparison of traced and replayed rates.

Chapter 9

Case Study: RAIF

In Chapter 7 we presented several profiling examples of file systems developed earlier. However, the profiling method that we developed is especially useful for file system development. In our laboratory we are constantly developing new file systems and design other OS components. We found it necessary and most convenient to use latency profiling while working on *all* five of the following file systems: Replayfs [55], RAIF [52], UnionFS [109], secure deletion Ext3 extensions [51, 57], ACIDFS [108], as well as Kefence [50] and some other unpublished projects we were working on during the past year. In this chapter we describe a typical example of the latency analysis usage to optimize a file system during its development phase.

9.1 Redundant Array of Independent Filesystems

Redundant Array of Independent Filesystems (RAIF) is the first RAID-like storage system designed at the file system level that imposes virtually no restrictions on the underlying stores and allows per-file storage policy configuration.

For many years, grouping hard disks together to form RAIDs has been considered a key technique for improving storage survivability and increasing data access bandwidth [77]. However, most of the existing hardware and software RAID implementations require that the storage devices underneath be of one type. For example, several network stores and a local hard drive cannot be seamlessly used to create a RAID. RAID configurations are fixed and are the same for all the files because hardware and software RAIDs operate at the data-block level, where high level meta-information is not available. This results in inefficient storage utilization when important data is stored with the same redundancy level as less-important data. Other common RAID limitations are related to long-term maintenance. For example, data recovery is slow and may require a restart if interrupted.

There are several implementations of RAID-like file server systems that combine network servers [4, 38], or even combine network and local drives [35]. However, past systems targeted some particular usage scenario and had a fixed architecture. Inflexibilities introduced at design time often result in sub-optimal resource utilization. RAIF leverages the RAID design principles at the file system level, and offers better configurability, flexibility and ease of use in managing data security, survivability, and performance.

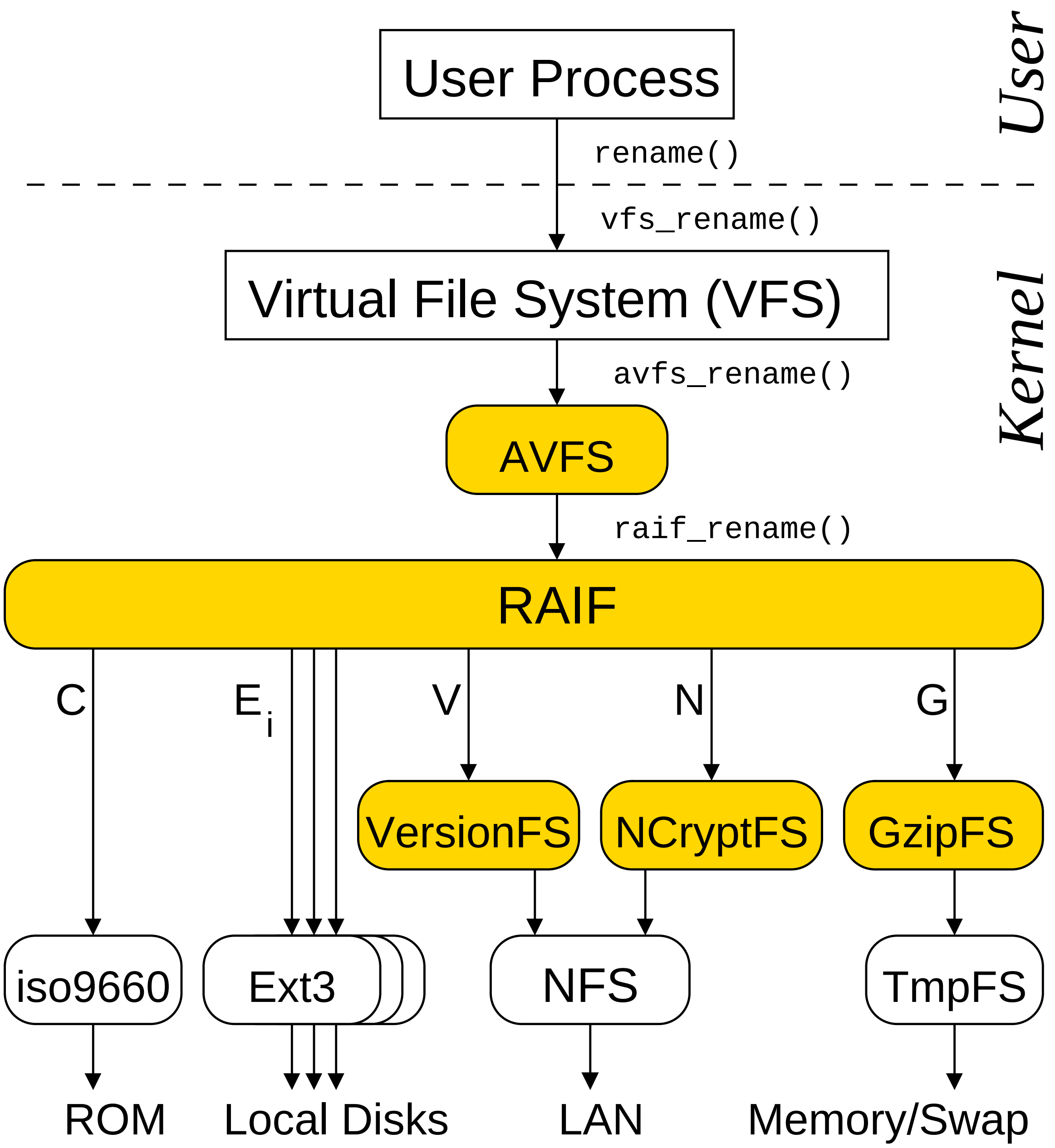


Figure 9.1: A possible combination of RAIF fan-out stacking and other file systems stacked linearly. Letters below RAIF denote the branch labels.

File system development is as difficult as any OS code development: the kernel is a complex environment that is unforgiving to mistakes. Developers have always sought methods to speed up OS code development and techniques that will result in having to write less code. RAIF is a fan-out RAID-like stackable file system. Stackable file systems are a useful and well-known technique for adding functionality to existing file systems [115]. They allow for the incremental addition of features and can be dynamically loaded as external kernel modules. Stackable file systems overlay another *lower* file system, intercept file system events and data bound from user processes to the lower file system, and in turn manipulate the lower file system's operations and data, and pass the changed ones down to the lower file system. Developing stackable file systems is easier than developing native file systems. For example, a basic stackable encryption file system need only intercept data buffers that come from the `write` system call and encrypt those buffers before passing them to the lower file system; similarly, buffers are intercepted in `read` and decrypted before being returned to user processes. Past stackable file systems developed by us and others have assumed a simple one-to-one mapping: the stackable file system was layered on top of one lower directory on a single file system. A different class of file systems that use a one-to-many mapping (a *fan-out*) has been previously suggested [39, 87] and was recently included in the FiST [110, 115] templates.

RAIF derives its usefulness from three main features: flexibility of configurations, access to high-level information, and easier administration.

1. As a stackable file system RAIF can be mounted over any combination of lower file systems. For example, it can be mounted over several network file systems like NFS and Samba, AFS distributed file systems [42], and local file systems at the same time; in one such configuration, fast local branches may be used for parity in a RAID4-like configuration. If the network mounts are slow, we could explore techniques such as data recovery from parity even if nothing has failed, because it may be faster to reconstruct the data using parity than to wait for the last data block to arrive. Stackable file systems can be mounted on top of each other. Examples of existing stackable file systems are: an encryption [112], data-integrity verification [59], an antivirus [73], and a compression file system [114]. These file systems can be mounted over RAIF as well as below it over only some slow or untrusted branches. Figure 9.1 shows an example RAIF mount configuration.

2. RAIF operates at the file system level and has access to high-level file system meta-data that is not available to traditional RAIDs operating at the block level. This meta-data information can be used to store files of different types using different RAID levels, optimizing data placement and readahead algorithms to take into account varying access patterns for different file types. For example, RAIF can on one hand stripe large multimedia files across different branches for performance, but use two parity pages for important financial data files that must be available even in the face of two failures. Dynamic RAIF-level migration offers additional benefits.

3. Third, administration is easier because files are stored on ordinary unmodified lower-level file systems. Therefore, the size of these lower file systems can be changed,

they can be easily backed up using standard software. The data is easier to recover in the case of failure because it is stored in a more accessible format.

9.2 Benchmarking with Postmark

We have evaluated RAIF performance on a DELL PowerEdge server with a 2.8GHz CPU and 2GB of RAM. We used an external disk array consisting of four Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disks (same as in Section 6). We used the Auto-pilot benchmarking suite [111] to run all of the benchmarks. The lower-level file systems were remounted before every benchmark run to purge the page cache. We ran each test at least ten times and used the Student-t distribution to compute 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. In each case the half-widths of the confidence intervals were less than 5% of the mean.

For the remainder of the evaluation, we use RAIFL-B to refer to RAIF level L with B branches. We use RAIDL-B to refer to the Linux RAID driver, where L and B have the same meaning as in RAIF.

Postmark v1.5 [60] simulates the operation of electronic mail servers. It performs a series of file appends, reads, creations, and deletions, showing how RAIF might behave in an I/O-intensive environment. We chose a Postmark configuration to stress the I/O: it creates 60,000 files, between 512–10K bytes, and performs 600,000 transactions. All operations were selected with equal probability. Note that this Postmark configuration is different from the one described in Section 6 because of the faster CPU that we used here. Figure 9.2 shows the Postmark configuration file that we used. Every other Postmark's operation is either a `create` or an `unlink`. Due to VFS restrictions, these RAIF operations are executed sequentially on lower branches and are CPU-intensive. This makes Postmark a challenging benchmark for RAIF.

We ran Postmark for 2, 3, and 4 branches under RAID and RAIF levels 0, 1, 4, and 5.

```
set size 512 10240
set number 60000
set seed 42
set transactions 600000
set location /n/test/raif
set subdirectories 600
set read 4096
set write 4096
set buffering false
set bias read 5
set bias create 5
```

Figure 9.2: Postmark configuration that we used for RAIF benchmarking.

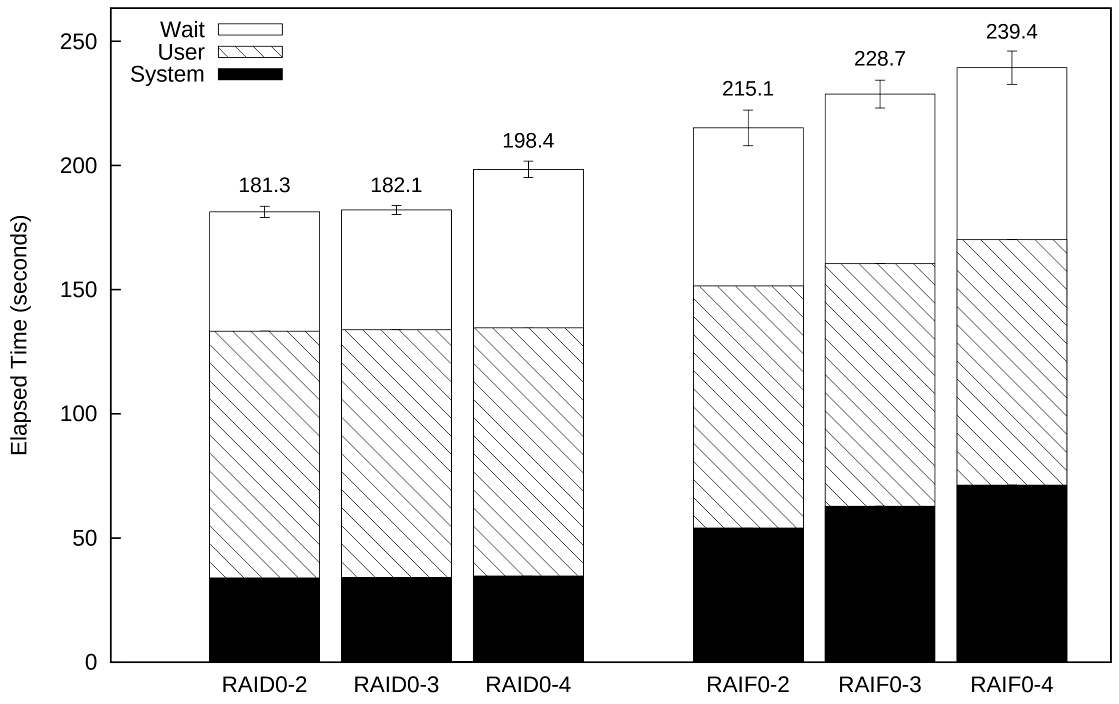


Figure 9.3: Postmark results for RAID0 and RAIF0 with varying number of branches.

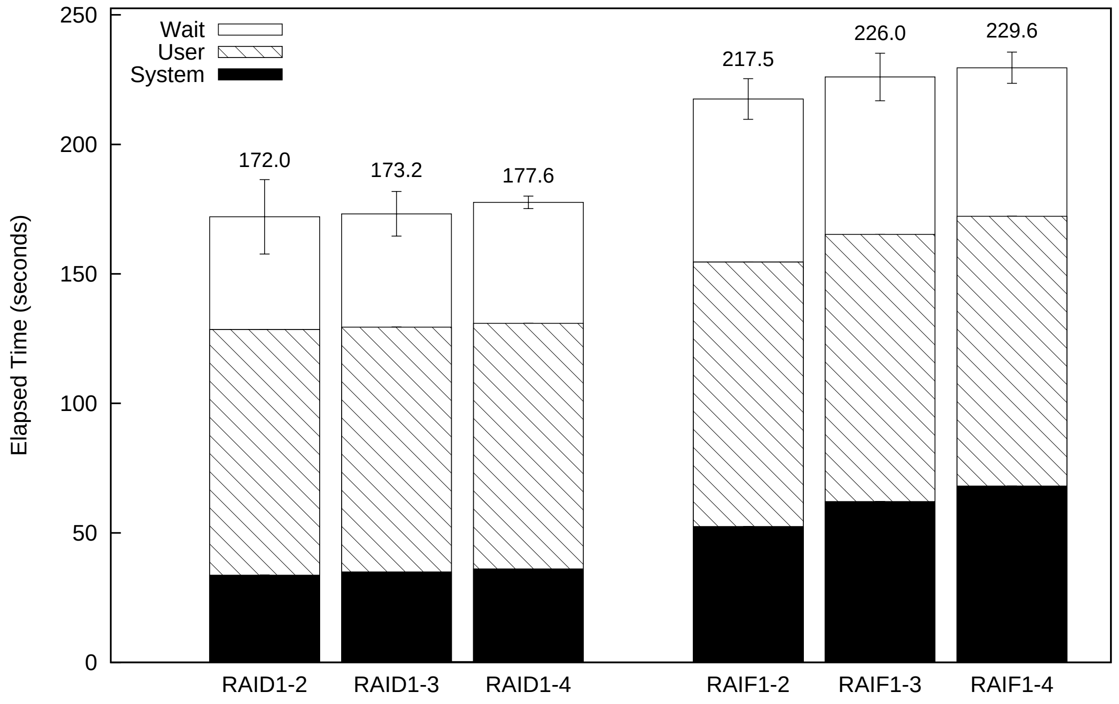


Figure 9.4: Postmark results for RAID1 and RAIF1 with varying number of branches.

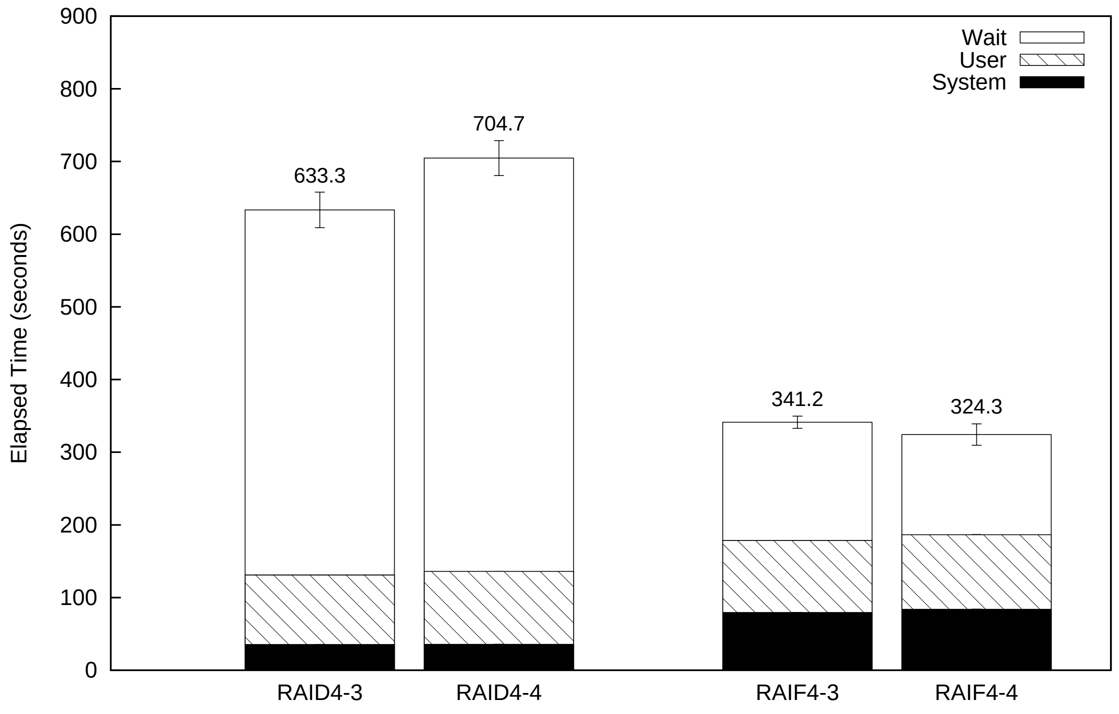


Figure 9.5: Postmark results for RAID4 and RAIF4 with varying number of branches.

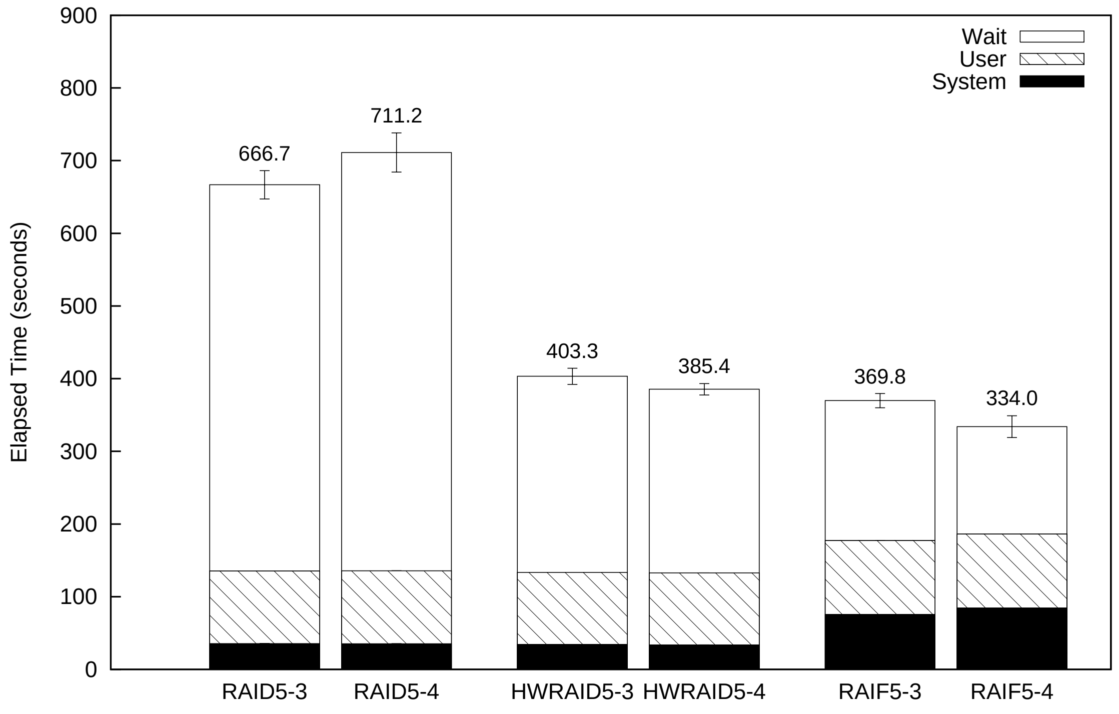


Figure 9.6: Postmark results for RAID5 and RAIF5 with varying number of branches.

9.2.1 RAIF0

As we can see from Figure 9.3, RAIF0-2 was 18.6% slower than RAID0-2, due to a 59.1% increase in system time and a 32.4% increase in wait time. The performance degraded slightly when more branches were added (overheads of 25.6% and 20.7% for three and four branches, respectively). This was due to the increased system time associated with extra branches.

9.2.2 RAIF1

Figure 9.4 shows the benchmarking results for RAID1 and RAIF1. The results for RAIF1 were similar to those for RAIF0, with similar increases to system time overheads as more branches were added. In this case, the elapsed time overheads were 26.4%, 30.5%, and 29.3% for configurations with 2, 3, and 4 branches, respectively. For RAIF1, we release cache pages of lower file systems after all write operations. This allowed us to decrease RAIF1-3 and RAIF1-4 overheads by approximately ten times.

9.2.3 RAIF4

Figure 9.5 shows the benchmarking results for RAID4 and RAIF4. Whereas the system time of RAIF4 was higher than RAID4, the wait time was significantly lower, resulting in overall better performance. The system time of RAIF4-3 was 2.2 times that of RAID4-3, but it had 67.6% less wait time, resulting in a 46.1% improvement. Similarly, the system time of RAIF4-4 was 2.3 times that of RAID4-4, but the wait time was reduced by 75.8%, resulting in an overall improvement of 54.0%.

9.2.4 RAIF5

Figure 9.6 shows the benchmarking results for RAID5, a hardware RAID card (HWRAID), and RAIF5. RAID5 and RAIF5 performance was similar to RAID4 and RAIF4. For RAIF5-3, system time was 2.1 times that of RAID5-3, wait time was 63.8% lower, and there was an overall improvement of 44.5%. The system time of RAIF5-4 was 2.4 times that of RAID5-4, the wait time improved by 74.3%, and the elapsed time improved by 53.0%.

We also benchmarked one of the hardware implementations of RAID5, and RAIF5 was faster than that implementation as well. In particular, we benchmarked the Adaptec 2120S SCSI RAID card with an Intel 80302 processor and 64MB of RAM. The hardware and driver-level implementations had similar system time overheads. RAIF5-3 had 28.6% less wait time than HWRAID5-3, and was 8.3% faster overall. RAIF5-4 had 41.5% wait time improvement over HWRAID5-3, and the elapsed time improved by 13.3%.

To understand why RAIF4 and RAIF5 performed so well under the Postmark workload we collected several latency profiles, which we will describe next.

9.3 RAIF5 Profiling

First, we profiled the Ext2 file systems mounted below RAIF to verify that the results are not caused by possible bugs in RAIF which could be related to non-equal load distribution. We compared these profiles using our analysis automation script (using the `TOTOPS` method). We verified that RAIF distributes requests to lower branches equally with only a 5% deviation.

Second, we profiled Ext2 mounted over the Linux RAID5 driver and Ext2 mounted below RAIF5, as shown in Figures 9.7 and 9.8 respectively. In both cases these systems operated over four disks. In the figures, all operations are sorted based on their total latencies. As we can see, all read-related operations complete quickly from the caches and do not require waiting for the disk accesses. However, `write` operation takes a lot of time in both cases.

Using the `TOTLAT` method we can see that `writepages` method took the longest total time for Ext2 mounted over RAID5. Both the `writepages` and the `write_inode` operations are called asynchronously by the kernel and both of them result in block-level writes. Not surprizingly, both of them are similarly processed by the block-level RAID5 driver. The rightmost peak that we can see in Figure 9.7 on the `writepages` profile (buckets 20–28) and the peak on the `write_inode` profile (buckets 22–27) are caused by the long disk head seeks. In particular, the Linux RAID driver experienced 6,669 long disk head seeks while writing dirty buffers to the disk. RAIF, on the other hand, waited for only 1,664 disk head seek operations (buckets 20–26). Also, these seek operations were much shorter than in the case of the RAID driver. This is because RAIF operates above file system caches and parity pages are cached as normal data pages. Linux RAID, however, operates with buffers logically below file system caches. This means that the Linux RAID driver has to read existing parity pages from the disk for partial-block write operations.

Finally, we profiled Ext2 using the Adaptec 2120S RAID5 controller as shown in Figure 9.9. In this case, the `writepages` operation completes quickly. This is because our hardware RAID5 controller cached requests using its 64MB-big cache and returned control back to the main system without waiting for the parity calculation and issuing disk read and write requests. However, sometimes this cache was not enough and some `write_inode` operations were performed synchronously. In particular, just 60 such requests (buckets 26–29) took about 20 seconds of time.

Overall, we can see that RAIF performs well for write-oriented workloads because it efficiently caches both data and parity pages using all available memory in the system. Driver-level RAID, however, only caches data pages and must read many parity pages from the disk frequently, incurring long disk head seeks. Entry-level hardware RAID controllers offload some of the CPU processing and use their memory to cache both data and parity. However, the on-board CPUs of such cards are relatively slow and their amount of memory is small. Therefore, we can see that RAIF is a cost effective solution that allows us to utilize the CPU time and all system memory on-demand and leave it for other tasks when they are not needed.

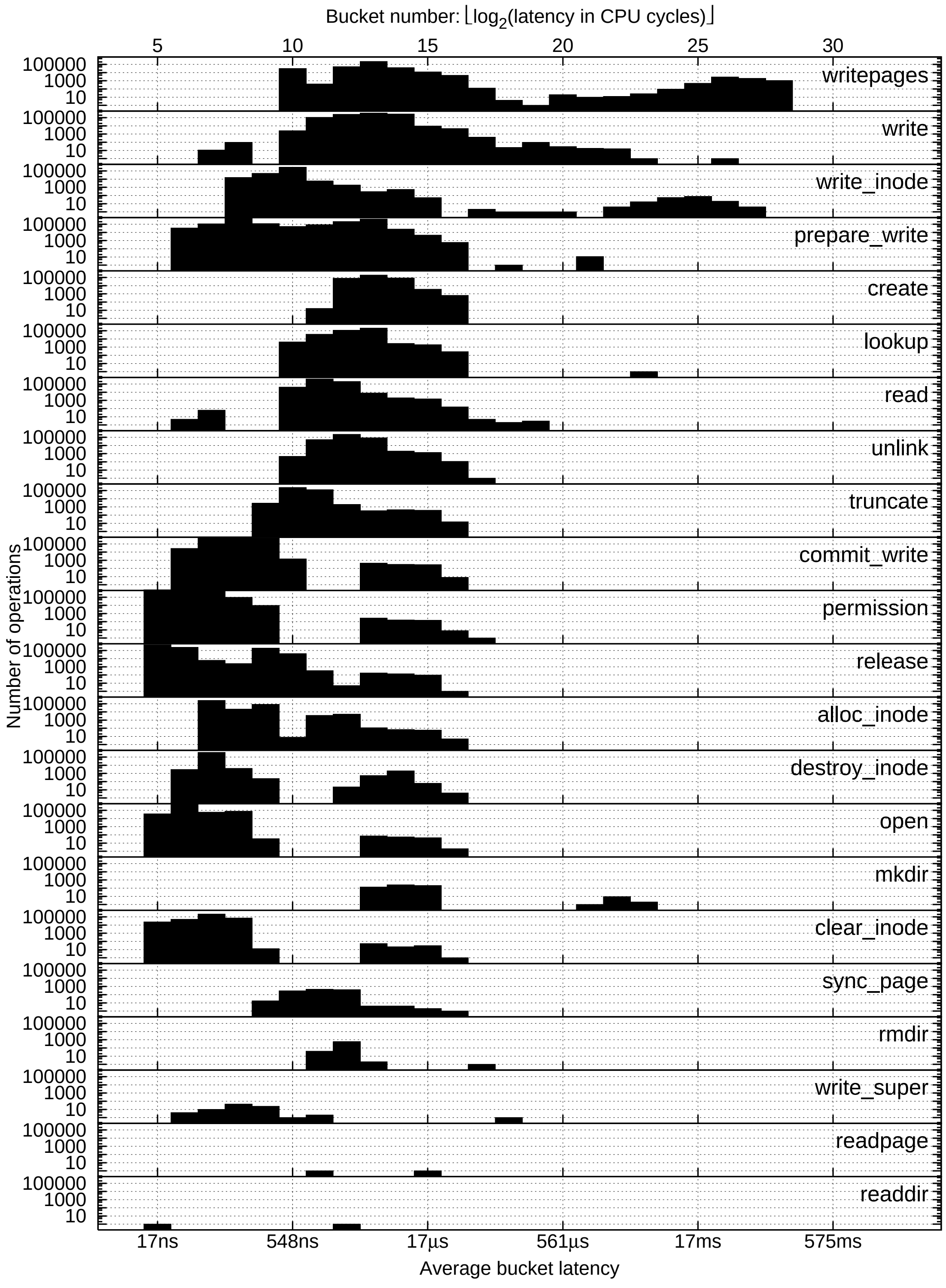


Figure 9.7: The profile of Ext2 mounted over the Linux RAID5 driver with four disks using the Postmark workload.

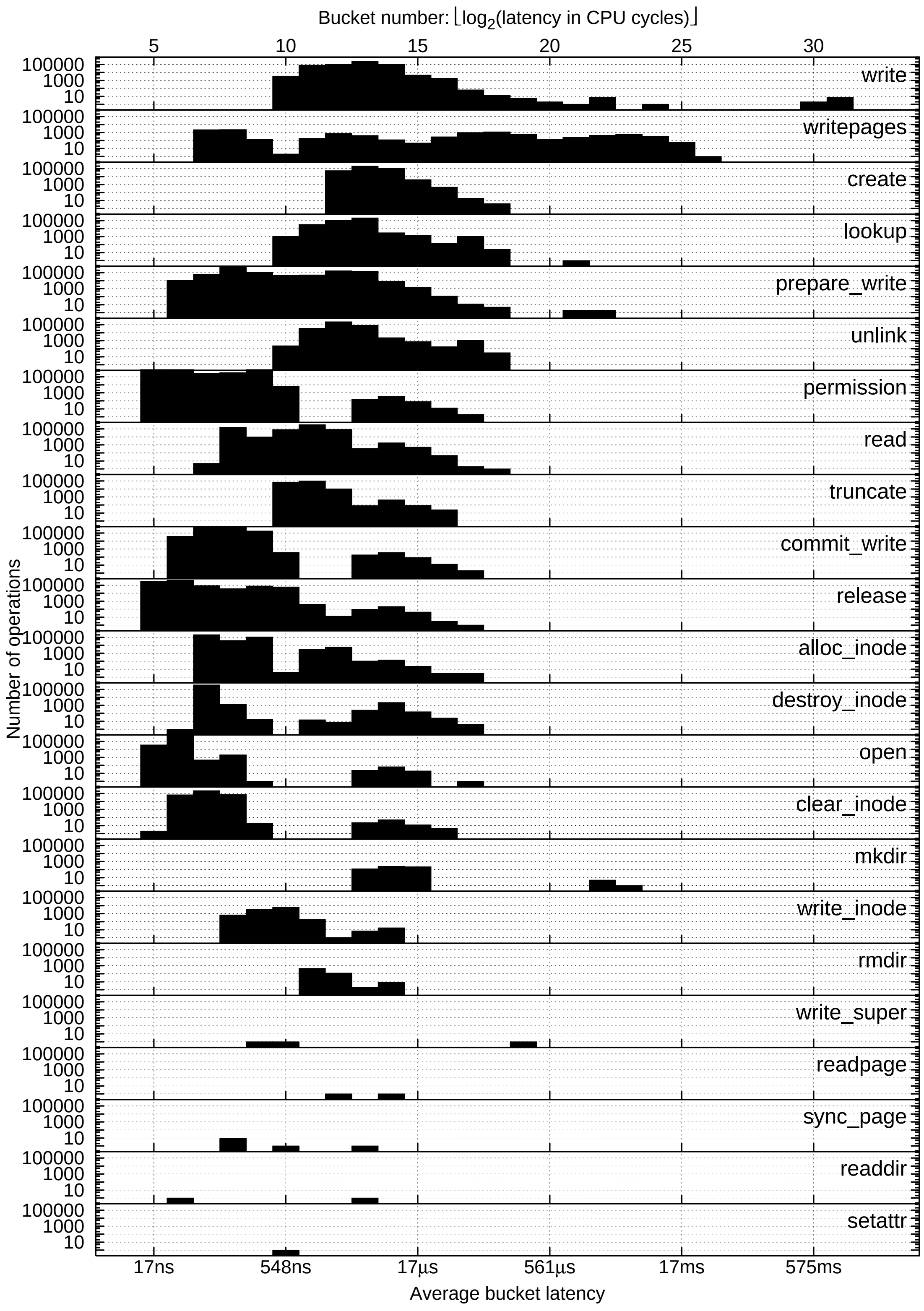


Figure 9.8: The profile of Ext2 mounted under RAIF5 using the Postmark workload.

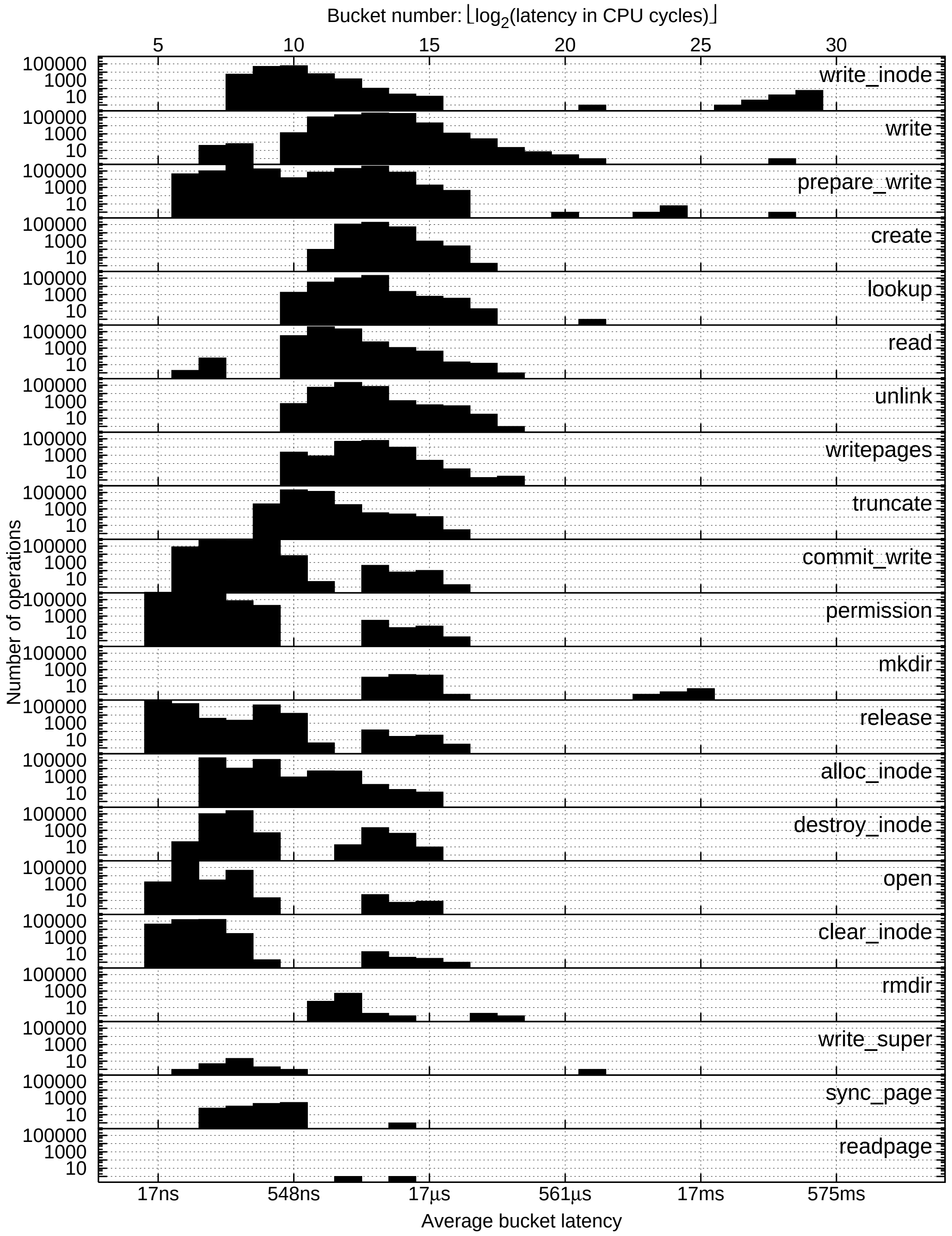


Figure 9.9: The profile of Ext2 using the Adaptec 2120S hardware RAID5 controller with four disks using the Postmark workload.

9.4 Prior and Related Work

Preliminary description of the RAIF architecture was published in [52]. Since then we changed the RAIF architecture in may ways. The major changes are related to storing per-file information, introduction of the policy-based management, and the re-architecture of the page-cache–related operations to improve performance.

9.4.1 Fan-Out Stackable File Systems

A class of stackable file systems known as *fan-out* file systems mount on top of more than one file system to provide useful functionality [39, 87]. However, so far the most common application of fan-out has been unification [9, 40, 63, 79, 109]. Unification file systems present users with a merged view of files stored on several lower branches. RAIF is a stackable fan-out file system that can mount on top of several underlying file systems to provide RAID-like functionality.

9.4.2 Block-Level RAID

Replication and striping of data (possibly combined with the use of error-correction codes) has been commonly used for decades. It is a common way to improve data survivability and performance on homogeneous [77] and heterogeneous [26] configurations of local hard drives. Modern block-level virtualization systems [43] rely on Storage Area Network (SAN) protocols such as iSCSI [3] to support remote block devices. The idea of using different RAID levels for different data access patterns at the block level was used in several projects at the driver [35] and hardware [106] levels. However, the lack of higher-level information forced the developers to make decisions based either on statistical information or semantics of particular file systems [95]. Exposed RAID (E$\times$RAID [28]) reveals information about parallelism and failure isolation boundaries, performance, and failure characteristics to the file systems. Informed Log-structured file system (I.LFS) uses E$\times$RAID for dynamic load balancing, user control of file replication, and delayed replication of files. RAIF already operates at the file system level and possesses all the meta information it needs to make intelligent storage optimization decisions [28]. Solaris's ZFS is both a driver and a file system [99]. Despite having all the necessary information, it supports storage policies on a storage-pool basis only. This means that whole devices and whole file systems use the same storage policies. RAIF provides more versatile virtualization on a per-file basis.

9.4.3 File-System–Level RAID

Higher level storage virtualization systems operate on files [33]. Their clients work as file systems that send networked requests to the servers [1, 38]. Clients find files on servers using dedicated meta-servers or hash functions [41]. Ceph and Zebra are distributed file systems that use per-file RAID levels and striping with parity [38, 105]. They have dedicated meta servers to locate the data. Ursa Minor's networking protocol supports special properties for storage fault handling [1]. Coda's client file system is a wrapper over a pseudo-device that directly communicates with a user-mode caching server [62]. The server replicates data on other similar servers.

File-system–level storage virtualization systems can support per-file storage policies. However, they still have fixed and inflexible architectures. In contrast, RAIF's stacking architecture allows it to utilize the functionality of existing file systems transparently and to support a variety of configurations without any modifications to file systems' source code. Thus, RAIF can use any state-of-the-art and even future file systems. Also, any changes to the built-in protocols of any fixed storage virtualization system will require significant time and effort and may break the compatibility between the storage nodes.

9.4.4 Storage Utilization

Storage resizing is handled differently by different storage-virtualization systems. For example, ZFS starts by lazily writing new data to newly added disks (which are initially free) and old disks [99]. Similarly, specially designed hash functions can indirectly cause more data to be written to the newly added disk [41]. This approach works only if all old data eventually gets overwritten, which is not the case in many long-term storage systems. An alternative approach is to migrate the data in the background [23, 34]. RAIF supports both lazy and background data migration.

9.4.5 Load Balancing

Media-distribution servers use data striping and replication to distribute the load among servers [4, 23]. The stripe unit size and the degree of striping have been shown to influence the performance of these servers [93]. Replication in RAIF uses proportional-share load balancing using the expected delay as the load metric. This approach is generally advocated for heterogeneous systems [91]. However, the number of performed I/O operations may be a better metric when the workload includes a mix of random and sequential operations [67].

Quality of service and fair resource sharing is another concern in shared storage systems [22, 43].

9.4.6 RAID Survivability

Remote mirroring is a popular storage disaster-recovery solution [61]. In addition to data replication, distributed RAID systems [97] use m/n redundancy schemes [85] to minimize the extra storage needs. RAIF can use a number of remote branches to store remote data copies or parity.

Two popular solutions to the silent data corruption problem [13] are journalling [29] (*e.g.*, using non-volatile memory) and log-structured writes [28, 99]. RAIF can support journalling with no major design changes.

Chapter 10

Conclusions

We designed a new file system profiling method that is versatile, portable, and efficient. The method allows the person profiling to consider and analyze the OS and the events being profiled at a high level of abstraction. In particular, the events can be anything that contributes to the OS execution latencies. The resulting profiles indicate pathologies and their dependencies. Access to the source code allows us to investigate these abstract characteristics such as lock or semaphore contentions. However, even without the source code, most of the problems can be described and studied in detail. In addition to its versatility, our method also allows profiling with high precision of about 40 CPU cycles and negligible overheads of only about 200 CPU cycles per profiled operation. The collected profiles are small and do not require the use of locks, which is especially important for SMP systems. When run with an I/O-intensive workload, we measured elapsed time overhead of less than 1%.

The developed method is intuitive and allows us to easily spot pathological patterns, anomalies, or simply differences in behavior—the job that the "computers" behind our eyes can perform especially well. However, to aid this analysis, we have implemented a set of special scripts that can select a smaller set of profiles or highlight the desired characteristics. This way we automate the tasks that computers on our desks are especially good for.

Our profiles can be captured entirely from the user-level. However, to collect even more information and to decrease the overheads even further we have captured profiles at the file system level. We have designed a flexible file system source code instrumentation system called FoSgen. FoSgen parses extensions written using the FiST language and applies them to Linux and FreeBSD file systems. FSprof is one such file system extension that can measure the latencies of all file system operations and collect them in the buckets. We have also created a stackable profiling file system for Windows file systems.

We used our method to collect and analyze profiles for task schedulers and several popular Windows, FreeBSD, and Linux file systems (Ext2, Ext3, Reiserfs, NTFS, NFS, CIFS, and Base0fs). We discovered, investigated, and explained multi-model latency distributions within several common file system operations. We also identified pathological performance problems related to lock contention, network protocol inconsistency, and I/O interference. We have shown how we used the developed latency profiling method to explain the performance of RAIF—a RAID-like file system that we developed.

10.1 Future Work

In this section we describe possible future research directions of our profiling method.

- Latency profiling produces small and informative profiles. This feature is especially suitable for profiling of distributed systems because capture and collection of profiles generates almost no inter-node traffic. Therefore, we plan to implement a distributed highly scalable profiling system.

- Captured profiles are small and can be analyzed in a short time. We plan to create a system for large scale automatic performance problems detection. In particular, we plan to profile file systems under random workloads and concurrently analyze these profiles to discover performance problems or implementation bugs.

- In this dissertation we focused on file system profiling. Nevertheless, this profiling method is applicable to most if not all OS components. We plan to apply our method to more OSs and a variety of their subsystems. We anticipate that higher resolution profiles will allow us to discover and explain infrequent OS events.

- The generated profiles contain multi-dimensional information. We have used simple two- and three-dimensional views for its visualization. More advanced scientific data visualization methods may make the process of visual analysis easier and more efficient.

- It is hard to profile and analyze systems behavior in virtual machines. OSprof can be used to capture and correlate events at all the levels: in VMs and on the host OS. We plan to use OSprof to find performance problems in VMware and Xen.

- In this dissertation we provided several examples of layered profiling. However, layered profiling can be performed at the granularity of every function. In that case, one would need a system for selective instrumentation of involved functions to minimize overheads.

- FoSgen supports Linux and FreeBSD operating systems. As we discussed before, it can be extended to support Windows. Also, it is clear that it can potentially support Solaris and other OSs whose VFS is similar to FreeBSD's or Linux's.

- We plan to extend FoSgen to support more possible instrumentation extensions. For example, FoSgen is an ideal solution to add tracing functionality directly to file systems. This solution would be similar to the Tracefs stackable file system [6] but would have smaller overheads. Another example is a journalling FiST extension. If applied by FoSgen, such an extension could add journalling functionality to existing non-journalling file systems.

- RAIF is a promising new approach to RAID systems development. The addition of journal support for write operations can significantly increase its reliability. Such a journal could be stored on another hard drive or a battery-backed RAM.

Bibliography

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-based Storage. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST '05)*, pages 59–72, San Francisco, CA, December 2005. USENIX Association.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 74–89, Bolton Landing, NY, October 2003. ACM SIGOPS.

[3] S. Aiken, D. Grunwald, A. R. Pleszkun, and J. Willeke. A Performance Analysis of the iSCSI Protocol. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03)*, pages 123–134. IEEE Computer Society, April 2003.

[4] S. Anastasiadis, K. Sevcik, and M. Stumm. Maximizing Throughput in Replicated Disk Striping of Variable Bit-Rate Streams. In *Proceedings of the Annual USENIX Technical Conference*, pages 191–204, Monterey, CA, June 2002.

[5] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP '97)*, pages 1–14, Saint Malo, France, October 1997. ACM.

[6] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004. USENIX Association.

[7] A. C. Arpaci-Dusseau. Implicit coscheduling: Coordinated scheduling with implicit information in distributed system. *ACM Transactions on Computer Systems (TOCS)*, 19(3):283–331, August 2001.

[8] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001. ACM.

[9] AT&T Bell Laboratories. *Plan 9 – Programmer's Manual*, March 1995.

[10] Bell Laboratories. *prof*, January 1979. Unix Programmer's Manual, Section 1.

[11] P. J. Braam. The Lustre Storage Architecture. `www.lustre.org/documentation.html`, October 2002.

[12] 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 *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224. ACM Press, June 1997.

[13] N. Brown. Re: raid5 write performance, November 2005. `http://www.mail-archive.com/linux-raid@vger.kernel.org/msg02886.html`.

[14] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pages 42–54, 2000.

[15] R. Bryant and J. Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 271–282, Atlanta, GA, October 2000. USENIX Association.

[16] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the Annual USENIX Technical Conference*, pages 29–44, Monterey, CA, June 2002. USENIX Association.

[17] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.

[18] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 15–28, 2004.

[19] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Seattle, WA, December 1994.

[20] Chakravarti, Laha, and Roy. *Handbook of Methods of Applied Statistics, Volume I.* John Wiley and Sons, 1967.

[21] 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.

[22] T. Chiueh, K. Gopalan, A. Neogi, C. Li, S. Sharma, S. Shan, J. Chen, W. Li, N. Joukov, J. Zhang, F. Hsu, F. Guo, and S. Doong. Sago: A Network Resource Management System for Real-Time Content Distribution. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'02)*, pages 557–562, National Central University, Taiwan, ROC, December 2002.

[23] C. Chou, L. Golubchik, and J. C. S. Lui. Striping doesn't scale: How to achieve scalability for continuous media servers with replication. In *International Conference on Distributed Computing Systems*, pages 64–71, Taipei, Taiwan, April 2000.

[24] W. Cohen. Gaining insight into the Linux kernel with Kprobes. *RedHat Magazine*, March 2005.

[25] Microsoft Corporation. Microsoft Windows XP Professional Resource Kit Documentation: Optimizing NTFS Performance. `www.microsoft.com/resources/documentation/Windows/XP/all/reskit/en%2Dus/`.

[26] T. Cortes and J. Labarta. Extending Heterogeneity to RAID level 5. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, June 2001. USENIX Association.

[27] E. Cota-Robles and J. Held. A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 159–172, New Orleans, LA, February 1999. ACM SIGOPS.

[28] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 177–190, Monterey, CA, June 2002. USENIX Association.

[29] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, CA, December 2005. USENIX Association.

[30] 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.

[31] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Annual USENIX Conference on Large Installation Systems Administration*, San Diego, CA, October 2003. USENIX Association.

[32] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using Latency to Evaluate Interactive System Performance. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI 1996)*, pages 185–199, Seattle, WA, October 1996.

[33] G. A. Gibson, D. F. Nagle, W. Courtright II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD Scalable Storage Systems. In *Proceedings of the 1999 USENIX Extreme Linux Workshop*, Monterey, CA, June 1999.

[34] T. Gibson. *Long-term Unix File System Activity and the Efficacy of Automatic File Migration*. PhD thesis, Department of Computer Science, University of Maryland Baltimore County, May 1998.

[35] K. Gopinath, N. Muppalaneni, N. Suresh Kumar, and P. Risbood. A 3-tier RAID storage system with RAID1, RAID5, and compressed RAID5 for Linux. In *Proceedings of the FREENIX Track at the 2000 USENIX Annual Technical Conference*, pages 21–34, San Diego, CA, June 2000. USENIX Association.

[36] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, June 1982.

[37] LBNL Network Research Group. The TCPDump/Libpcap site. `www.tcpdump.org`, February 2003.

[38] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 29–43, Asheville, NC, December 1993. ACM.

[39] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[40] D. Hendricks. A Filesystem For Software Development. In *Proceedings of the USENIX Summer Conference*, pages 333–340, Anaheim, CA, June 1990.

[41] R. J. Honicky and E. Miller. Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 2004.

[42] J. H. Howard. An Overview of the Andrew File System. In *Proceedings of the Winter USENIX Technical Conference*, February 1988.

[43] L. Huang, G. Peng, and T. Chiueh. Multi-dimensional Storage Virtualization. In *Proceedings of the 2004 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 14–24. ACM Press, June 2004.

[44] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[45] S. C. Johnson. Yacc – Yet Another Compiler-Compiler. Technical Report CS-TR-32, Bell Laboratories, Murray Hill, NJ, July 1975.

[46] M. Jones and J. Regehr. The Problems You're Having May not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 96–102, Rio Rico, AZ, March 1999.

[47] N. Joukov. [PATCH-2.6] i_sem contention in 2.6.10 generic_file_llseek, February 2005. `http://www.mail-archive.com/linux-fsdevel@vger.kernel.org/msg01628.html`.

[48] N. Joukov. Re: [RFC] Support for stackable file systems on top of nfs, November 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=113193082115222`.

[49] N. Joukov, R. Iyer, A. Traeger, C. P. Wright, and E. Zadok. Versatile, Portable, and Efficient OS Profiling via Latency Analysis. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005. ACM Press. Poster presentation, `http://doi.acm.org/10.1145/1095810.1118607`.

[50] N. Joukov, A. Kashyap, G. Sivathanu, and E. Zadok. Kefence: An electric fence for kernel buffers. In *Proceedings of the First ACM Workshop on Storage Security and Survivability (StorageSS 2005)*, pages 37–43, FairFax, VA, November 2005. ACM. (**Won best short paper award**).

[51] N. Joukov, H. Papaxenopoulos, and E. Zadok. Secure deletion myths, issues, and solutions. In *Proceedings of the Second ACM Workshop on Storage Security and Survivability (StorageSS 2006)*, pages 61–66, Alexandria, VA, October 2006. ACM.

[52] N. Joukov, A. Rai, and E. Zadok. Increasing distributed storage survivability with a stackable raid-like file system. In *Proceedings of the 2005 IEEE/ACM Workshop on Cluster Security, in conjunction with the Fifth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, pages 82–89, Cardiff, UK, May 2005. IEEE. (**Won best paper award**).

[53] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[54] N. Joukov, A. Traeger, C. P. Wright, and E. Zadok. Benchmarking File System Benchmarks. Technical Report FSL-05-04, Computer Science Department, Stony Brook University, December 2005. `www.fsl.cs.sunysb.edu/docs/fsbench/fsbench.pdf`.

[55] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST '05)*, pages 337–350, San Francisco, CA, December 2005. USENIX Association.

[56] N. Joukov, C. P. Wright, and E. Zadok. FSprof: An In-Kernel File System Operations Profiler. Technical Report FSL-04-06, Computer Science Department, Stony Brook University, November 2004. `www.fsl.cs.sunysb.edu/docs/aggregate_stats-tr/aggregate_stats.pdf`.

[57] N. Joukov and E. Zadok. Adding Secure Deletion to Your Favorite File System. In *Proceedings of the third international IEEE Security In Storage Workshop (SISW 2005)*, pages 63–70, San Francisco, CA, December 2005. IEEE Computer Society.

[58] A. S. Kale. Vmware: Virtual machines software. `www.vmware.com`, 2001.

[59] A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004. USENIX Association.

[60] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[61] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 59–72, San Francisco, CA, March/April 2004.

[62] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–225, Asilomar Conference Center, Pacific Grove, CA, October 1991. ACM Press.

[63] D. G. Korn and E. Krell. A New Dimension for the Unix File System. *Software-Practice and Experience*, 20(S1):19–34, June 1990.

[64] M. Kospach. *Statistics::Distributions - Perl module for calculating critical values and upper probabilities of common statistical distributions*, December 2003.

[65] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, March 1951.

[66] J. Levon and P. Elie. Oprofile: A system profiler for linux. `http://oprofile.sourceforge.net`, September 2004.

[67] Ixora Pty Ltd. Disk load balancing. `www.ixora.com.au/tips/tuning/disk_load.htm`.

[68] R. N. Mantegna and H. E. Stanley. *An Introduction to Econophysics: Correlations and Complexity in Finance*. Cambridge University Press, 2000.

[69] M. K. McKusick. Using gprof to tune the 4.2BSD kernel. `http://docs.freebsd.org/44doc/papers/kerntune.html`, May 1984.

[70] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[71] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 279–295, January 1996.

[72] Microsoft Corporation. File System Filter Manager: Filter Driver Development Guide. `www.microsoft.com/whdc/driver/filterdrv/default.mspx`, September 2004.

[73] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An On-Access Anti-Virus File System. In *Proceedings of the 13th USENIX Security Symposium (Security 2004)*, pages 73–88, San Diego, CA, August 2004. USENIX Association.

[74] A. Morton. sleepometer. `www.kernel.org/pub/linux/kernel/people/akpm/patches/2.5/2.5.74/2.5.74-mm1/broken-out/sleepometer.patch`, July 2003.

[75] 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.

[76] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer USENIX Technical Conference*, pages 247–256, Anaheim, CA, Summer 1990. USENIX.

[77] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD*, pages 109–116, June 1988.

[78] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137–152, Boston, MA, June 1994.

[79] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 25–33, New Orleans, LA, December 1995. USENIX Association.

[80] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[81] Z. N. J. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadat for a Time-Shifting File System. Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University, 2003. `http://hssl.cs.jhu.edu/papers/peterson-ext3cow03.pdf`.

[82] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. Analysis and Evolution of Journaling File Systems. In *Proceedings of the Annual USENIX Technical Conference*, Anaheim, CA, May 2005.

[83] W. H. Press, S. A. Teukolskey, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2002.

[84] H. Reiser. ReiserFS. `www.namesys.com/`, October 2004.

[85] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore Prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 1–14, San Francisco, CA, March 2003. USENIX Association.

[86] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Asilomar Conference Center, Pacific Grove, CA, October 1991. Association for Computing Machinery SIGOPS.

[87] D. S. H. Rosenthal. Evolving the Vnode interface. In *Proceedings of the Summer USENIX Technical Conference*, pages 107–118, Anaheim, CA, June 1990. USENIX Association.

[88] 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.

[89] Y. Rubner, C. Tomasi, and L. J. Guibas. A Metric for Distributions with Applications to Image Databases. In *Proceedings of the Sixth International Conference on Computer Vision*, pages 59–66, Bombay, India, January 1998.

[90] M. Russinovich. Inside Win2K NTFS, Part 1. `www.winnetmag.com/Articles/ArticleID/15719/pg/2/2.html`, November 2000.

[91] B. Schnor, S. Petri, R. Oleyniczak, and H. Langendorfer. Scheduling of parallel applications on heterogeneous workstation clusters. In *Proceedings of PDCS'96, the ISCA 9th International Conference on Parallel and Distributed Computing Systems*, pages 330–337, Dijon, France, September 1996.

[92] J. Seward, N. Nethercote, and J. Fitzhardinge. Valgrind. `http://valgrind.kde.org`, August 2004.

[93] P. Shenoy and H. M. Vin. Efficient striping techniques for variable bit rate continuous media file servers. Technical Report UM-CS-1998-053, University of Massachusetts at Amherst, 1998.

[94] Silicon Graphics, Inc. Kernprof (Kernel Profiling). `http://oss.sgi.com/projects/kernprof`, 2003.

[95] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[96] D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000*, chapter 12: File Systems, pages 683–778. Microsoft Press, Redmond, WA, 2000.

[97] M. Stonebreaker and G. A. Schloss. Distributed raid—a new multiple copy algorithm. In *Proceedings of the 6th International Conference on Data Engineering (ICDE'90)*, pages 430–437, February 1990.

[98] Sun Microsystems. *Analyzing Program Performance With Sun Workshop*, February 1999. `http://docs.sun.com/db/doc/805-4947`.

[99] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[100] M. J. Swain and D. H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[101] Sysinternals.com. Filemon. `www.sysinternals.com/ntw2k/source/filemon.shtml`, 2004.

[102] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net`, February 2005.

[103] VMware. Timekeeping in vmware virtual machines. `www.vmware.com/pdf/vmware_timekeeping.pdf`.

[104] L. Wall, H. Stenn, and R. Manfredi. dist-3.0. Technical report, Comprehensive Perl Archive Network (CPAN), 1997. `ftp.funet.fi/pub/languages/perl/CPAN/authors/id/RAM`.

[105] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 307–320, Seattle, WA, November 2006. ACM SIGOPS.

[106] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[107] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[108] C. P. Wright. *Extending ACID Semantics to the File System via ptrace*. PhD thesis, Computer Science Department, Stony Brook University, May 2006. Technical Report FSL-06-04.

[109] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.

[110] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Computer Science Department, Stony Brook University, October 2004. `www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf`.

[111] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A Platform for System Software Benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 175–187, Anaheim, CA, April 2005. USENIX Association.

[112] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[113] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

[114] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, Boston, MA, June 2001. USENIX Association.

[115] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.

Appendix A

FSprof FiST Extension

FiST extension below includes all the functionality necessary to collect and output collected latency distribution statistics. Because it also includes the `/proc` interface it is good for Linux only. As we described in Section 3, an OS-independent extension requires a separate module for the `/proc` interface.

```
%{
  /*
   * fsprof.fist: collect latency distributions
   *
   * Copyright (c) 2006 Nikolai Joukov and Erez Zadok
   * Copyright (c) 2006 Stony Brook University
   */

int  fsprof_init(void);
void fsprof_exit(void);
unsigned long long fsprof_pre(int op);
void fsprof_post(int op, unsigned long long init_cycle);

%}

debug off;
license "GPL";

%%

%op:all:precall {
  unsigned long long fsprof_init_cycle =
    fsprof_pre(fistOP_%op);
}

%op:all:postcall {
  fsprof_post(fistOP_%op, fsprof_init_cycle);
}
```

```
%op:init:precall {
  int fsprof_err = fsprof_init();
  if (fsprof_err)
    return fsprof_err;
}

%op:init:postcall {
  if (fistLastErr())
    fsprof_exit();
}

%op:exit:postcall {
  fsprof_exit();
}

%%
/*
 *  fsprof.fist: collect latency distributions
 *  Copyright (c) 2006 Nikolai Joukov and Erez Zadok
 *  Copyright (c) 2006 SUNY at Stony Brook
 */

#include <linux/config.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define FSPROF_MAX_DIGIT 32
#define FSPROF_MAX_TIME 0

/* this is our per-file system structure */
struct fsprof_statistics {
  unsigned long opcounts[fistOP_MAX];
  unsigned long long init_cycle;
  unsigned long long read_cycle;
  unsigned long long tot_cycles[fistOP_MAX];
  struct proc_dir_entry *fsprof_proc_root;
  unsigned long distribution[fistOP_MAX] \
    [FSPROF_MAX_TIME + 1][FSPROF_MAX_DIGIT + 1];
};

/* For now just use a static variable. */
static struct fsprof_statistics* this = NULL;
```

```
/* macros to read the TSC register */
#ifdef __ia64__
#define RDTSC(qp) \
  do { \
    unsigned long result; \
    __asm__ __volatile__( \
      "mov %0=ar.itc" : "=r" (result) :: "memory"); \
    qp = result; \
  } while (0);
#else
#define RDTSC(qp) \
  do { \
    unsigned long lowPart, highPart; \
    __asm__ __volatile__( \
      "rdtsc" : "=a" (lowPart), "=d" (highPart)); \
    qp = (((unsigned long long) highPart) << 32) | \
      lowPart; \
  } while (0);
#endif

/* function to reset statistics */
static void fsprof_stat_reset(void)
{
  memset(this, 0, sizeof(struct fsprof_statistics));
  RDTSC(this->init_cycle);
}

#define ADD_TO_BUFFER(str) \
  do { \
    char* tmp; \
    len = strlen((str)); \
    if (*ppos < total + len) { \
      if (count < done + len) { \
        len = count - done; \
      } \
      if (total < *ppos) { \
        len -= *ppos - total;\
        tmp = (str) + *ppos - total; \
      } else {\
        tmp = (str); \
      } \
      if (copy_to_user(buf, tmp, len)) { \
        done = -EFAULT; \
        goto out; \
      } \
      buf+= len; \
      done+= len; \
      if (count == done) \
        goto out; \
    } \
    total+= len; \
  } while(0);

/*
 * /proc interface read operation.
 * Dumps collected statistics in the plain text form
 * (e.g., ’cat /proc/fsprof’).
 */
static ssize_t fsprof_proc_read(struct file *file,
                                char *buf,
                                size_t count,
                                loff_t *ppos)
{
  char localbuf[1024];
  int len, done = 0, total = 0;

  unsigned int i, ii, iii, last_tick;

  if (!this->read_cycle)
    RDTSC(this->read_cycle);

  last_tick = (unsigned int)((this->read_cycle -
              this->init_cycle) >> 32);
  if (last_tick > FSPROF_MAX_TIME)
    last_tick = FSPROF_MAX_TIME;

  for (i = 0; i < fistOP_MAX; i++) {
    if (this->opcounts[i] > 0) {
      sprintf(localbuf, "OP_%s %lu %llu\n",
              fistOPnames[i],
              this->opcounts[i],
              this->tot_cycles[i]);
      ADD_TO_BUFFER(localbuf);
      for (ii = 0; ii < last_tick + 1; ii++) {
        len = 0;
        for (iii = 0; iii < FSPROF_MAX_DIGIT + 1; iii++) {
          if (this->distribution[i][ii][iii] != 0)
            len++;
        }
        if (len) {
          for (iii = 0; iii < FSPROF_MAX_DIGIT + 1; iii++) {
            sprintf(localbuf, " %lu",
              this->distribution[i][ii][iii]);
            ADD_TO_BUFFER(localbuf);
          }
          ADD_TO_BUFFER("\n");
        } else {
          ADD_TO_BUFFER("-\n");
        }
      }
    }
  }
  this->read_cycle = 0;
out:
  if (done > 0)
    *ppos += done;
  return done;
}

/*
 * /proc interface write operation.
 * Any write resets statistics
 * (e.g., ’echo 1 > /proc/fsprof’).
 */
static ssize_t fsprof_proc_write(struct file *file,
                                 const char *buf,
                                 size_t count,
                                 loff_t *ppos) {
  fsprof_stat_reset();
  return count;
}

struct file_operations fsprof_file_operations = {
  read:  fsprof_proc_read,
  write: fsprof_proc_write
};

#define PROC_NAME "fsprof"

/* Creates a /proc entry for user-mode access to statistics. */
static int fsprof_proc_create(void)
{
  struct proc_dir_entry *proc_de;
  int err = 0;

  proc_de = create_proc_entry(PROC_NAME, 0, NULL);
  if (!proc_de) {
    printk(KERN_ERR "Adding proc entry failed\n");
    goto out;
  }
  proc_de->owner = THIS_MODULE;
  proc_de->data = (void *)this;
  proc_de->proc_fops = &fsprof_file_operations;
out:
  return err;
}

/* Removes the proc entry. */
static void fsprof_proc_destroy(void)
{
  remove_proc_entry(PROC_NAME, NULL);
}

/*
 * Allocates memory and creates /proc entry for statistics.
 * No on-demand allocation to reduce run-time overheads.
 */
int fsprof_init(void)
{
  int err = 0;

  if (this)
    goto out;

  err = -ENOMEM;

  if (sizeof(struct fsprof_statistics) > PAGE_SIZE)
    this = vmalloc(sizeof(struct fsprof_statistics));
  else
    this = kmalloc(sizeof(struct fsprof_statistics),
                   GFP_KERNEL);

  if (!this)
    goto out;

  fsprof_stat_reset();

  err = fsprof_proc_create();

out:
  return err;
}

/* Remove /proc entry and free kernel memory at the end. */
void fsprof_exit(void)
{
  fsprof_proc_destroy();
  if (this) {
    if (sizeof(struct fsprof_statistics) > PAGE_SIZE)
      vfree(this);
    else
      kfree(this);
    this = NULL;
  }
}

/* Returns TSC register value and increments per-op counter. */
unsigned long long fsprof_pre(int op)
{
  unsigned long long ullic;
  this->opcounts[(op)]++;
  RDTSC(ullic);
  return ullic;
}

/*
 * Calculates latency, its log, and increments
 * the corresponding bucket.
 */
void fsprof_post(int op, unsigned long long ullic)
{
  unsigned long long l;
  unsigned long long ll_delay;
  unsigned int i, ii, iii, i_delay;

  RDTSC(l);
  ll_delay = (l - ullic);
  this->tot_cycles[(op)] += ll_delay;

  iii = 1;
  i_delay = (unsigned int)(ll_delay >> 5);
  for (i = 0; i < FSPROF_MAX_DIGIT; i++) {
    if (i_delay < iii)
      break;
    iii <<= 1;
  }

  /* time unit for sampled profiles */
  ii = (int)((l - this->init_cycle) >> 32);
  if (ii > FSPROF_MAX_TIME)
    ii = FSPROF_MAX_TIME;
  this->distribution[(op)][ii][i]++;
}
```