FSprof: An In-Kernel File System Operations Profiler

Nikolai Joukov, Charles P. Wright, and Erez Zadok Computer Science department Stony Brook University Stony Brook, NY 11794-4400 Technical Report FSL-04-06

kolya,cwright,ezk@cs.sunysb.edu

ABSTRACT

Developing efficient file systems is difficult. Often, profiling tools are useful for analyzing system bottlenecks and correcting them. Whereas there are several techniques to profile system call activity or disk-block activity, there are no good tools to profile file systems—which logically reside below system calls and above disk drivers. We developed a tool called FSprof that instruments existing file systems' source code to profile their activity. This instrumentation incurs negligible runtime overhead. For file systems that do not have source code available, we also developed a thin file-system wrapper. When a profiled file system runs, it records operation frequencies and precise latencies and sorts them into configurable exponential buckets. We wrote additional tools to help verify, analyze, and display the profiling data.

We ran FSprof on several popular Linux file systems: Ext2, Ext3, Reiserfs, and a stackable (layered) file system called Wrapfs. Our analysis revealed interesting discoveries about file systems and benchmarks. We analyzed bi-modal and even tri-modal distributions we found in certain operation latencies, which result from complex interactions between file system caches and disks. We illustrate how simple file system designs can lead to serious lock contention and slow down the entire operating system. We show how seemingly similar file system benchmarks can unexpectedly behave rather differently. We also observed that a tiny percentage of certain calls can have a disproportionate overall effect.

FSprof is the first tool specifically designed for analyzing file system behavior, using high precision and a fine level of detail. FSprof helps developers collect and organize information, then diagnose and optimize file system performance.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques and Performance Attributes; D.4.3 [**Operating Systems**]: File Systems Management; D.4.8 [**Operating Systems**]: Performance—*Measurements, Monitors, and Operational Analysis*

General Terms

Design, Measurement, Performance

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

Keywords

File Systems, Profiling, Instrumentation

1. INTRODUCTION

File systems control access to data; therefore, their performance is critical to many applications. Developing file systems is a difficult task. Profiling tools are generally helpful in debugging and optimizing complex software systems. Alas, there are no profiling tools available specifically for file systems. Today, file system developers must use one of several unsatisfactory techniques to profile file systems:

- System call monitoring can show which file-system-related system calls applications invoke. The system call API is similar to file system APIs, but it is unsuitable for file system monitoring for two reasons. First, system calls do not map perfectly to file system events. For example, every system call that accesses a file name (e.g., open or mkdir) results in a file system lookup operation that translates pathnames to OS objects, typically followed by a permission-checking operation. In other words, file system designers develop to a different API than the system call API. Second, memorymapped operations do not show up as system calls, but result in file system read_page or write_page operations, invoked from a page fault handler. This means that system call tracing for the purpose of file system profiling ignores important and commonly-used memory-mapped events.
- Disk-device or block-level profiling shows raw block read and write requests going to a disk device [30]. However, at this low level, vital file system information is lost. At the device level, all we see are block reads and writes, and their offsets within the disk. There is no knowledge about whether the block is data or meta-data; what the block's offset within a file is; what other blocks the block relates to; and more.
- For network file systems such as NFS, protocol-level or packetlevel profiling is often done [9]. Such profiling is limited by the protocol and often does not accurately reflect the file system's operations. One reason is that network file system clients cache information and change sequences of operations; because of this they appear different inside the network file system client's OS vs. the network. Similarly, network file system servers perform caching and processing before executing file system operations, so the network profile is representative of neither the client's nor the server's filesystem activity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

• Whole-kernel profilers provide information about the CPU execution time of every OS function, including file system functions. These profilers use a combination of sampling and hooks at function entry/exit points to produce an approximation of how much CPU and I/O each function used, and a corresponding call graph. Unfortunately, these profilers do not have file-system–specific information so they must record information on every kernel function, leading to higher overheads. Finally, the profiles do not provide the distribution of time used by a function, but rather the average time used.

In other words, existing profiling techniques are unsuitable for precise file system profiling. This is not surprising, since none of these techniques operate at the exact level of the file system: system calls are above, disk-block and networks are below file systems, and whole-kernel profiling is too general and therefore inefficient.

File systems are complex pieces of software that may interact with several other subsystems: the VM system and page caches, disk device drivers, and networks. As such, file systems include a mix of CPU-intensive and I/O-intensive operations with widely varying latencies and frequencies. These interactions are non-trivial and make it particularly difficult to analyze file system behavior.

The lack of proper file system profiling tools has forced file system developers to resort to ad-hoc or indirect techniques to analyze file system behavior. Our experience over the last decade is no different. For example, we often wrote small programs to microbenchmark a specific file system operation, typically by calling system calls; as mentioned above, system calls embody multiple file system operations and thus cannot separate the behavior of each file system operation. At other times we used a large compile to benchmark a file system, Postmark [14] to stress meta-data operations, or fsx [18] to stress the file system's interaction with the VM.

In this paper we introduce *FSprof*, the first profiling tool that was specifically designed for analyzing file system behavior. FSprof can provide accurate counts of each file system operation at the file system level, and very precise timing for file system operations. FSprof operates in two modes. First, if source code is available, FSprof instruments the file system source with built-in profiling and timing. This instrumentation was carefully designed to ensure that the overhead introduced was negligible: less than 100 CPU cycles per file system operation and below 1% of elapsed time for I/O-intensive workloads. Second, if source code is not available, we have developed a thin file-system wrapper with FSprof built in, which can be mounted on top of any other file system, to intercept and measure the mounted-on file system's operations.

Using FSprof we analyzed several common file system benchmarks: a few compile benchmarks, Postmark, and a recursive grep microbenchmark. Our analysis revealed several unexpected results. First, we scientifically verified that file systems contain complex interactions; this was seen in bi-modal and tri-modal operation latency distributions we found, investigated, and explained. Second, we found that for some common combinations of file systems and benchmarks, a small fraction of operations accounted for the vast majority of time the file system spent executing a certain type of operation; this implied that small changes in benchmarks can result in non-negligible performance differences. We show that file systems' performance is generally sensitive to a disproportionally small number of operations. Third, we found that seemingly similar (large-compile) benchmarks can produce rather different results due to differences in file system operation mixes. Finally, we show that file systems must be designed very carefully, or face significant performance bottlenecks (i.e., lock contention), or worse-hurt the performance of the entire operating system.

FSprof is a unique tool that for the first time offers kernel de-

velopers a direct window into the inner workings of file systems. We believe that FSprof is a promising tool and technique, because we were able to discover and explain several counter-intuitive filesystem behaviors in just a few months with FSprof. With very low overheads and precise measurements, we hope that FSprof will become a common tool used by file system developers in the future.

The rest of this paper is organized as follows. Section 2 describes the guiding principles of our design and Section 3 describes the implementation of our system. We evaluate our system in Section 4. In Section 5 we present several usage scenarios and analyze profiles of several real-world file systems and workloads. We describe related work in Section 6. We conclude and discuss future directions in Section 7.

2. DESIGN

When designing FSprof, we made three main decisions: latency aggregation, using exponential buckets to collect results, and only profiling VFS operations.

Latency aggregation We capture the latency of file system operations. It can serve as an ideal metric for file system profiling because the operation latency contains information about the operation's CPU execution time as well as the associated I/O request delays.

Capturing the function execution latency is simple, and requires minimal code instrumentation. Early code-profiling tools rejected latency as a viable performance metric, because in multitasking environments the program can be rescheduled at any arbitrary point in time, perturbing the results. Fortunately, execution in the Unix kernel is different from execution in user space. The kernel can be preempted only at certain points, when the file system operation specifically waits on I/O requests, waits to get a lock, or yields the CPU. Most functions have only a few such points, and their presence simply adds to the possible delay (as observed by both the user and FSprof). Some kernels can be compiled with kernel code execution preemption enabled. However, even in that case, profiles generated by single processes are not affected by the preemption as we show in Section 5.1.2. Analyzing multi-process workloads on a preemptible kernel is beyond the scope of this paper.

Exponential buckets The captured latencies are sorted and stored in exponential buckets. Different code execution paths form different peaks on the histogram; therefore, there is no need to preserve information about every individual execution of an operation.

Collecting statistical information in buckets is a common program profiling technique, which allows measuring the most interesting information without burdensome storage and processing requirements. Usually, information is stored in linear histograms where information is distributed in buckets that represent several CPU instructions or code lines. In contrast, we store the number of operations with a given latency in the bucket corresponding to that latency. Therefore, our histogram is not directly tied to the source code. Nevertheless, the distribution of latencies forms well-defined peaks that can be easily correlated with the file system code. We used exponentially distributed buckets because the difference between the latencies of the different execution paths can be several orders of magnitude. For example, to return cached information might take tens of instructions (i.e., only a few nanoseconds), but to access the disk or network takes tens of milliseconds.

VFS operations profiling We capture latencies of only Virtual File System (VFS) operations. This allows us to add minimal memory and time overheads, while capturing all file-system–related activities. The VFS interface defines a limited set of operations on



Figure 1: Profile of the Ext2 lookup operation under the kernel build workload. (Number of operations in every bucket.)



Figure 2: Profile of the Ext2 lookup operation under the kernel build workload. (Total delay of each bucket.)

file system primitives, which are the only interface between the upper OS layers and a file system. Therefore, it is sufficient to profile VFS operations to collect information about all of a file system's activity. There are 74 VFS operations on five objects, but most file systems only define a subset of them (e.g., Ext2 defines functions for only 28 VFS operations).

Not all measured latencies directly contribute to the total system and elapsed time (including I/O) as seen by user space, because some VFS operations call other VFS operations. For example, after a directory entry is found, lookup calls read_inode to locate and read the on-disk inode structure into memory. Therefore, one needs to understand the file system call graph structure to correlate the total system and I/O times with the captured profile. Fortunately, in many cases the call graph structure can be inferred from the profile. Operations that call other VFS operations often have latency distribution peaks that are a superposition of the peaks of the called operations. More details about the VFS operation call graphs and modes of operation can be obtained using incremental profiling techniques described in Section 5.1.1.

To capture the dependence of file system operations on a certain part of a run, we save the latency distribution at pre-defined intervals. Therefore, the profile is a 4-dimensional view of file system operations consisting of:

- VFS operation
- Latency
- Number of operations with this latency
- Elapsed time interval

Figure 1 shows an example 3D view of the lookup operation on Ext2. The z axis contains the number of operations that fall within

struct	file_operations	ext2_dir_operations	=	{
	read:	generic_read_dir,		
	readdir:	ext2_readdir,		
	ioctl:	ext2_ioctl,		
	fsync:	ext2_sync_file,		
};				

Figure 3: Ext2 file system directory operations. The kernel exports the generic_read_dir function for use by multiple file systems.

a given bucket (the x axis) within a given elapsed time interval (the y axis). Figure 2 shows the estimated delay for each bucket on the z axis, which is the number of operations in the x^{th} bucket multiplied by $1.5 \cdot 2^x$. A small number of invocations in buckets 22–25 are responsible for a large portion of the operation's overall delay.

Our profiles contain more information than those generated by any other file-system—profiling technique available today, and provide information about both system and I/O times associated with particular VFS operations. The profile can be used to draw general conclusions from particular operation patterns, because information about all the delays of a given operation is available. Often benchmarks focus on a specific set of file-system operations, so the overhead related to less-frequently—used operations is not apparent. Our profiles can be used to draw conclusions about and gain insight into how an operation mix would perform on a given file system, even if such a workload was never executed on it. Currently, many file system developers optimize performance for some particular workload. In the future, the insight gained by using our profiles could help developers optimize to any workload.

3. IMPLEMENTATION

We describe four aspects of our implementation: source-code instrumentation in Section 3.1; latency aggregation in Section 3.2; results representation in Section 3.3; and an alternative method of gathering profile results with stackable file systems in Section 3.4.

3.1 Source Code Instrumentation

We have chosen source code instrumentation to insert latency measurement code into existing file systems, because it is simple and has small profiling overheads. Our code instrumentation process consists of four steps:

- 1. Copying source files with profiling code to the file system's directory and adding them to the Makefile.
- 2. Scanning all source files for VFS operation vectors.
- 3. Scanning all source files for operations found in the previous step, and inserting latency calculation macros in the function body.
- 4. Including the header file which declares the latency calculation macros into every C file that needs it.

These four actions are performed by a shell script using sed.

The source code instrumentation script 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 a hard-coded associated function. For example, Figure 3 shows the definition of Ext2's open-file operations for directories. The instrumentation script scans every file from the file system source directory for operations 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. A profiling startup macro called FSPROF_PRE(op) is inserted at the beginning of every function found to begin measuring the latency. Every return statement and the end of void functions is preceded with an FSPROF_POST(op) macro, which ends the latency measurement. In both cases, *op* is replaced with the name of the current VFS operation. For non-void functions of type *f_ype*, the return statements are transformed from return foo(x) to:

```
{
    f_type tmp_return_variable = foo(x);
    FSPROF_POST(op);
    return tmp_return_variable;
}
```

This transformation captures the latency of the expression foo(x).

Often, file systems use generic functions exported by the kernel. For example, the Ext2 file system uses the generic_read_dir kernel function for its read operation as shown in Figure 3. To capture the latencies of such external functions, we created a separate C file with wrappers for these functions that are instrumented using the standard macros. These wrapper functions have the names of the wrapped functions preceded by the FSprof_prefix. We use actual wrapper functions as opposed to inline functions or macros, so that our function has an address for the operations vector to use. The same prefix is inserted for all external functions in the operation vectors. Our wrapper functions are similarly instrumented with the macros and are called instead of the original external functions.

The instrumentation program is a shell script with sed fragments, and is only 220 lines long. The script does not parse C program code; instead, it looks for particular patterns within filesystem code. Despite its simplicity, the script successfully instrumented all the file systems we tried it on (Ext2 and Ext3 [8], Reiserfs 3.6 and 4.0 [22], NFS [20], NTFS [24], and FiST-generated stackable file systems [33]) under Linux kernel versions 2.4.24 and 2.6.8. We have paid special attention to detecting possible parsing errors. In particular, the script provides informative error messages if a deviation from expected patterns is detected.

3.2 Latency Aggregation

The FSPROF_PRE(op) macro declares a variable to store the time and a call to the FSprof_pre function that actually writes the time into that variable. Before returning the invocation time, the FSprof_pre function also updates a per-operation counter. The FSPROF_POST(op) macro calls the FSprof_post function, which takes the operation name and the function invocation time stored by FSprof_pre as arguments. It queries the current time and calculates the operation latency. Then it adds the latency to the total operation's latency value and adds one to bucket k such that $2^{k-1} \leq latency < 2^k$. (Bucket sizes can be configured, but we primarily used \log_2 buckets.) By having separate counters in the FSprof_pre and FSprof_post functions, post-processing scripts can verify that there were no errors in code instrumentation, storing values, and sending the profile to user-space.

To profile fast VFS operations, we needed to measure the time with a resolution of tens of nanoseconds. The only simple and fast way to do it is to use the CPU cycle counter (TSC on x86), which is built into most modern microprocessors. This method is fast because querying the counter consists of just a single CPU instruction. On Linux, other less precise and slower time-keeping functions (e.g., *gettimeofday*) eventually use the same counter. On multiprocessor systems, Linux synchronizes the cycle counters on all the CPUs with a precision of one microsecond. The scheduler is trying to schedule the task on the same CPU as it ran on before. If, however, a task is rescheduled to another processor, then the measured latency has only a microsecond precision. Fortunately, the probability that a fast VFS operation is rescheduled is small because the quantum of time a process is allowed to run (usually 1/10 of a second) is much longer than the latency of the fast operations. On the other hand, a slow operation's latency is not seriously affected by an errant microsecond, because the buckets are distributed exponentially. Therefore, we believe that our profile results are credible even on multiprocessor systems. To be consistent, we store and analyze all time values in terms of CPU cycles.

The profile can be accessed through an entry in the /proc file system. Profiles can be read from /proc in plaintext format. Plaintext is more convenient than binary data, because it is directly humanreadable 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. Writing to the per-file–system /proc entry resets the profile's counters.

3.3 Results Representation

We wrote a script that generates 2-dimensional and 3-dimensional views from the 4-dimensional profile data. We found the following three data views for a particular VFS-operation especially useful:

- The number of invocations with a given latency within each elapsed-time interval.
- The total number of invocations with a given latency.
- The total operation latency in each elapsed-time interval.

The last two views are obtained from the original 4D profiles by summing up the values in one of the dimensions.

In addition, the data-processing script checks 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. These numbers are added to the profile during different profiling phases, so this verification catches potential code instrumentation errors.

We created a set of scripts that generate formatted text views and Gnuplot [11] scripts to generate 3D and 2D views of the data. All the figures representing profiles in this paper were automatically generated.

3.4 Stackable Profiling

Stackable file systems are a layer between the VFS code and lower-file systems (e.g., Ext2) [33]. They appear as normal file systems to the VFS code that invokes VFS operations and as the VFS to the file system they are layered on. Essentially, they pass through the calls from the VFS to the file system they are layered on as shown in Figure 4. This makes them ideal for profiling file systems whose source code is not available. We instrumented a simple pass-through stackable file system, called *Wrapfs*, that is part of the open-source FiST toolkit [33]. We analyze the resulting overheads for the instrumented stackable Wrapfs in Section 5.1.3.

4. EVALUATION

Using Ext2 as a baseline, we evaluated the overhead of FSprof with respect to memory usage, CPU cache usage, latency added to each profiled operation, and execution time.

We conducted all our experiments on a 1.7GHz Pentium 4 machine with 256KB of cache and 1GB of RAM. It has an IDE system disk, and the benchmarks ran on a dedicated Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disk with an Adaptec 29160 SCSI controller. We unmounted and remounted all tested file systems before every benchmark run. We also ran a program we wrote called



Figure 4: Stackable File System.

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 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.

4.1 Memory Usage and Caches

We evaluated the memory and CPU cache overheads of FSprof. 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. 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. The third memory overhead comes from storing profiling results in memory. If only cumulative totals are kept, then 1KB of memory is used. For the lengthy kernel compile benchmark with a 2 second resolution, 4MB of memory is used. Since we merely append to the memory buffer, only a small 1KB active portion of this 4MB is used at any given time; therefore only 1KB of CPU data cache is affected.

4.2 CPU Time Overhead

To measure the CPU-time overheads, we ran Postmark [14] on an unmodified and also on an instrumented Ext2 file system. Postmark simulates the operation of electronic mail servers. It performs a series of file system operations such as create, delete, append, and read. We configured Postmark to use the default parameters, but we increased the defaults to 20,000 files and 200,000 transactions. We selected this configuration because it 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 file system 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 5, there are three additional components added by FSprof: making function calls (solid lines), reading the TSC register (dotted lines), and storing the results in the correct buckets (dashed lines). To understand the details of this per-operation overhead, we created two additional file systems. The first contains only empty profiling function bodies, so the only overhead is calling the profiling functions. The system time increase over Ext2 was 0.28 seconds (1.5%). The second file system reads the TSC register, but did not include code to sort the information and store it into buckets. The system time increased by 0.36 seconds over Ext2

(2.0%). Therefore, a 1.5% system time overhead is due to calling the profiling functions, 0.5% is due to reading the TSC, and 2.0% is due to sorting and storing the profiling information.



Figure 5: Profiled function components.

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 overhead to be 92 cycles per operation. This result is confirmed by the fact that the smallest values we observed in any profile are always in the range of 64–128 cycles. The 92 cycle overhead is well below most operation latencies, and can influence only the fastest of VFS operations that perform very little work (e.g., if sync_page is called to write a dirty page to disk, but it returns immediately if the page is not dirty).

We discuss the performance impact of the stackable profiler in Section 5.1.3.

5. USAGE EXAMPLES AND ANALYSIS

There are two general ways in which FSprof can be used. In Section 5.1 we describe how profiles can be used to study and compare the behavior of file systems and hardware platforms. In Section 5.2 we describe how FSprof can be used to analyze and compare different workloads on a fixed hardware and software configuration.

The workload we call *grep* -r in the paper is generated by running *grep* -r for a non-existent string on this Linux 2.4.20 source tree. Standard confidence interval calculation techniques are not generally applicable for analysis of the generated profiles because averaging can result in the information loss. For example, the bdflush thread is started every 5 seconds. Averaging the profiled buckets' values results in losing information, because slight variations in the runs are diluted. Therefore, we need to analyze a single characteristic profile. We ran all our experiments at least 5 times and used the run with the median elapsed time for analysis. We unmounted and remounted the profiled file systems and purged the caches using *chill* between all the runs.

5.1 System Profiling

In this section we discuss three specific analyses. In Section 5.1.1 we demonstrate incremental file system profiling under Ext2; in Section 5.1.2 we show the impact of lock contention on several generations of journaling file systems; and in Section 5.1.3 we investigate the influence of stackable file systems on performance.



Figure 6: Distribution of VFS operations latencies for Ext2 for the *grep -r* workload.

5.1.1 Incremental File System Profiling

In this section we show how FSprof can be used to incrementally analyze file system behavior. The default profiling method provides information about the latency distribution of the file system's VFS operations under a given workload. For example, Figure 6 and Table 1 show the latencies for Ext2 for a run of *grep -r*. This overview 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 generalpurpose Linux generic_file_read function, which then calls the readpage operation. Therefore, we can infer from Table 1 that the lookup, read, and readdir operations are responsible for more than 99% of file system latency under the given workload, or 11.0 seconds. This conclusion correlates well with the difference between the overall elapsed time and user time, which is 11.2 seconds. The 0.2 second difference is attributed to the system call overhead and context switches.

To investigate the origin of the peaks in the distribution, one needs more statistical information than the distribution of latencies. For this purpose, we designed a set of macros that can be used in place of the basic macros that we insert at the beginning and end of the VFS functions. These extended macros collect the distribution of other parameters in the same exponential buckets and their corresponding latencies. Similar to general Exploratory Data Analysis (EDA) [31] techniques, the decision about which parameters to capture depends on more detailed information about the VFS operations and as such cannot be fully automated. We assume that the person performing incremental analysis of a file system is capable of reading source code and replacing the default macros with the extended macros, because this person is most likely a developer or a maintainer of the file system. The extended macros take the value being profiled and peak locations as arguments. For each peak, a

Operation	Count	Delay	Delay	
		(CPU Cycles)	(ms)	
follow_link	110	87,924	0.05	
lookup	13,640	3,069,646,302	1,805.67	
open	12,915	1,986,912	1.17	
read	27,408	7,320,153,190	4,305.97	
readdir	1,687	7,736,036,614	4,550.61	
read_inode	13,641	2,943,218,320	1,731.30	
readpage	43,991	477,748,104	281.03	
release	12,915	1,291,492	0.76	
sync_page	20,141	108,733,448	63.96	
write_inode	12,107	10,567,072	6.22	
write_super	1	2,552	0.00	

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



Figure 7: Distribution of the block caching information (top) and the difference between the currently requested block number and a previously-read block number (bottom) for the three peaks of read_inode.

separate profile is constructed.

For example, we investigated the origin of the three peaks in Figure 6 (buckets 8-11, 17-18, 20-24) in the read_inode operation's latency distribution. First, we profiled the function itself by moving the original macros closer together and locating the subroutines responsible for the latency. The top-level VFS operation functions themselves are usually simple and do not cause any considerable delay. In the case of the Ext2 read_inode operation, the delay is defined by the call to sb_bread. In particular, the macros placed right before and after this call collected the same information as the macros placed at the beginning and the end of the whole read_inode function. Therefore, our next step was to understand why sb_bread had three peaks. This function takes two arguments: a pointer to the super block structure and the number of the physical block on the disk corresponding to the inode. From the operation latency distributions we know that the leftmost (fastest) peak is created by very fast operations that complete in less than two microseconds. We hypothesized that no disk I/O operations were involved and the information is likely in the cache. To verify our hypothesis, we substituted the default latency profiling macro with the one that profiles boolean values. In particular, we profiled the buffer head structure's BH_Uptodate state flag before the sb_bread call. The top of Figure 7 shows the profiling results based on BH_Uptodate*1024. We did not simply use BH_Uptodate because the profiling tool combines all of the buckets below 2^5 , which is less than its overhead in CPU cycles. All of the operations in the first peak of the original profile now fall into the bucket representing 1024-2047 cycles, which indicates that they are in the cache. All of the operations in the slower two peaks fall into the lowest bucket, indicating they are not in the cache.

Next, we investigated the two slower read_inode peaks. We profiled the difference between the block numbers of the current and the preceding sb_bread requests. The results shown on the



Figure 8: Profile of Reiserfs 3.6 (default configuration) under the *grep -r* workload.



Figure 9: Profile of Reiserfs 3.6 (with *notail*) under the *grep -r* workload.

bottom of Figure 7 indicate that there is a strong correlation between the peaks and the profiled value. In particular, 88% of the operations in the middle peak are requests to read inodes located close to a previously read inode. On the other hand, 94% of the read requests in the slowest peak are requests to read inodes located 16K–8M disk blocks away from the last read inode. We know that blocks with close block numbers are usually located physically close to each other. Therefore, we can conclude that the middle peak is defined by the reads that do not require long disk head seeks and the right peak is defined by the operations that do require them. These results correspond to our knowledge of Ext2, which uses an FFS-like allocation scheme [8]. Inodes in the same directory are stored in the same cylinder group (so they have close block numbers), but inodes in different directories are spread throughout the disk.

5.1.2 Lock Contention and Journaling



Figure 10: Latency distribution compared to elapsed time for write_super (left) and read (right) on Reiserfs 3.6 (default configuration).



Figure 11: Latency distribution compared to elapsed time for write_super (left) and read (right) on Reiserfs 3.6 (with *no-tail*).

In this section we demonstrate how FSprof can be used to discover lock contention problems in file systems. We profiled Reiserfs 3.6 under the grep -r workload. Reiserfs 3.6 is now the default file system in several distributions of Linux. The profile with the default Reiserfs configuration is shown in Figure 8. By default, Reiserfs enables tail merging which combines small files and the ends of files into a single disk block. As seen in Figure 9, we also profiled Reiserfs with the notail option, which disables tail merging. The first interesting observation we can make is that the write_super operation takes longer than most other operations: 64M-1G CPU cycles (0.038-0.58 seconds) for the default configuration, and 16M-128M CPU cycles (0.009-0.075 seconds) with notail. The second observation is that there is a clear correlation between the longest dirty_inode, read, and write_super operations. When the latency distribution is viewed along with elapsed time, the correlation becomes especially obvious as shown in Figures 10 and 11. 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. 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.

Figure 12 shows the distribution of latencies of Ext3. The Ext3 write_super operation takes less than 4K CPU cycles, because it updates the journal asynchronously. It calls the Linux journaling interface's log_start_commit function and releases the BKL shortly after that. In contrast, the Reiserfs 3.6 write_super operation calls flush_old_commits, which returns only after all updates are written to disk. Since this operation is performed while the file system's superblock and the BKL are held, the file system and the kernel are blocked for the duration of the whole operation. The bdflush kernel thread is started 5 seconds after the previous run completes. This suggests that every 5 seconds, several mounted Reiserfs partitions may *each* sequentially block the kernel for up to 0.58 seconds—even if the CPU is free to service other processes.



Figure 12: Profile of the Ext3 file system under the *grep -r* work-load.



Figure 13: Profile of the Reiserfs 4 file system under the *grep -r* workload.

At the time of this writing, Reiserfs 4.0 became available in version 2.6.8.1-mm2 of the Linux kernel. Using FSprof, we instrumented Reiserfs 4.0 under 2.6.8.1-mm2 with kernel preemption enabled. We profiled Reiserfs 4.0 under the *grep -r* workload. Results presented in Figure 13 demonstrate that the file system 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 operationbecause it is called with the BKL held. This tends to reduce lock contention considerably and improve Reiserfs 4.0's performance overall. However, inode access time updates are still the longest individual operations in Reiserfs 4.0. In Reiserfs 4.0, the lookup operation has only a single peak that is slower than the fastest peak in Ext3, but faster than the middle peak (the fastest disk I/O) in Ext3. The lookup operation has also improved significantly from Reiserfs 3.6 to 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 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 -1); that is why NFSv3 implements a special protocol message called READDIRPLUS which combines directory reading with stat information [7]. 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.

To verify that the changes in the Reiserfs profile are not caused by the Linux kernel changes between 2.4 and 2.6, as well as to estimate the impact of kernel preemption, we profiled Ext3 on 2.6.8.1mm2. The Ext3 code has not changed significantly between 2.4 and 2.6. The resulting profile is very similar to the one shown in Figure 12. Therefore, we conclude that the improvement in Reiserfs 4.0 is indeed thanks to the code changes. We have not observed any artifacts that could be caused by preemptive kernel scheduling. This is likely because we ran only one process (grep) and there was no other process to preempt it.

It was previously known that Reiserfs spends a lot of time waiting on locks [5], but FSprof provided us information about the particular modes of operation and the corresponding conditions that cause problems. This information can be directly used to optimize the code and determine the most harmful operation scenarios.

5.1.3 Influence of Stackable File Systems

We used FSprof to evaluate the impact of file system stacking on the captured profile. Figure 14 shows the latency distribution of Wrapfs, 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 allocation of VFS objects such as inodes, dentries (directory entries), or memory pages. As we can see in Figure 14, Wrapfs'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 [33]. Therefore, stackable file systems create copies of each lower-level object they encounter.

This behavior of the 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



Figure 14: Distribution of operation latencies for unmodified Wrapfs mounted over Ext2, and for the Ext2 file system, under a *grep -r* workload.

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 Wrapfs 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 double representation of inodes implies double caching of data pages.

We found a relatively simple solution to the problem. 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:

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 com-



Figure 15: Distribution of operation latencies for Wrapfs without double caching mounted over Ext2, and for the Ext2 file system, under a *grep -r* workload.

pile and Postmark workloads. No races or other problems were observed.

As can be seen in Figure 15, the no-double-caching patch described above decreases the system time compared to the original Wrapfs, 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.

5.2 Workload Characterization

Compile benchmarks are often used to evaluate file system behavior. We show that even seemingly similar mixes of source files generate considerably different VFS operation mixes. Therefore, results obtained during different compile benchmarks can not 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 2 shows the general characteristics of the packages. The build process of these packages consists of a preparation and a com-

	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 2: Compile benchmarks' characteristics.

pilation 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 cachepurging 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 16 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 16(a) and 16(b) show that even though the SSH and Amutils 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 16(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 1 and 2 on page 3 show the profile of the lookup operation during the kernel build process, where we see that the preparation phase causes the vast majority of lookups that incur disk I/O.

Table 3 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 [26]). SSH's average lookup delay is only slightly higher because the higher percentage of misses is compensated by the high fraction of disk operations 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-head seeks. Therefore, its average lookup delay is the highest.



(c) Kernel 2.4.20

Figure 16: Various compile profiles on Ext2. Note that each plot uses a different scale for both the operation count and delay. The count uses the left scale, and the delay uses the right scale.

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

Table 3: Distribution of the Ext2 lookup operations among the three peaks representing a page cache hit, short disk seek, and long disk seek. The cache miss ratio is calculated as the sum of the operations in the middle and slowest peaks over the total number of operations.

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 CPU and I/O times.

6. RELATED WORK

The simplest and oldest way to profile code is to measure the total program execution time and its user level and in-kernel executed components in different conditions [29]. In the late 1970s, new code profiling techniques emerged. UNIX prof [3] instruments source code at function entry and exit points. An instrumented program'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 sampling is a relatively inexpensive way to capture how long a program fragment is executed in multi-tasking environments where a task can be rescheduled at any time during any function execution interval. Gprof [12] 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 early 1980s [16]. However, instrumented kernels had a 20% increase in code size and an execution time overhead up to 25%. Kernprof [26] uses a combination of PC sampling and kernel hooks to build profiles and call graphs. Kernprof interfaces with the Linux scheduler to count time that a kernel function spent sleeping (e.g., to perform I/O) in the profile.

More detailed profiles with granularity as small as a single code line can be collected using the *tcov* [27] profiler. Most modern CPUs contain special hardware counters for use by profilers. The hardware counters allow correlation between profiled code execution, CPU cache states, branch prediction functionality, and ordinary CPU clock counts [4]. Essentially, all modern CPU execution profilers rely on some combination of program counter sampling, scheduler instrumentation, functions entry/exit point instrumentation, and hardware counters. Finally, there are special tools to monitor particular problems such as lock contention [6, 19] or memory leaks and cache purging [25].

In general, fewer and less developed tools are available to profile disk 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, as much as possible, file systems designs avoid seeks [17, 23]. Unfortunately, modern hard drives expose very little information about their internal data placement to the OS. The OS can only assume that blocks with close logical block numbers are also 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. Some disk vendors make statistics, such as the number of seek operations, available through the SMART [1] interface. The Linux kernel optionally maintain statistics about block-device I/O operations and makes them available through the /proc file system, yet little information is reported about their timing.

Network packet sniffers [13] can be used to capture and analyze network traffic of network-based file systems [9]. This technique is useful for analyzing a protocol, but both the client and server often do additional processing that is not captured in the trace: searching caches, allocating objects, reordering requests, and more. iSCSI can also be traced at the network layer, and Fiber Channel connections can be traced at a hardware level [15]. These techniques share the fundamental disadvantage that they can only capture block-level information. Additionally, Fiber Channel tracing requires specialized hardware.

There are several methods for integrating profiling into code. The most popular one is direct source code modification because it imposes minimal overhead and is usually simple. For example, tracking of lock contention or I/O activity may require modification of several fixed places in the kernel source code to profile a variety of file systems. If, however, every function requires profiling modifications, then the compiler may conduct such an instrumentation. For example, gcc -p automatically inserts calls to mcount at the beginning and at the end of every function. The custom mcount function is then called on each function entry and exit point. More sophisticated approaches include run-time kernel code instrumentation [28] and layered call interception. For example, the Linux Security Modules [32] allow control interception for many OS operations. Similarly, stackable file systems allow one to insert a thin layer between the VFS and other file systems [33].

Often, to profile a file system, various combinations of benchmarks and microbenchmarks are run on top of the file system [5, 10, 21, 2], and conclusions are drawn from those results. Unfortunately, this is an inaccurate method of file system profiling, because these benchmarks operate in terms of system calls, which do not map precisely to file system operations. Often, the only result from a benchmark is a single metric (e.g., elapsed time or transactions per second). For complex benchmarks, these metrics can hide fundamental or pathological system behavior that may become visible under different workloads.

7. CONCLUSIONS

We designed and developed FSprof, the first tool specifically created to profile file systems. FSprof produces accurate profiles. FSprof's profiles are generated based on VFS operations, not system calls, block-level I/O operations, or CPU utilization. FSprof's profiles capture operations as the file system implements them, which is more useful for analysis. File systems perform both CPU and I/O-intensive tasks; FSprof accounts for both of these important components. FSprof is efficient: it has low per-operation overhead of less than 100 CPU cycles. When run with an I/O-intensive work-load, FSprof's elapsed time overhead is less than 1%.

We used FSprof to collect and analyze profiles for several popular file systems (Ext2, Ext3, Reiserfs, and Wrapfs), in only a short period of time. To aid this analysis, we developed automatic processing and visualization scripts to present FSprof results clearly and concisely. We discovered, investigated, and explained bi-modal and tri-model latency distributions within several common VFS operations. We also identified pathological performance problems related to lock contention. We quantified the effects of double caching, a technique used by many file system interception architectures, on stackable file systems. Finally, we profiled several large compile benchmarks, and found that they do not follow a fixed pattern. For example, traditional metrics like, lines of code and number of source files, do not sufficiently describe a compile benchmark because different compilation patterns generate sufficiently different VFS operations mixes.

7.1 Future Work

We plan to instrument the Linux dirty-buffer flushing daemon, bdflush. Currently, asynchronous writes are not handled by the VFS, but rather by block device drivers. The bdflush daemon directly invokes block device driver functions after the file system marks a buffer as dirty. We will modify bdflush such that these buffers can be tied to the correct FSprof file-system profile.

We plan to use FSprof to characterize more workloads, and how they affect file system operations. We also plan to use FSprof to examine how the OS's state affects benchmark results. For example, many researchers unmount the tested file system between tests. However, this clears only part of the caches (this is why we also ran *chill* to purge the caches). The location of meta-data and data on the disk changes over the time, which may result in significant performance changes. We plan to quantify the effects of file-system aging using FSprof.

We plan to develop a stackable VFS interception mechanism that does not require separate copies of objects (e.g., inodes, dentries, or pages). It will replace operation vectors dynamically with FSprofinstrumented wrappers. This will reduce the stacking overhead, remove influences on OS caches, and simplify stackable code paths.

FSprof adds a small overhead on each function call, yet this overhead can differ from machine to machine. FSprof can automatically measure its self-delay on startup and subtract it from the profiling results to increase the profiling resolution for very fast operations.

8. REFERENCES

- AT Atachment with Packet Interface 7: Volume 3 Serial Transport Protocols and Physical Interconnect (ATA/ATAPI-7 V3). Technical Report T13/1532D, Revision 4b, International Committee on Information Technology Standards (INCITS), April 2004.
- [2] 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.
- [3] Bell Laboratories. *prof*, January 1979. Unix Programmer's Manual, Section 1.
- [4] 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, page 42, 2000.
- [5] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *Proceedings* of the Annual USENIX Technical Conference, FREENIX Track, pages 259–274, Monterey, CA, June 2002.
- [6] 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*, October 2000.
- [7] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.
- [8] 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.
- [9] 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.
- [10] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, San Antonio, TX, June 2003.
- [11] Gnuplot Central. www.gnuplot.info.
- [12] 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.
- [13] L. N. R. Group. The TCPDump/Libpcap site. www.tcpdump.org, February 2003.
- [14] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html.
- [15] LeCroy. FCTracer 4G. www.catc.com/products/FCTracer4G.html, 2004.

- [16] M. K. McKusick. Using gprof to tune the 4.2BSD kernel. http: //docs.freebsd.org/44doc/papers/kerntune.html, May 1984.
- [17] 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):181–97, August 1984.
- [18] C. Minshall. fsx. http://cvsup.freebsd.org/cgi-bin/cvsweb/ cvsweb.cgi/src/tools/regression/fsx/, December 2001.
- [19] A. Morton. sleepometer. www.kernel.org/pub/linux/ kernel/people/akpm/patches/2.5/2.5.74/2.5. 74-mml/broken-out/sleepometer.patch, July 2003.
- [20] 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 137–52, Boston, MA, June 1994.
- [21] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy. A Performance Comparison of NFS and iSCSI for IP-Networked Storage. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 101–114, San Francisco, CA, March/April 2004.
- [22] H. Reiser ReiserFS. www.namesys.com/v4/v4.html, October 2004.
- [23] 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.
- [24] M. Russinovich. Inside Win2K NTFS, Part 1. www.winnetmag.com/Articles/ArticleID/15719/ pg/2/2.html, November 2000.
- [25] J. Seward, N. Nethercote, and J. Fitzhardinge. Valgrind. http://valgrind.kde.org, August 2004.
- [26] Silicon Graphics, Inc. Kernprof (Kernel Profiling). http://oss.sgi.com/projects/kernprof, 2003.
- [27] Sun Microsystems. Analyzing Program Performance With Sun Workshop, February 1999. http://docs.sun.com/db/doc/805-4947.
- [28] A. Tamches. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. PhD thesis, University of Wisconsin-Madison, 2001.
- [29] K. Thompson and D. M. Ritchie. *tm*. Bell Laboratories, November 1971. Unix Programmer's Manual, Section 1.
- [30] A. Veitch and K. Keeton. The Rubicon workload characterization tool. Technical Report HPL-SSP-2003-13, Storage Systems Department, Hewlett Packard Laboratories, March 2003. www.hpl.hp.com/research/ssp/papers/ rubicon-tr.pdf.
- [31] P. F. Velleman and D. C. Hoaglin. Applications, Basics, and Computing of Explaratory Data Analysis. Duxbury Press, Boston, MA, USA, 1981.
- [32] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. K. Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.
- [33] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.