The Effects of Memory-Rich Environments on File System Microbenchmarks

An-I A. Wang, *Computer Science Department, UCLA, awang@lasr.cs.ucla.edu*
Geoffrey Kuenning, *Computer Science Department, Harvey Mudd College, geoff@cs.hmc.edu*
Peter Reiher, Gerald J. Popek,[*] *Computer Science Department, UCLA, {reiher, popek@lasr.cs.ucla.edu}*

Keywords: L2 cache, memory, file system, benchmark, microbenchmark.

Abstract

File system performance has been greatly influenced by disk caching mechanisms. As the size of memory increases, common workloads are more likely to run completely from memory, and the effects of L2 caching and underlying hardware are becoming more visible.

This paper investigates performance anomalies observed when measuring and comparing the memory performance of various leading file systems. We discovered that without considering the effects of L2 caching policy, memory footprints of file systems, states of L2 cache, and memory page alignments, existing microbenchmarks could produce numbers that are significantly misleading and could result in poor designs and improper conclusions about relative file system performance.

We recommend that the design of file system microbenchmarks for memory-rich environments should carefully consider the initial states of memory and L2 cache, subtle warm-up effects, the cache eviction policy, interactions between workload size and the memory and cache sizes, and the management granularity of memory and the L2 cache.

1   INTRODUCTION

The performance of modern file systems is heavily dependent on multi-level caching, and minor tuning of the caching mechanisms can have huge effects on the final results. In traditional benchmarking environments where disk is the primary storage medium, the major caching effects on performance come from cached disk blocks in memory. However, as the memory size increases, more programs execute entirely from memory with no disk activity. The amount of memory available has made caching of disk contents vastly more effective. However, one result of this change is that the effects of L2 caching are becoming increasingly visible in benchmarks and in real programs. In particular, microbenchmark numbers, which guide many important file system design decisions, are now highly susceptible to the subtleties of underlying hardware.

Currently, popular file system benchmarks are designed to measure the file system's disk performance. Exercising the disk with a sizable working set is still the dominant practice. An implicit assumption was that memory performance numbers could always be obtained from these benchmarks by simply shrinking the size of the working set below the size of the physical memory. However, we discovered that benchmarks also need to consider L2 caching effects to provide true insight into file system performance.

Through the process of examining performance anomalies encountered when comparing memory-based and disk-based file systems, this paper demonstrates how the L2 caching policy, memory page alignment, the state of the L2 cache, and the sizes of file system footprints can significantly affect file system microbenchmark numbers in non-intuitive ways. To the extent that file systems are designed or chosen for use based on the results of such benchmarks, these effects can be misleading. In some cases, these effects cause a 20% or greater deviation in the key metrics produced by these benchmarks. This paper (Section 7) also makes a number of recommendations on how to design microbenchmarks to measure file system performance in the emerging memory-rich computing environments.

To our best knowledge, this study is the first investigation into the memory performance of systems using existing microbenchmarks that are designed to measure the disk performance of file systems.

2   MICROBENCHMARK ANOMALIES REVEALED

During the microbenchmarking phase of our research on the *Conquest* memory-based file system [18], we encountered a number of performance anomalies that seemed inexplicable.

The performance study involved both memory-based and disk-based file systems. The memory-based file systems were *ramfs* by Transmeta [14] and *Conquest*, and the disk-based file systems were *ext2fs* [2], *reiserfs* [8], and *SGI XFS* [16]. Without examining the details of individual file systems, intuitively memory-based file systems should be able to outperform disk-based file systems under all circumstances. However, the results were surprising.

The first two major anomalies were found using the Sprite LFS large-file microbenchmark [12]. This discovery was particularly unexpected because of its popularity for evaluating file systems [12, 13, 9, 15].

---

[*] Gerald Popek is also associated with United Online.

The Sprite LFS large-file microbenchmark writes a large file sequentially (with `fsync`), reads from it sequentially, writes a new large file randomly (with `fsync`), reads it randomly, and finally reads it sequentially. The final read phase was originally designed to measure an important case for a log-structured file system. The file size we used was 1 MB. This file fit into main memory for all file systems tested. The detailed experimental settings are described in Section 3.
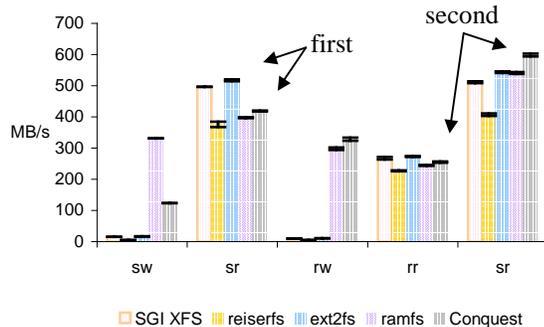
Figure 1: Sprite LFS large-file microbenchmark for one 1-MB file. The benchmark consists of five phases—sequential write, sequential read, random write, random read, and sequential read. The graph shows two major anomalies: (1) For the first sequential read, memory-based file systems are slower than some of the disk-based file systems; (2) Memory is believed to provide a relatively uniform access speed for any access patterns. However, random read is slower than the second sequential read.

Figure 1 shows our early microbenchmark results and raises questions regarding two major anomalies:

1. ***Why are memory-based file systems slower than some of the disk-based file systems in sequential read performance?*** In particular, why are *Conquest* and *ramfs* significantly slower than *ext2fs* and *SGI XFS* in terms of the bandwidth for the first sequential read? This question is puzzling because the code bases for *Conquest* and *ramfs* are significantly smaller and simpler than those of *ext2fs* and *SGI XFS*. Smaller and simpler code should intuitively translate into better performance (and, indeed, it usually does).
2. ***Why are random memory reads significantly slower than sequential memory reads?*** Conceptually, the access times for any random memory location should be relatively uniform, unless optimizations accelerate the sequential accesses. However, since data is moved in 4-KB blocks, most bytes should be transferred sequentially.

To conduct detailed studies of these anomalies, we chose to use the Sprite LFS large-file microbenchmark for its relative ease of use for instrumentation, profiling, and analyses.

3 EXPERIMENTAL SETTINGS

For ease of presentation and discussion, the remainder of the paper compares Transmeta's *ramfs* to the popular *ext2fs*.

The measurements for *Conquest*, *reiserfs* and *SGI XFS* are omitted because the *ramfs* and *ext2fs* numbers capture the relevant differences between disk-based and memory-based file systems. *Conquest*-specific questions (such as, why *Conquest* can outperform *ramfs*) are covered in [18].

Both *ramfs* and *ext2fs* follow the VFS interface and use the same generic read and write routines provided by the VFS. However, *ramfs* uses the caching data structures under VFS to store file system contents and metadata directly, and those temporary data structures do not provide persistence of data after a system reboot. *Ramfs* should approximate the practical achievable bound for memory performance of file systems. *Ext2fs*, on the other hand, uses disk as the final storage destination for data and metadata. *Ext2fs* is also one of the most widely used file systems in the UNIX world, and it outperforms other disk-based file systems on a wide variety of benchmarks [2].

Table 1: Experimental platform.

Experimental platform	
Manufacturer model	Dell PowerEdge 4400
Processor	1 GHz 32-bit Xeon Pentium
Processor bus	133 MHz
Memory	4x512 MB, Micron MT18LSDT6472G, SYNCH, 133 MHz, CL3, ECC
L2 cache	256 KB Advanced
Disk	73.4 GB, 10,000 RPM, Seagate ST173404LC
Disk partition for testing	6.1 GB partition starting at cylinder 7197
I/O adaptor	Adaptec AIC-7899 Ultra 160/m SCSI host Adaptor, BIOS v25306
OS	Linux 2.4.2

Our experimental platform is described in Table 1. Both the *ramfs* and *ext2fs* file systems were mounted and created with default settings. For each file system, the performance numbers were collected over six runs, but averaged over only the last five runs to avoid warm-up effects. All results are presented at the 90% confidence level.

4 WHY IS RAMFS SLOWER THAN EXT2FS?

As shown in Figure 1, *ramfs* is slower than *ext2fs* during the first sequential read phase of the LFS benchmark. This section describes a series of hypotheses and experiments performed to trace the cause of this memory-based file system running slower than the disk-based one.

4.1 Sequential Read Differences Between *ext2fs* and *ramfs*

Since both *ramfs* and *ext2fs* share the same generic code under VFS, when workloads fit in the main memory we expect both file systems to achieve similar performance, but perhaps reaching a steady state at different speeds. Therefore, we ran an experiment in which we performed two sequential reads after the sequential write.

Table 2: Test of repeated sequential reads. A sequential write (with `fsync`) is followed by two sequential reads.

	ext2fs	*ramfs*
sequential write (MB/s)	14 ($\pm$0.29)	390 ($\pm$0.80)
sequential read (MB/s)	520 ($\pm$ 1.4)	400 ($\pm$0.65)
sequential read (MB/s)	530 ($\pm$ 0.65)	520 ($\pm$2.5)
sequential read (MB/s)	530 ($\pm$2.6)	530 ($\pm$0.73)

Table 2 shows that a warm-up effect for sequential reads is evident for both file systems, and *ramfs* achieves nearly the same performance as *ext2fs* in the steady state. This somewhat confirms our intuition that the shared code base should result in similar performance, but there is still a significant warm-up effect to explain.

Why does ext2fs warm up faster than ramfs? We had several hypotheses: (1) The default file system on the test machine is *ext2fs*; therefore, path resolutions (for various files and shared libraries) that involve *ext2fs* may cause contention for cache lines with *ramfs* during benchmarking. In a personal communication, Linus Torvalds [17] also speculated that: (2) *ramfs* may have cache layout problems, and (3) *ramfs* may have a different warm-up behavior for sequential reads than *ext2fs* because of different data cache buffer use.

We attempted to examine various hypotheses using the Linux kernel profiling facility, but it yielded inconclusive results. Detailed profiling significantly distorted the collected numbers at the microsecond scale.

Then we tried one sequential read on a pre-existing 1-MB file, followed by another sequential read.

Table 3: Test of the warm-up behavior for sequential reads

	ext2fs	*ramfs*
sequential read (MB/s)	480 ($\pm$0.98)	480 ($\pm$1.3)
sequential read (MB/s)	520 ($\pm$ 2.7)	520 ($\pm$0.0)

Table 3 shows that both *ramfs* and *ext2fs* have similar numbers for two consecutive sequential reads, suggesting that our various hypotheses are unlikely candidates to explain the fast warm-up effects of *ext2fs*. Since both sets of numbers are statistically the same, even if external factors such as the choice of default file systems exist, the effects are likely to be trivial. If *ramfs* has cache layout problems, its performance here should be noticeably slower than *ext2fs*. Also, both file systems warm up at similar rates, so different ways of using the memory data cache do little to explain the anomaly.

4.2 Were the Initial States of the Microbenchmark Bad?

As a sanity check, we took a number of measurements to ensure the same initial states for various experiments. We rebooted our benchmarking machine after each experimental run, but the booting process might still lead to inconsistent caching states. We modified the original Sprite LFS large-file microbenchmark to perform a sequential read on a pre-existing 1-MB file followed by a sequential write (with `fsync`) and a sequential read on the same file. This precaution should ensure that all experiments started with the same state after performing the first sequential read.

Table 4: Initial condition test. A sequential read was performed on a pre-existing 1-MB file, followed by a sequential write (with `fsync`) and a sequential read.

	ext2fs	*ramfs*
sequential read (MB/s)	480 ($\pm$1.5)	480 ($\pm$1.4)
sequential write (MB/s)	14 ($\pm$ 0.47)	400 ($\pm$1.2)
sequential read (MB/s)	520 ($\pm$ 0.73)	400 ($\pm$1.3)

Table 4 shows that both *ext2fs* and *ramfs* have similar performance for the first sequential read, confirming that both sets of experiments have similar initial states. Therefore, the cause of the anomaly is likely to be the sequential write. However, ***why would the same sequential write operation cause the subsequent sequential read to speed up by 40 MB/s for ext2fs, while slowing it down by 80 MB/s for ramfs***? Is it possible that a sequential write somehow warms up the L2 cache locality for *ext2fs* and destroys the locality for *ramfs* for the subsequent sequential read?

4.3 Does L2 Caching Buffer Management Cause the Anomaly?

Linux's memory manager provides high-speed memory allocation by writing back dirty memory pages to disk speculatively in several ways. Notably, if the memory is being allocated at a rapid rate, the memory write-back will be performed at a higher rate to anticipate near-term allocations. Also, whenever the allocated memory exceeds a certain threshold, the memory manager will also start writing back dirty pages to avoid an exhaustive search for allocations.

L2 caching tries to handle bursty allocations in a different manner. The Pentium® III has a streaming I/O option to bypassing L2 caching for large sequential accesses [7]. Should the high I/O bandwidth of *ramfs* trigger the streaming I/O mechanism, it would leave the L2 cache with little content available for reuse during the subsequent read operation.

Therefore, we designed an experiment to test the write request rates on the triggering of any streaming I/O mechanism. For the sequential write, we inserted a `usleep` call to sleep 1 μs within each iteration of write call, so the write request rate was significantly slowed down to a similar speed for both *ramfs* and *ext2fs*. If either the cache buffer management or the streaming I/O mechanism caused this performance anomaly, we should then see sequential read results significantly different from the last row of Table 4.

Unfortunately, Table 5 shows that the rates of write operations have little effect on the sequential read bandwidth, disproving hypotheses involving special sequential hardware logic as the explanation for the anomaly.

Table 5: L2 caching policy test. Each `write` system call is followed by a sleep of 1 μs. The slow sequential write of a file is followed by a normal sequential read.

	ext2fs	ramfs
sequential write (with `usleep`) (KB/s)	200 ($\pm$0.32)	200 ($\pm$0.40)
sequential read (KB/s)	510 ($\pm$ 1.9)	410 ($\pm$5.5)

4.4 Do Random Writes Also Cause the Anomaly?

Results obtained from the slow sequential write experiment raised the suspicion that the sequential nature of a write may contribute little to this anomaly. Therefore, we conducted an experiment with a random write (with `fsync`) followed by a sequential read.

Table 6: Test on effects of random writes. A random write (with `fsync`) is followed by a sequential read.

|	ext2fs	ramfs
random write (MB/s)	9.2 ($\pm$0.27)	280 ($\pm$2.3)
sequential read (MB/s)	510 ($\pm$ 1.4)	400 ($\pm$1.4)

Table 6 shows that a random write can also cause abnormal behavior in the subsequent sequential read. Therefore, any write access can cause a subsequent read to achieve higher bandwidth for *ext2fs* and lower bandwidth for *ramfs*, pointing toward a completely different hypothesis.

4.5 Forming a New Hypothesis

One puzzling aspect of this performance anomaly was the inverse association between the size of the file system memory footprint and the performance. From the code base, the cache footprint of *ramfs* was significantly smaller than *ext2fs*. But, **why would a smaller cache footprint result in worse performance?**

Given that the Sprite LFS large-file microbenchmark measures the bandwidth of read and write operations, the entire benchmark should be data-intensive, and the timing should be strongly correlated to the amount of data being transferred into and out of the L2 cache. Since microbenchmarking experiments revealed only the black-box behavior of the system, a simple and naïve model of the L2 cache was used to visualize the underlying mechanisms within the black box.

Suppose an L2 cache has four cache lines, and the first two phases of the microbenchmark operate on a file with a size of 16 file segments, with each segment occupying one cache line. For the scenario of a large cache footprint, the file system and the benchmark being run have an active footprint of three out of four cache lines (Figure 2). This simple model assumes that the model starts with only the active cache footprint in the cache. The model also assumes that the effects of cache collisions are minimal, and that cache collisions will never evict the cache lines containing the active footprint.
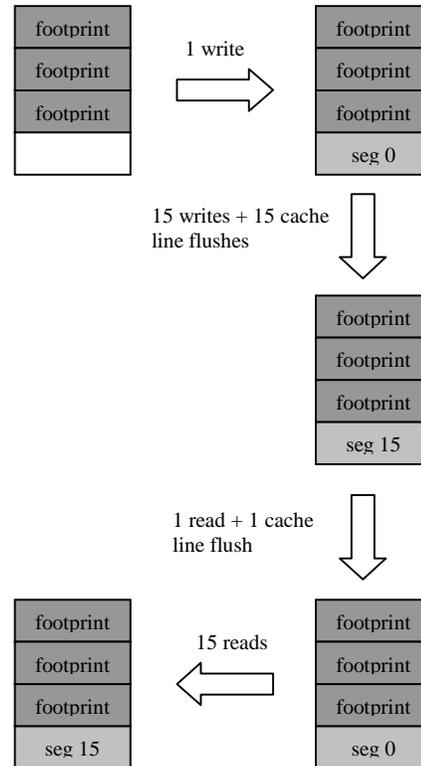
Figure 2: The model for a large cache footprint. The first two phases of the Sprite LFS large-file benchmark are analyzed with a simple L2 cache with four cache lines. The active footprint of the file system and benchmark being tested consists of three out of four cache lines, and the file being tested has a size of 16 segments, with each segment occupying one cache line.

During the sequential write phase, the first write can directly modify the clean cache line, and all subsequent writes require purging the dirty cache content before modification. During the sequential read phase, the first read requires purging the dirty cache content, because the previous write operation leaves the end of the file in the last cache line, and the read cannot reuse the content. However, all subsequent reads can simply overwrite the previously read-in cache line from memory, since those lines are not modified during the read operation and thus require no eviction for committing changes.

For the scenario of a small cache footprint, the active footprint is decreased to one cache line (Figure 3). During the sequential write phase, the first three writes can directly modify the clean cache lines, and 13 subsequent writes require purging dirty cache content. During the sequential read phase, the first three reads need to purge the dirty cache content since cache stores the last three cache lines of the file being benchmarked, and the read starts from the beginning. The other 13 reads can simply overload the previously read-in cache lines from memory.
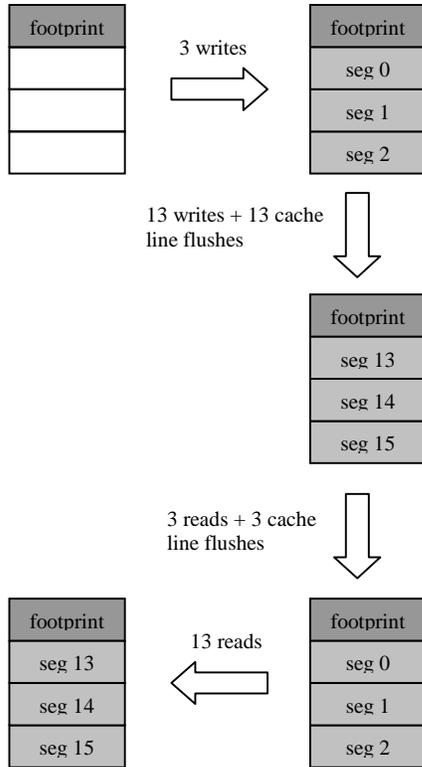
Figure 3: The model for a small cache footprint. The first two phases of the Sprite LFS large-file benchmark are analyzed with a simple L2 cache with four cache lines. The active footprint of the file system and benchmark being tested consists of one cache line, and the file being tested has a size of 16 segments, with each segment occupying one cache line.

Table 7 summarizes the differences between the two scenarios. Although simple, this model shows that it is possible for a smaller cache footprint to result in a slower sequential read after a sequential write. A smaller cache footprint leaves more room to cache dirty data during the sequential write phase, and the sequential read phase needs to finish committing previous dirty cache content before reading new information back to cache. In essence, the file system with the larger footprint has already paid nearly the complete price for cache flushes during its write phase, while the file system with the smaller footprint has used the cache to defer some of the costs until later.

Table 7: Summary of the first two phases of Sprite LFS large-file macrobenchmark results using a simple cache model with four cache lines and a file with a size of 16 cache lines.

	Large cache footprint	Small cache footprint
sequential write	16 writes, 15 flushes	16 writes, 13 flushes
sequential read	16 reads, 1 flushes	16 reads, 3 flushes

This hypothesis also explains why the anomaly of the sequential read occurs also after a random write, since a smaller cache footprint leaves more room to cache dirty data for any write patterns.

4.6   Verifying the New Hypothesis

We tried to verify our hypothesis through low-level instrumentation, high-level microbenchmarking, and reproducing the same behavior at the user-level.

4.6.1 Performance Counter Instrumentation

Our first measurement was of the amount of data being transferred into and out of the L2 cache. The instrumentation tool we used was Rabbit [5], a performance-counters library for Intel processors running Linux. The package allowed us to monitor a few registers with little or no disturbance to the overall performance numbers, even for microbenchmarks. We monitored CPU registers that contain information regarding the movement of data for the L2 cache. We counted the total number of events and also tracked the elapsed times to compute the volume of data being moved.

At the beginning of a sequential read after a write operation, *ramfs* has to move 230 KB of modified data out of the 256-KB L2 cache, while *ext2fs* moves only 66 KB of modified data from the L2 cache. Given that a write operation precedes the read operation, the moved bytes are likely to represent the recently written part of the 1-MB file. Also, given that *ramfs* has a smaller cache footprint than *ext2fs*, the L2 cache can hold more dirty data when running *ramfs* than *ext2fs*. Since both file systems need to read in the 1 MB of data after moving the dirty data out of the cache, the discrepancy in data volume being moved explains why *ext2fs* can achieve higher read bandwidth than *ramfs*.

4.6.2 Microbenchmark Behaviors

At the microbenchmark level, we first ran an experiment with a few sequential write operations (with `fsync`). We expected the first sequential write to affect the performance of the second sequential write. Also, starting from the second sequential write, the bandwidth numbers should not fluctuate as much.

Table 8: Test on repeated sequential writes.

	ext2fs	*ramfs*
sequential write (MB/s)	13 ($\pm$0.61)	380 ($\pm$1.4)
sequential write (MB/s)	16 ($\pm$ 0.32)	350 ($\pm$0.80)
sequential write (MB/s)	16 ($\pm$ 0.32)	350 ($\pm$0.65)

Table 8 shows that the second sequential write for *ext2fs* has an increased bandwidth (probably due to the caching of *ext2fs* file system code), while the second sequential write for *ramfs* has a reduced bandwidth (probably because of the need to purge many dirty cache lines from the previous sequential write).

Originally, we planned to modify *ramfs* to use a larger L2 footprint, but altering the source code would make our test methods less portable and repeatable for future investigations. Instead, we used a 128-KB file to see if *ramfs* can outperform *ext2fs* when both the file being tested and the *ramfs* footprint can fit into the 256-KB L2 cache.

(Note that since *ext2fs* flushed 66 KB of data in the previous experiment, we expected that the 128-KB file and the *ext2fs* footprint would not fit into the 256-KB L2 cache.) The experiment consists of a sequential write (with `fsync`), followed by two sequential reads.

Table 9: L2 cache test. A sequential write on a 128-KB file (with `fsync`) is followed by two sequential reads.

	ext2fs	*ramfs*
sequential write (MB/s)	4.9 ($\pm$0.43)	600 ($\pm$0.61)
sequential read (MB/s)	440 ($\pm$ 5.7)	810 ($\pm$25)
sequential read (MB/s)	1200 ($\pm$ 35)	1400 ($\pm$52)

Table 9 shows that when *ramfs* can read recently written content completely from the L2 cache, it can significantly outperform *ext2fs*.

4.6.3 Testing a User-Level Program

To extend the applicability of our findings, we created a trivial array-indexing program to reproduce the write-back effect of L2 caching. The user program walks through an integer array of 1 MB, assigning each array entry a different integer, followed by two loops that retrieve the value of each array entry to a variable. Although they were designed to exercise the same behavior as the benchmark, these types of operations are reasonably realistic in various applications.

Table 10: A user-level test.

	User program
sequential write (MB/s)	220 ($\pm$0.32)
sequential read (MB/s)	150 ($\pm$ 31)
sequential read (MB/s)	200 ($\pm$ 1.3)

Table 10 shows an anomaly similar to that in the microbenchmark, verifying that the effects of L2 caching are easily visible in places other than simple file system microbenchmarks.

However, these numbers have peculiarities of their own. These bandwidth numbers appear to be low compared to Table 2, and the sequential write bandwidth is slightly higher than the sequential read bandwidth. We do not yet have an explanation for these peculiarities, since a detailed investigation would be an unnecessary digression for the purpose of this paper. However, this user program does show the difficulty of designing microbenchmarks that actually match real program behavior.

4.6.4 Lessons Learned

The numbers obtained from these three levels of experiments teach us several lessons:

- Intuitively, a smaller cache footprint should yield faster performance, but under certain workloads, a smaller footprint can also increase the effects of a write-back cache policy, causing the performance of a subsequent phase of a benchmark to swing in either direction.

- The effects of memory and L2 caching are easily visible at the application level, beyond file system microbenchmarks.
- To measure the memory performance of file systems under microbenchmarks, we have to consider the L2 cache states at each phase of measurement.

Incorporating the states of the L2 cache into the design of microbenchmarks can introduce some complexity, since the cache size, cache line eviction policy, and n-way associativity are different from machine to machine. Also, microbenchmarks need to consider whether the dependencies between adjacent file system operations are intended and reflective of actual access patterns.

4.6.5 Modifications to the Sprite LFS Large-File Microbenchmark

Using these lessons, we modified the Sprite LFS large-file microbenchmark in the following way: Each phase operates on 11 files (instead of a single file) in a fixed order, and the numbers are averaged across the last 10 files. The numbers collected from the first file are dropped to reduce the warm-up effect of L2 caching on final results and to collect the steady state numbers for each phase of the microbenchmark.

Figure 4 presents the results of this newer version of the microbenchmark for *ext2fs* and *ramfs*. The anomaly of *ramfs* being slower than *ext2fs* has disappeared. Both *ext2fs* and *ramfs* performed equally well on reads since the file content was now read from the memory buffer. However, another anomaly still exists: namely, sequential memory reads are much faster than random memory reads.
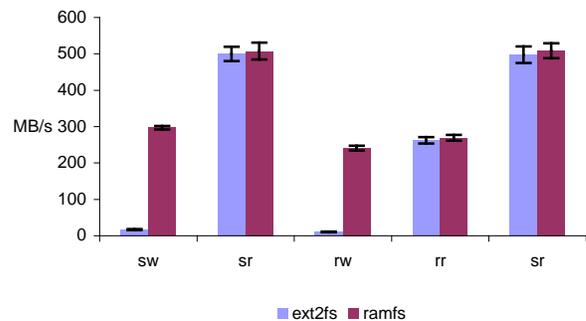
Figure 4: Sprite LFS large-file microbenchmark, with file size equal to 1 MB, and each phase repeated ten times. The benchmark consists of five phases—sequential write, sequential read, random write, random read, and sequential read.

5 WHY ARE RANDOM MEMORY ACCESSES SLOWER THAN SEQUENTIAL ACCESSES?

The performance of random memory accesses could differ from that of sequential memory accesses for a few reasons. First, sequential accesses might be accelerated by either hardware or software sequential prefetching logic. Second, each random file access involves an extra seek system call, incurring extra kernel-crossing overhead. Third, memory hardware might have a faster sequential access mode. Since the upper bits of memory addresses

usually stay the same for sequential accesses, memory accesses might not need to wait for the upper bits of a memory address to stabilize before accessing the memory content.

Unfortunately, none of these three hypotheses plausibly explain the performance gap we encountered. First, since both random and sequential accesses are performed at the granularity of 4-KB blocks in the benchmark, most bytes are transferred sequentially, leaving a minimal effect on random and sequential read performance due to sequential logic or memory hardware.

In addition, given that sequential reads achieve about 500 MB/s for a 1-MB file, the elapsed time to perform a sequential read is about 2 milliseconds; a random read is 4 milliseconds. Unless the overhead of locating a random memory block or making an extra system call can cost 2 milliseconds, this hypothesis cannot explain why random memory reads are so slow.

Therefore, we once again resorted to the Rabbit performance-counter library to monitor CPU registers during the microbenchmark. We paid special attention to corresponding *ext2fs* and *ramfs* numbers that differed by more than a factor of two. What caught our attention was that random memory accesses incurred many misaligned memory accesses, while sequential memory accesses incurred none.

From the source code of our microbenchmark, we found that random memory accesses were indeed unaligned.[1] Therefore, such accesses usually required fetching data from two memory words, while the same-sized access in the sequential mode fetched data from only one word. Thus, each memory reference by the random read phase made two memory accesses most of the time. This effect caused the random memory read bandwidth to be half of the sequential read bandwidth. Many applications that perform so-called *random* reads (e.g., linkers) do so on word-aligned or page-aligned boundaries, so the benchmark is not necessarily indicative of real-world performance.
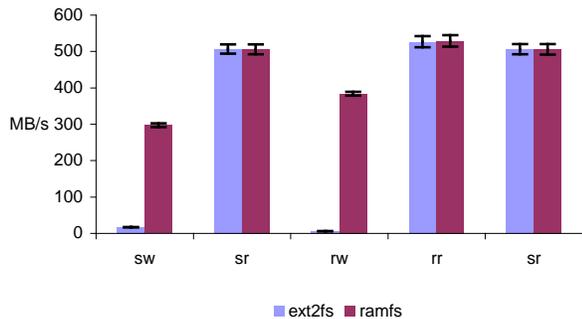
Figure 5: Sprite LFS large-file microbenchmark, with file size equal to 1 MB, each phase repeated ten times, and memory-aligned random memory reads. The benchmark consists of five phases—sequential write, sequential read, random write, random read, and sequential read.

---

[1] We used a version of the Sprite LFS benchmark downloaded in the spring of 2001. A later version of the benchmark performs random accesses on page-aligned boundaries, so apparently this benchmark problem was independently discovered and fixed in the interim.

After finding the cause of the anomaly, we modified the microbenchmark to perform page-aligned random reads. Figure 5 shows a newer version of the data for the Sprite LFS large-file microbenchmark, with the additional modification of aligned random memory reads.

However, we spotted yet more glitches to explain. ***Why are random memory accesses now faster than sequential memory accesses?*** Also, ***why is the performance difference between random and sequential memory writes larger than that between corresponding reads?***

6 WHY ARE RANDOM MEMORY ACCESSES FASTER THAN SEQUENTIAL ACCESSES?

The performance differences between sequential and random memory performance can be explained through probability analysis. Caching works because of the possibility of reusing cache content. When a file is larger than the cache size, sequential accesses provide no opportunity for cache reuse before a given line is replaced, while random accesses will sometimes hit in the cache.

The performance difference between a random and a sequential write is greater than the performance gap for corresponding reads because of the relative cost of cache misses. When reading a 1-MB file from memory through a 256-KB L2 cache, initial read misses might involve some purging of dirty L2 cache content to memory. Once the L2 cache is filled with read-only content, subsequent read misses will likely involve reading the file content into the cache directly, with few dirty cache lines to evict. On the other hand, a write miss (occurring when accessing a 1-MB file from memory through a 256-KB L2 cache) is very likely to involve evicting a dirty cache line, reading in the target cache content, and modifying the cache line. For a write operation, since the disadvantage of a cache miss is greater, the relative advantage of a hit is also greater. Therefore, the performance difference between random and sequential writes is greater than that of reads.

A microbenchmark that operated on 128-KB files was used to verify the above hypothesis. One would expect no performance differences among file operations for the following reasons: (1) both sequential or random write are less likely to flush the dirty L2 cache content because the file size is small compared to the size of the L2 cache; (2) since an `fsync` call is invoked at the end of each file close, and dirty L2 cache content is flushed, the subsequent write operation does not need to commit the dirty data for the previous write; and (3) without the need to flush existing dirty L2 cache content, the cost of write misses should be similar to the cost of read misses, assuming that data migration is the dominating performance cost.

With the expectation of uniform performance across all file operations, once again we see surprising results in Figure 6: random accesses under *ramfs* are 20% slower than sequential accesses.
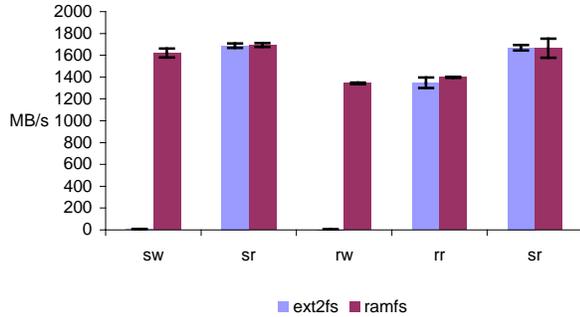
Figure 6: Sprite LFS large-file microbenchmark, with file size equal to 128 KB, each phase repeated ten times, and memory-aligned random memory reads. The benchmark consists of five phases—sequential write, sequential read, random write, random read, and sequential read.

Based on the differences between achieved bandwidths, we can compute the difference in elapsed time to be about 150 μs. The extra `lseek` system call and the random number generation call may account for this timing difference. To test this explanation, we padded sequential accesses with a dummy `lseek` call and a dummy random number generation call within each iteration of a file access.
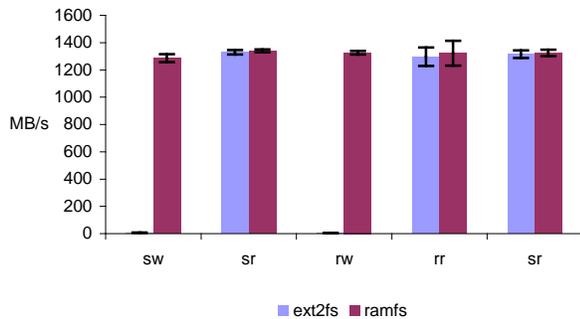
Figure 7: Sprite LFS large-file microbenchmark, with file size equal to 128 KB, each phase repeated ten times, and memory-aligned random memory reads. Sequential accesses are augmented with dummy `lseek` and random number generation calls. The benchmark consists of five phases—sequential write, sequential read, random write, random read, and sequential read.

Figure 7 finally shows the graph that we expected to see, with uniform memory access times for all file accesses under *ramfs*. However, this experiment points out that with the speed of the L2 cache on modern machines, the overhead of a system call and crossing the kernel boundary can potentially degrade file system throughput by 20%. This effect shows that performance numbers are very sensitive to minute details in the design of microbenchmarks and experiments. The effect should thus be considered in the design of data-intensive applications.

7   BONNIE++ MICROBENCHMARK

To see if similar anomalies are present in other microbenchmarks, we also tried the popular Bonnie++ benchmark (version 1.02) [4], which is largely a C++ version of the original Bonnie benchmark [1]. In addition to

the bandwidth tests in the original Bonnie, Bonnie++ adds six tests of file create/stat/unlink to simulate operations that are common bottlenecks on large Squid and INN servers and machines with tens of thousands of mail files under `/var/spool/mail`.

Anomalies were also observed during the last six tests of Bonnie++, which first sequentially `create` 1024 directories with 16 empty files each, sequentially `stat` those files in the same order of creation, and sequentially `unlink` (delete) them. The benchmark then `creates` same number of files in a random order to fill 1024 directories with 16 empty files each. The benchmark also `stats` and `unlinks` those files in a random order. Figure 8 presents the Bonnie++ benchmark results. All files can fit into the main memory.

Our previous findings largely explain the anomalies observed in the Bonnie++. The RAM-based file systems are slower than some of the disk-based file systems during the sequential `stat` phase because the RAM-based file systems need to commit more dirty data from L2 cache to memory at the beginning of the sequential `stat` than do the disk-based file systems. Also, for RAM-based file systems, random `stats` are significantly faster than sequential stats due to the probabilistic reuse of cache content.
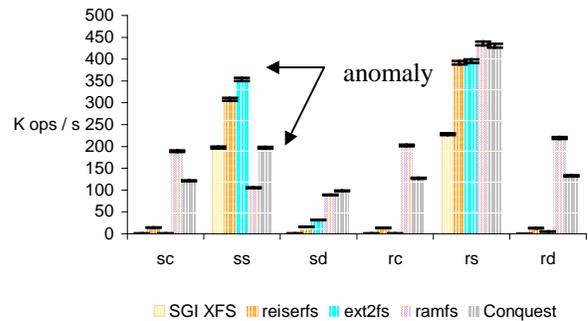
Figure 8: Last six tests of the Bonnie++ benchmark—sequential `create`, sequential `stat`, sequential `delete`, random `create`, random `stat`, and random `delete`. The benchmark first `creates` 1024 directories, each with 16 empty files, in a sequential order. It then `stats` and `unlinks` (deletes) those files in a sequential order. The benchmark then moves on to `create` files in a random order, `stat` those files in a random order, and `unlink` them in a random order. Note that for the test case of `stating` files sequentially, memory-based file systems are slower than some of the disk-based file systems.

8   RECOMMENDATIONS FOR MICROBENCHMARKING FILE SYSTEMS

As we move toward memory-rich environments, the increasingly visible effects of memory and L2 caching behaviors have far-reaching implications on microbenchmarking of systems in general.

Given that the main purpose of microbenchmarks is to measure individual design points (file operations in the case of file systems), the design of microbenchmarks for memory-rich environments should address the following concerns:

- **Assure identical initial states for memory and L2 cache.** Before conducting each experiment, the measurement machine should be rebooted when possible to reset the states of memory and L2 cache. In particular, because modern memory management involves pre-allocation and fragmentation management, simple deallocation of memory resources is unlikely to restore the memory and L2 cache states to their conditions prior to benchmarking.

- **Watch out for warm-up effects.** Numbers collected at the beginning of each phase of a microbenchmark should be treated with care. If warm-up effects are inherent to user patterns, they can be included for data analysis. However, if the intent is to measure the steady-state performance numbers, numbers collected during the warm-up period should be dropped, as in one of modifications we made to the Sprite microbenchmark (Section 4.6.5).

- **Avoid false accounting due to L2 cache policy.** Modern machines often use asynchronous mechanisms to speed up computation. The write-back policy of L2 caching is an example of how the timing of a read operation is distorted due to a prior write. One simple way to avoid this type of false accounting is to insert additional file operations between phases of benchmarks and measure only the steady-state numbers.

- **Vary the workload sizes across the size boundaries of L2 cache and memory.** Microbenchmark numbers strongly depend on the size of the workload relative to the size of the L2 cache and memory. In our Sprite benchmark example, changing the workload size effectively changes our conclusions about the relative access speeds of random and sequential accesses to memory content.

- **Watch out for cache-line and memory-management granularity.** In the case of the Sprite benchmark, the simple misalignment of random accesses contributes a factor-of-two performance difference compared to sequential memory accesses.

Of course, this list is by no means exhaustive. As program execution speed gets closer to hardware speeds, we expect to see more high-level performance behaviors that are sensitive to low-level hardware, such as L1 caching.

9   RELATED WORK

The most widely used benchmark in file system literature is the Andrew File System Benchmark [6], which captures the workload characteristics of a software development environment. As memory sizes have grown, this benchmark no longer stresses modern file systems because the data size is too small. The Modified Andrew Benchmark (MAB) [11] was introduced to stress more modern file systems. However, since the growth of memory size has outpaced the average program size over the years, running programs from memory is the common case, and the memory performance of file systems is more relevant to user experience. Even in cases where disk performance is still relevant, it is unclear that benchmarks like MAB capture the dynamics of those cases.

Chen and Patterson [3] questioned the relative evolutions of hardware and benchmarks and proposed a self-scaling benchmark to address the rapid growth of memory size. However, the benchmark framework they proposed is based on a synthetic workload, and the benchmark provides information more relevant at the macrobenchmarking level than at the microbenchmarking level. In other words, their benchmark does not reveal important characteristics of the performance of individual file system operations.

There is still wide disagreement on how to design benchmarks to measure the disk I/O performance of file systems [10]. However, due to the shifting characteristics of memory-rich environments, our work is, to our knowledge, the first investigation of the memory performance of file systems under existing microbenchmarks that are designed to measure disk performance.

10   FUTURE WORK

Through the examination of one simple microbenchmark, we uncovered a number of potential problems with applying existing disk-based file system benchmarks to measure the memory performance of file systems. One natural extension is to analyze additional popular benchmarks to see whether they have similar problems.

However, there is a more far-reaching question. File system designers and benchmarkers are familiar with disk performance asymmetries, and have designed their systems and benchmarks to take those into account. But as CPU speed continues to increase, file system performance is becoming more sensitive to the underlying hardware than ever before. Various asynchronous mechanisms expose additional unanticipated dependencies among high-level operations and file system performance numbers. The time may be coming for those who design both file systems and file system benchmarks to consider how other system asymmetries, such as those related to the L2 cache, affect their designs.

From Table 7, one might deduce that the cost of a write needs to occur eventually when dirty cache lines are flushed; therefore, we are just shifting costs around. However, this is not always the case. In particular, cache flushing might be avoided depending on the ordering of read and write accesses. For instance, if our 1-MB file is sliced into eight 128-KB access units and the reads and writes are arranged to reuse the L2 cache content more aggressively, *ramfs* could achieve significant performance improvements by reducing the total number of flushes. Compilers can certainly take advantage of these underlying behaviors to generate more efficient code. A more general question is how to take advantage of the low-level dependencies among various file operations to improve high-level applications.

One of our last experiments (Figures 6 and 7) suggests that the overhead of kernel crossing for system calls may be very significant relative to the speed of the L2 cache.

Perhaps multiple system calls should be bundled to amortize the cost of crossing the kernel boundary.

11 CONCLUSIONS

Comparing the performance of several file systems should have been a straightforward task. But we discovered that in a modern memory-rich environment, memory footprints, the write-back policy of the L2 cache, the state of the L2 cache, and memory alignment can all complicate performance numbers in non-intuitive ways.

Smaller L2 cache footprints can leave more room for caching dirty data, amplifying performance swings for immediately subsequent operations. Misaligned random memory accesses can incur a twofold performance loss compared to sequential accesses. On the other hand, when the L2 cache size is a significant fraction of the size of a file operation, cache hits can make random accesses faster than sequential ones. Finally, at the speed of modern L2 caches, the difference of a single kernel crossing induced by a system call can cause a significant performance difference.

Ideally, a microbenchmark should be able to take into account underlying hardware parameters automatically and generate meaningful results; however, this task is difficult due to the growing complexity of hardware. We believe that hardware performance characteristics need to be exposed to both the benchmark and operating system through an appropriate interface. Such transparency will be invaluable for the design of higher-level services, such as file systems, compilers, applications, and benchmarks themselves

Microbenchmarking file systems in memory-rich environments is significantly more difficult than measuring systems with disk as the major bottleneck. Modern benchmarking requires intimate knowledge of underlying hardware, such as the cache eviction policy and set associativity. It is also critical to set the initial states of memory and the L2 cache, understand subtle memory and L2-cache warm-up effects, and align accesses to the management granularity of memory and the L2 cache. Designing good benchmarks demands specialized expertise and a thorough understanding of the hardware. No longer can a quickly and casually composed benchmark be expected to give insightful pictures of performance behavior.

12 Acknowledgements

13 References

[1] Bray T, Bonnie Source Code, Netnews Posting, 1990.
[2] Card R, Ts'o T, Tweedie S. Design and Implementation of the Second Extended Filesystem. *Proceedings of the First Dutch International Symposium on Linux*, ISBN 90-367-0385-9, December 1994.
[3] Chen PM, Patterson DA. A New Approach to I/O Performance Evaluation—Self-Scaling I/O Benchmarks, Predicted I/O Performance. *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.
[4] Coker R. Bonnie++ Documentation. http://www.coker.com.au/bonnie++/readme.html. 2001.
[5] Heller DE. Rabbit, A Performance Counters Library for Intel Processors and Linux, Iowa State University, 2000.
[6] Howard J, Lazar M, Menees S, Nichols D, Satyanarayanan M, Sidebotham R, West M. Scale and Performance in a Distributed File System, *ACM Transactions on Computer Systems*, 6(1), pp. 51-81, February 1988.
[7] Keshava J, Penkovski V. Pentium® III Processor Implementation Tradeoffs, *Intel Technology Journal*, 1999.
[8] Namesys. http://www.namesys.com, 2002.
[9] Mazieres D, Kaminsky M, Kaashoek MF, Witchel E. Separating key management from file system security. *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.
[10] Mogul JC. Brittle Metrics in Operating Systems Research. *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pp. 90-95, March 1999.
[11] Ousterhout J. Why Aren't Operating Systems Getting Faster as Fast as Hardware? *Proceedings of the Summer 1990 USENIX Conference*, pp. 247-256, June 1990.
[12] Rosenblum M, Ousterhout J. The Design and Implementation of a Log-Structured File System. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.
[13] Seltzer M, Smith KA, Balakrishnan H, Chang J, McMains S, Padmanabhan V. File System Logging Versus Clustering: A Performance Comparison. *Proceedings of the USENIX Annual Technical Conference*, January 1995.
[14] Shankland S. Transmeta Taking Linux Gadgets Mobile. CNET News.com, http://news.com.com/2100-1001-254020.html?legacy=cnet, March 13, 2001.
[15] Strunk J, Goodson G, Scheinholtz M, Soules C, Ganger G. Self-Securing Storage: Protecting Data in Compromised Systems, *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October, 2000.
[16] Sweeney A, Doucette D, Hu W, Anderson C, Nishimoto M, Peck G. Scalability in the XFS File System. *Proceedings of the USENIX Annual Technical Conference*, January 1996.
[17] Torvalds L. Why is *ramfs* slower than *ext2fs*? *Personal communication*, September 2001.
[18] Wang AIA, Kuenning GH, Reiher P, Popek GJ. *Conquest*: Better Performance Through a Disk/Persistent-RAM Hybrid File System. *Proceedings of the USENIX Annual Technical Conference*, June 2002.

An-I A. Wang is a post-doctorate student of computer science at UCLA. He received his Ph.D. and M.S. in computer science from UCLA in 2003 and 1998, and his B.A. in computer science from UC Berkeley in 1995. His research interests include file systems, distributed ad hoc network topology construction, optimistic replication, and performance analysis.

Geoffrey H. Kuenning is an assistant professor of computer science at Harvey Mudd College. He received his Ph.D. in computer science from UCLA in 1997, and his B.S. and M.S. in computer science from Michigan State University in 1973 and 1974. From 1974 to 1989, he worked in the areas of operating systems and embedded systems. His research interests include file systems, performance analysis, and computer system security.

Peter Reiher is an adjunct associate professor of computer science at UCLA. He received his Ph.D. and his M.S. in computer science from UCLA in 1987 and 1984, respectively. He received his B.S. in electrical engineering from the University of Notre Dame in 1979. Dr. Reiher's research interests include active networks, advanced operating systems, parallel discrete event simulation, and security for distributed systems.

Gerald Popek is an adjunct professor of computer science at UCLA. He received his Ph.D. and his M.S. in applied mathematics from Harvard University in 1973 and 1970, respectively. He received his B.S. in nuclear engineering from New York University in 1968. His research interests include distributed file systems, mobile computing, and computer security. Dr. Popek is also chief technology officer for United Online, a large Internet service provider.