

HFS+: The Mac OS X File System

Laura LeGault

February 22, 2009

Abstract

The Macintosh OS X operating system is built to interface with the HFS+ file system. Assuming that allocations and disk I/O can be detected by monitoring the internal clock and noting any significant slowdown, we attempted to discover block size, any effects of prefetching, the file cache size, and the number of direct pointers in the HFS+ inode. Our tests were met with unfortunate amounts of nondeterminism, likely the result of optimizing algorithms within the operating system.

1 Introduction

The HFS+ file system was originally introduced with the Mac OS version 8.1 in 1998 [1]. The system allows for 32-bit addressable blocks, 255-character filenames employing Unicode, expandable file attributes, a catalog node size of 4 kilobytes, and a maximum file size of 2^{63} bytes [2]. These characteristics are largely improvements over the previous HFS file system, and have carried on to present usage.

What we are primarily concerned with in the present paper however is the structure of the file system itself. The most telling structure for this analysis is in most cases the inode, or as it is referred to in the Mac OS literature [2], the *indirect node file*; however the structure in HFS+ with which we shall concern ourselves primarily is called the `HFSPPlusForkData` structure. This is where HFS+ stores information about the contents of a file: a 64-bit size of the valid data present (enforcing the maximum file size), a *clump size* signifying a group of allocation blocks which are allocated to a file at one time to reduce fragmentation, the total number of allocation blocks used in a file, and an array of extent descriptors.

1.1 Extents

While the allocation block is the basic unit in which memory is allocated (in *clumps* of same, as mentioned above), what the HFS+ system is primarily concerned with are the extents of blocks. An `HFSPPlusExtentDescriptor` contains merely a starting block address and a length - a base/bounds approach to file allocation. Each file's catalog record contains eight

direct extent descriptors, and most files (regardless of size) do not use more than their eight extents [2]. If a file is located in a very fragmented portion of the disk, it is possible to force a file to use more than eight extents. This creates an *extents overflow file*, which is stored as a B-tree. However, according to experiments performed by Amit Singh and reported in [3] on the HFS+ file system using Singh's `hfsdebug` tool, more than 99% of files on Mac OS X systems are completely unfragmented - that is, more than 99% of all files have only *one* extent.

Due to the existence of extents, it is nearly impossible to measure the actual size of allocation blocks. Thus we rely upon the literature [2] and Mac OS X's `sysctl -a hw` command, which agree that for modern systems, block size is set at 4 kilobytes. However, we also note that the purpose of extent use is to reduce fragmentation: unlike the standard linux inode, with a number of direct pointers to single allocation blocks, an indirect pointer, and a doubly indirect pointer, extents point to contiguous sections of memory.

2 Methodology

The present experiments were conducted on a PowerBook G4 running Mac OS X version 10.4.11 with a single 867 MHz PowerPC G4 processor and 640 MB of SDRAM.

2.1 Timer

We first discovered a highly-sensitive timer for measuring our experiments. While `gettimeofday` provides measurements in microseconds, we found the `rdtsc()` method to be much more sensitive. Our first basic trial involved a comparison of `gettimeofday` to `rdtsc()` and discovered that `rdtsc()` is much more sensitive to the actual time taken.

```
static __inline__ unsigned long long
rdtsc(void)
{
    unsigned long long int result = 0;
    unsigned long int upper, lower, tmp;
    __asm__ volatile(
        "0:      \n"
        "\tmftbu  %0   \n"
        "\tmftb   %1   \n"
        "\tmftbu  %2   \n"
        "\tcmpw   %2,%0 \n"
        "\tjne    0b   \n"
        : "=r" (upper), "=r" (lower),
          "=r" (tmp)
        );
    result = upper;
    result = result<<32;
    result = result|lower;

    return result;
}
```

2.2 Block Size/Prefetching

Though we had determined already through literature search and querying the system itself that block size was allegedly 4096 bytes, we did set up an experiment to check block size and prefetching effects as an exercise to begin the project. On a standard Linux file system, this could be tried by reading blocks of various sizes and noting when the slowdown occurred; this information could then be processed to determine at what point more data would need to be loaded into memory; block size would then be some even divisor of the prefetching amount.

On a system with direct pointers in the inode rather than extents, it would be reasonable to assume that regardless of prefetching effects, after the direct pointers had been followed, there would be some slowdown when fetching subsequent blocks due to the increased number of pointers to follow in the system. Comparing the prefetching effects of the direct data

and the prefetching effects of data requiring an indirect pointer, one could uncover exactly how many times the indirect pointer table had been accessed, and from that determine the number of blocks being prefetched and thus by simple arithmetic, block size. (We note that this is entirely hypothetical as it wouldn't work on the HFS system and we have not tested this hypothesis.)

However, given the HFS+ use of extents, we did not have this advantage. We began by testing long sequential reads of 512, 1024, 2048 and 4096 bytes, and monitoring these reads for any slowdowns. We then moved to quasi-random reads, using `lseek()` to move ahead from the current position by an increasingly large offset.

2.3 Inode: Direct Pointers

On a file system using standard one-block pointers in its inodes, this test would be very straightforward: using the block size discovered in the previous test, sequentially add blocks until a slowdown is observed, at which point you have run out of direct pointers and into an allocation of the indirect block as well as the block for adding to the file. However with extents, the only way to test the number of direct extents kept within the file would be to artificially create external fragmentation within the system. While this could theoretically be accomplished by repeatedly inserting new blocks into the beginning of the file, we could not come up with a prepend design that did not effectively involve extending the file, copying existing blocks, and writing a new block at the beginning - which still would be subject to the apparent unofficial OS X anti-fragmentation policy.

Our test instead involved creating an 800 MB file and monitoring output for any significant slowdowns, in hopes that a large file size would uncover fragmentation already present in the system.

2.4 File Cache

The tests for file cache were relatively straightforward - simply repeatedly read increasingly large portions of files. Outside of the timer, we also included a function which printed a random element in the buffer, having been advised in our undergraduate OS course of a situation wherein a benchmark did nothing but intensive, timed calculations and produced no output, and the compiler very nicely optimized all of the calculations away.

We used the file created during the direct pointer test to examine the file cache, as we did not find any preexisting files on the system large enough to fill up the reported 640 MB of memory.

3 Results

Our results were unfortunately nondeterministic, outside of the initial timer testing. Multiple runs did not uncover any real patterns, though we do note that for particularly the file cache tests, the times for all runs regardless of read size improved by a factor of three to four when run immediately after another test.

3.1 Timer

The most simple test, we merely compared the results of two subsequent `gettimeofday` calls and two subsequent `rdtsc()` calls as detailed above. The calls to `gettimeofday` averaged .3 microseconds over 10 calls, while `rdtsc()` averaged 4 clock ticks per run. We observed odd behavior with the `rdtsc()` run, noting that every set of 10 runs began with an initially anomalously high time (≈ 3 standard deviations higher than the mean - see Figure 1). We note that this anomalously high initial timer interval carries over into other tests as well, though whether that is an artifact of this test or actually reflective of internal behavior is at this point unknown.

3.2 Block Size/Prefetching

While this particular test was already known to be relatively futile short of running `sysctl -a hw` and reading the output, we still uncovered some interesting data. Our test ran 1000 sequential reads of block sizes 512 bytes, 1 KB, 2 KB, and 4 KB, and with the exception of some outliers (the first test of the first run showed anomalously high time), all performed nearly identically, each with one outlying run (if we ignore the first run of the 512 byte set). This data is presented in Figure 2.

When examining the 4 KB block fetch, we look at sequences of 10 individual reads and their times. Unfortunately only one of our runs displayed a slowdown halfway through, which we must assume is an anomaly (see Figure 3). When running several thousand reads, we would occasionally (approximately 10% of the time) see series wherein no slowdown occurred at all after the initial slow load - all reads were at either 66 or 67 ticks for a thousand reads. The

other 90% of the time, we observed randomly placed slowdowns, which did not cluster around any specific points.

3.3 Inode: Direct Pointers

As mentioned earlier, due to the use of extents and their relationship to fragmentation, we determined it would be extremely difficult to produce sufficient fragmentation to allocate an extent overflow file. Regardless of the difficulty of causing this to happen, even if it were to happen, there would be no way to use timing information to determine the number of direct extent pointers used, since the pointers can point to variable-sized chunks of data within the file system. However, we did manage to produce two runs with significant slowdown so as to suggest that another file may have been allocated at that point (see Figure 4).

3.4 File Cache

Our file cache results were of particular interest. Recall that the system on which we were running these experiments has 640 MB of SDRAM installed; that means a maximum possible file cache of 640 MB. However, when testing the cache, we observed nearly equal times for repeated read sizes over 700 MB as for 4 KB! The repeated 750 MB reads and 4 KB reads were as follows:

4 KB	462	431	392	375	430
	383	381	375	376	1089
750 MB	475	405	399	399	389
	383	391	383	382	383

4 Conclusions

There are many conclusions to be drawn from this experiment, though unfortunately none of them are the conclusions we wished to draw. In the HFS+ file system, the allocation block size is 4 KB; each inode-analogue contains eight direct extent pointers and one pointer to an extent overflow file which contains a B-tree of subsequent extents; prefetching effects are assumed but undiscovered, as is the size of the file cache.

From our experiments on the Mac OS X file system, we conclude that HFS+ is unfortunately too far removed from the standard Unix/Linux file system to respond effectively to timing tests. Tests to determine the fundamentals of this file system would

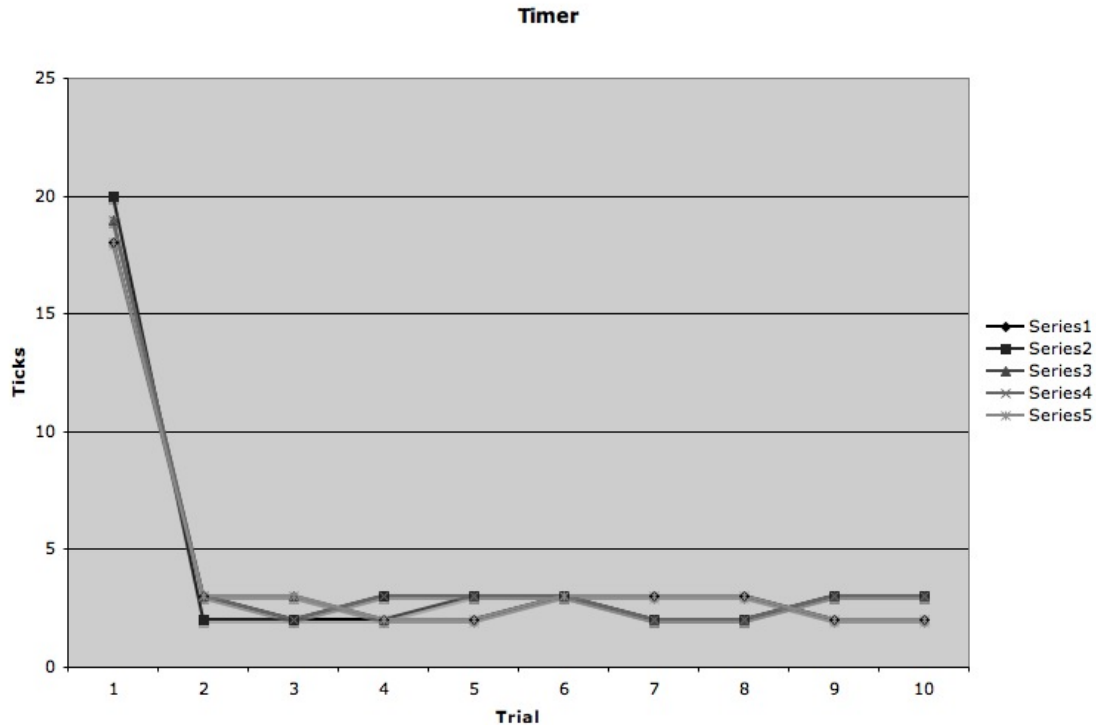


Figure 1: Five sets of ten runs of `rdtsc()`. In all sets we observe an initially slow start with consistent 2-3 tick runs following.

be best observed by a utility to monitor the HFS-PlusForkData structures directly and observe their changes - as they increase extent length, one could extrapolate block size; with precise information about extent locations one could forcibly introduce external fragmentation and induce further use of the extents, though direct access to the HFSPlusForkData structure would eliminate the need to test for the number of direct extents. A memory monitor might aid in judging the effects of cache size and prefetching be-

havior.

In short, we unfortunately conclude that the HFS+ file system is likely too complex for a timer-based experimentation project focusing on the structures inherent to the more classical file systems. We hypothesize that the apparently random slowdowns we observed were likely due to scheduler preemption more than behavior we were trying to observe, but with only a timer to use for our methodology there is no way of knowing for certain.

References

- [1] Technical note TN1121. Technical report, Apple Inc., <http://developer.apple.com/technotes/tn/tn1121.html>, 1998.
- [2] Technical note TN1150. Technical report, Apple Inc., <http://developer.apple.com/technotes/tn/tn1150.html>, 2004.
- [3] Amit Singh. *Mac OS X Internals*. <http://www.osxbook.com/book/bonus/chapter12/hfsdebug/fragmentation.html>, 2004.

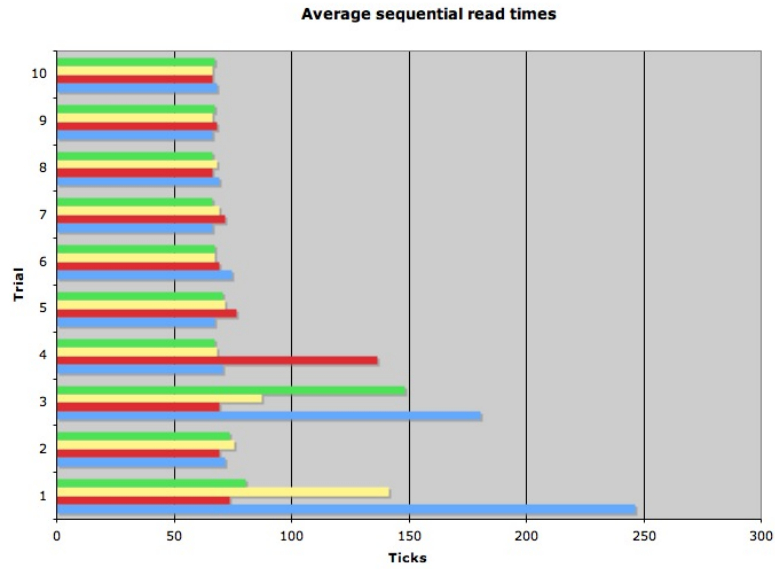


Figure 2: Average read times for sequential reads of blocks of size (1) 512 bytes, (2) 1 KB, (3) 2 KB, (4) 4KB

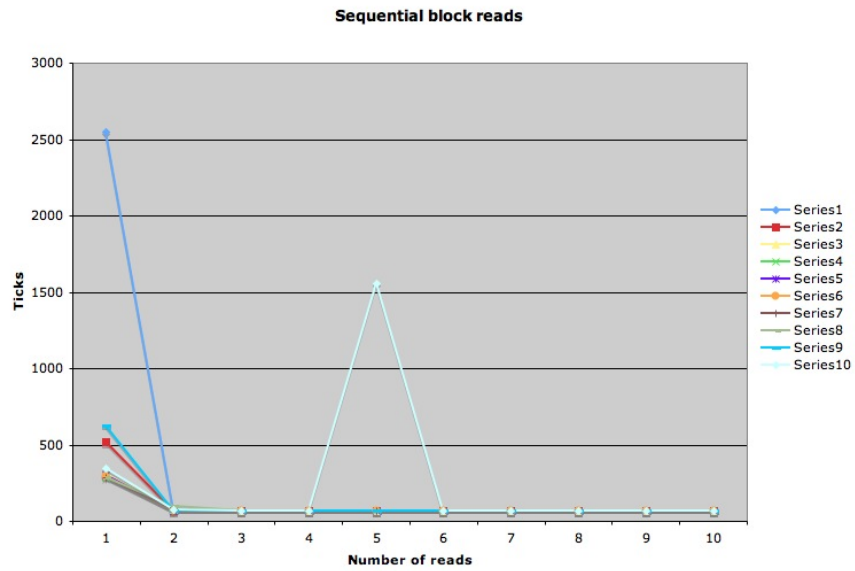


Figure 3: Sequential reads of size 4 KB on a large file. Observe the initial anomalously high time in the first run; likely an artifact of our timing device.

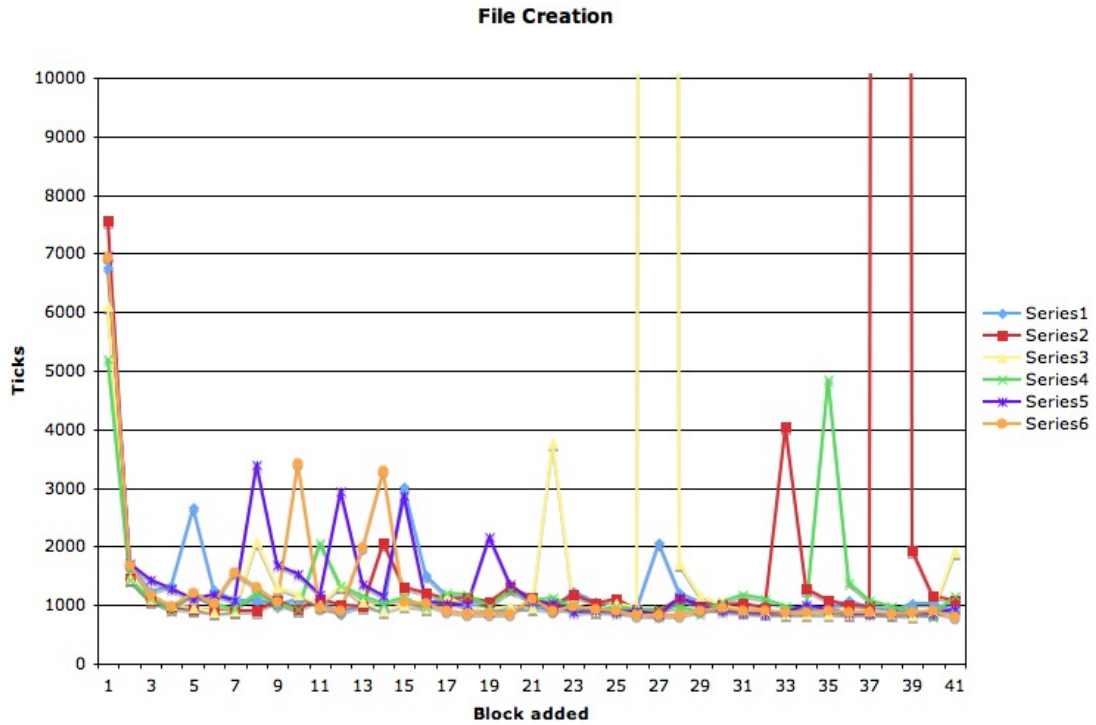


Figure 4: Time taken to append a 4 KB block to a file. Particular attention should be paid to the spikes in runs 2 and 3, which both reached over 100,000 ticks.

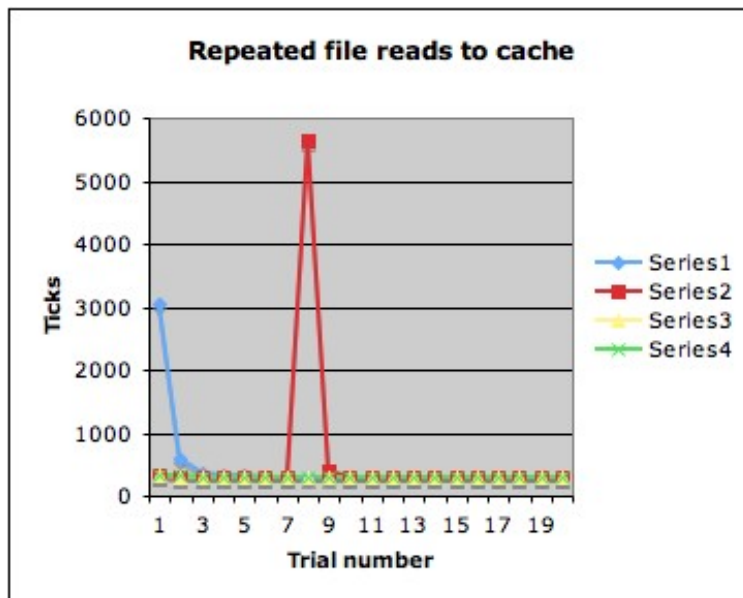


Figure 5: Times to read blocks of sizes (1) 512 MB, (2) 620 MB, (3) 640 MB, and (4) 750 MB. The spike at the beginning of (1) is likely an artifact of our timing device; we are unsure of the cause of the spike at trial 8 in (2).