Linux Memory Mapped System Call Performance

Kousha Najafi
Professor Eddie Kohler
Steve VanDeBogart

I. Introduction

Mmap and read are both fundamentally important system calls. Both calls are used to access bytes on disk. Read uses the standard file descriptor access to files while mmap transparently maps files to locations in the process's memory. Most operating systems also use mmap every time a program gets loaded into memory for execution. Even though it is important and often used, mmap can be slow and inconsistent in its timing.

A. Background

Mmap maps memory pages directly to bytes on disk. With mmap, whenever there is a pagefault, the kernel pulls the page directly from disk into the program's memory space. In contrast, with the pread system call, the kernel reads the file into kernel space, and then proceeds by copying the memory over to userspace. This extra memory copy should make pread slower than mmap.

Whenever there is a pagefault for a memory-mapped file, the hard drive seeks to the appropriate block and reads the data. Similarly, pread() atomically seeks and reads from disk, as opposed to using lseek() and read(). Because of this, both mmap and pread act similarly when reading files off disk, which makes pread a good comparison point to mmap.

B. Problem description

Preliminary testing showed various performance inconsistencies with mmap when compared to pread. Mmap was very inconstant with its timings; not only was mmap slower than pread, on successive runs of the same program, mmap's times varied 100% from the average.

Thus, the purpose of this research is to determine why mmap is slower than pread and why mmap timings are inconsistent.

II. Approach

The approach to figuring out the problem started with more in-depth testing. This testing showed that mmap could be faster than pread when reading files in reverse, but the timing inconsistencies made it slower on the average. This lead to instrumenting the linux kernel source.

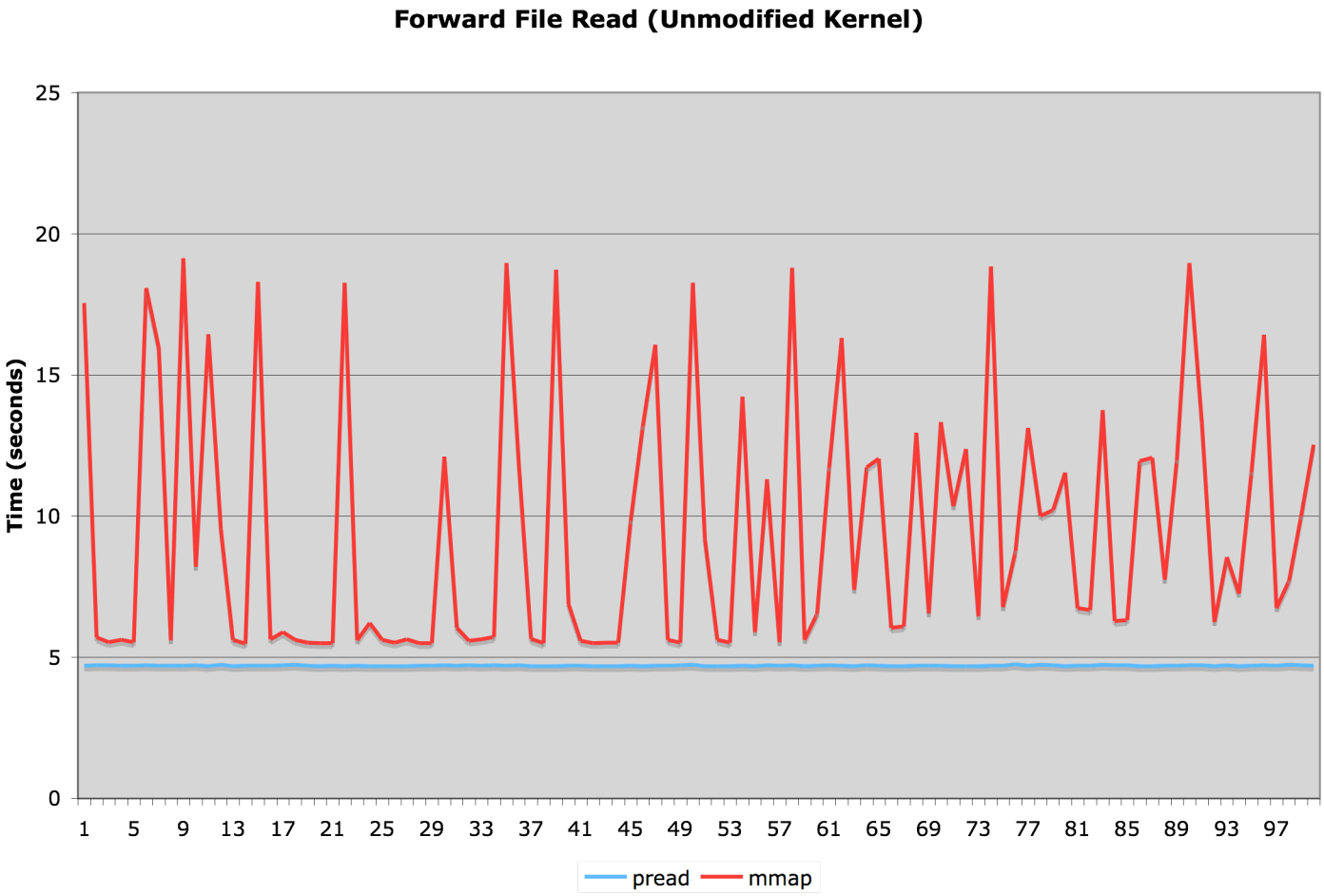
During various tests while intrumenting the linux source, the timing inconsistencies in mmap disappeared. This allowed mmap to be faster than pread on average when reading a file backwards. This happenstance was finally narrowed down to one print statement in a section of the mmap call that involved reading and readahead. Even though the timing inconsistencies were fixed, it was unknown the actual cause, and there was still the problem of the overall slowdown. Further instrumentation and testing lead to no more breakthroughs or interesting results.

The next plan of attack was thorough testing and simulation. The testing suit was expanded to include patterns of file reading. This included big file reading, both forwards and backwards, multiple small file reading, forwards and backwards, and “random” file reading with big and small files.

For simulation, the algorithms for pread and mmap were analyzed and simplified. Blocktraces were used to look at the different sequence of disk requests. This lead did not show much difference in the order of block requests, but it did show very interesting timing differences between the two system calls.

III. Experimental results

Preliminary testing showed both the inconsistency and slowdown of mmap versus pread. The following graph displays 100 consecutive runs of our test program. The program reads a 256MB file sequentially on a machine running with 128MB ram. The disk cache is flushed in between pread and mmap every iteration.

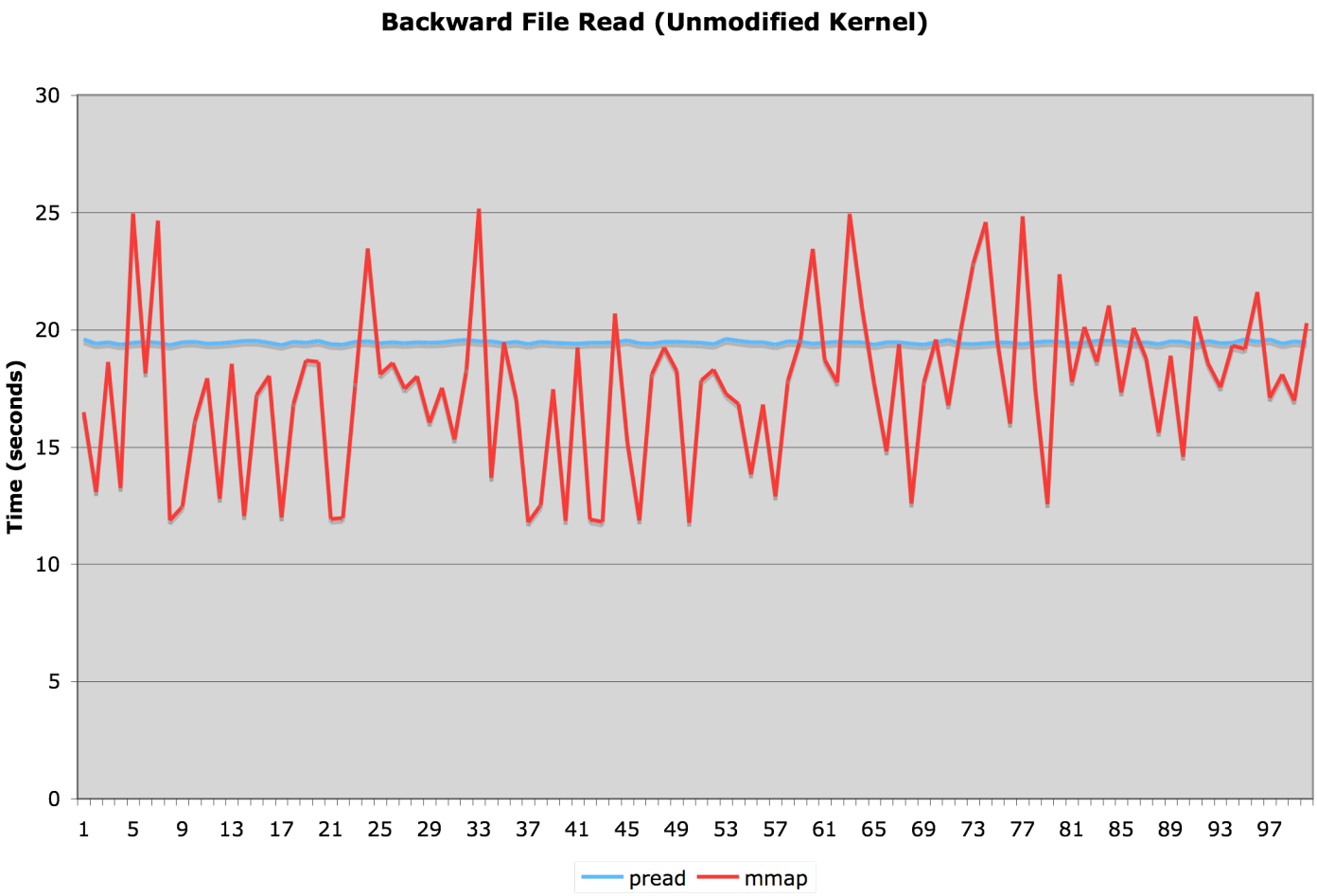


	pread	mmap
avg	4.69813978	9.50363338
min	4.67799	5.488
max	4.740224	19.120515
stdev	0.014100833	4.504765678

The straight blue line shows the consistent times of pread, while the red line shows the inconsistent timing of mmap. The consistency of pread is showed by its low standard deviation, about .3% of the average. In contrast, mmap is very inconsistent, with a standard deviation of a little over 47%. It is worth noting that the minimum of mmap,

5.488 seconds is comparable to the average time of pread, 4.698 seconds. This minimum speed of mmap is approached several times during the different iterations, showing the potential speed that mmap can achieve.

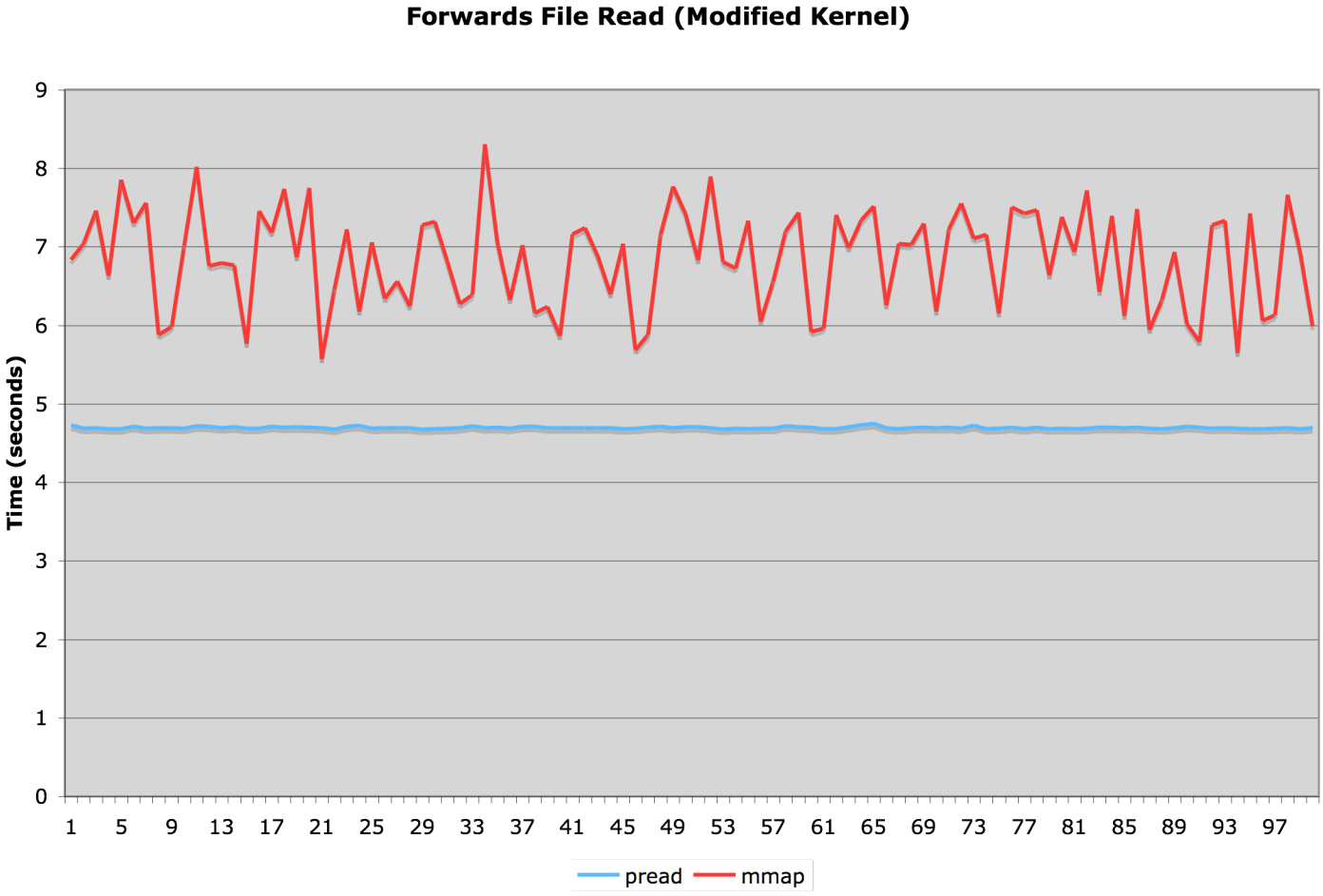
The next graph displays 100 consecutive iterations of reading a 256MB file backwards.



	pread	mmap
avg	19.4567397	17.51321265
min	19.337841	11.757977
max	19.59812	25.156444
stdev	0.057198775	3.451290738

This graph shows that mmap can be faster than pread in certain situations, but there is still highly fluctuating. The difference between the maximum and minimum time for mmap is a little over 200%. Mmap also has a slightly less standard deviation of about 20%. Surprisingly, the average time of mmap is actually faster than pread.

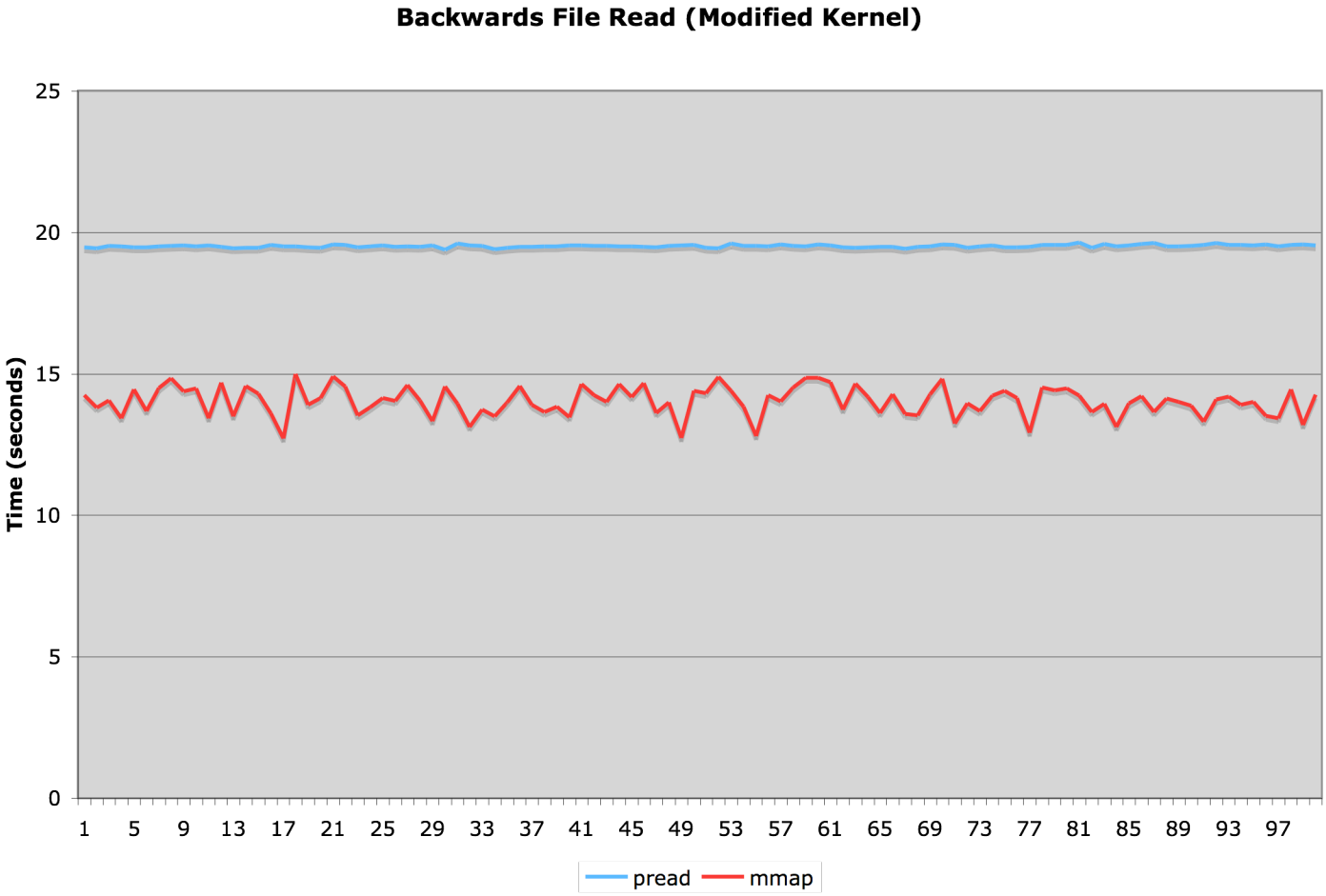
The following graph is similar to the first graph, the only difference is that we are running a modified kernel that slightly changes mmap's timing.



	pread	mmap
avg	4.69465562	6.84895584
min	4.676946	5.56658
max	4.749525	8.30888
stdev	0.013176367	0.634541817

The graph above shows a big difference from the first graph. Like before, pread is consistent with a small standard deviation of .3%. Even though here is still fluctuation in the timing of mmap, the standard deviation is only around 9%. In the unmodified kernel, the maximum time of mmap was over 300% greater than the minimum speed. On the modified kernel, the maximum is about 50% off the minimum. Also, the standard deviation is an order of magnitude less than the average time.

Similarly, the next graph shows a backwards file read on the modified kernel. All other parameters are the same.

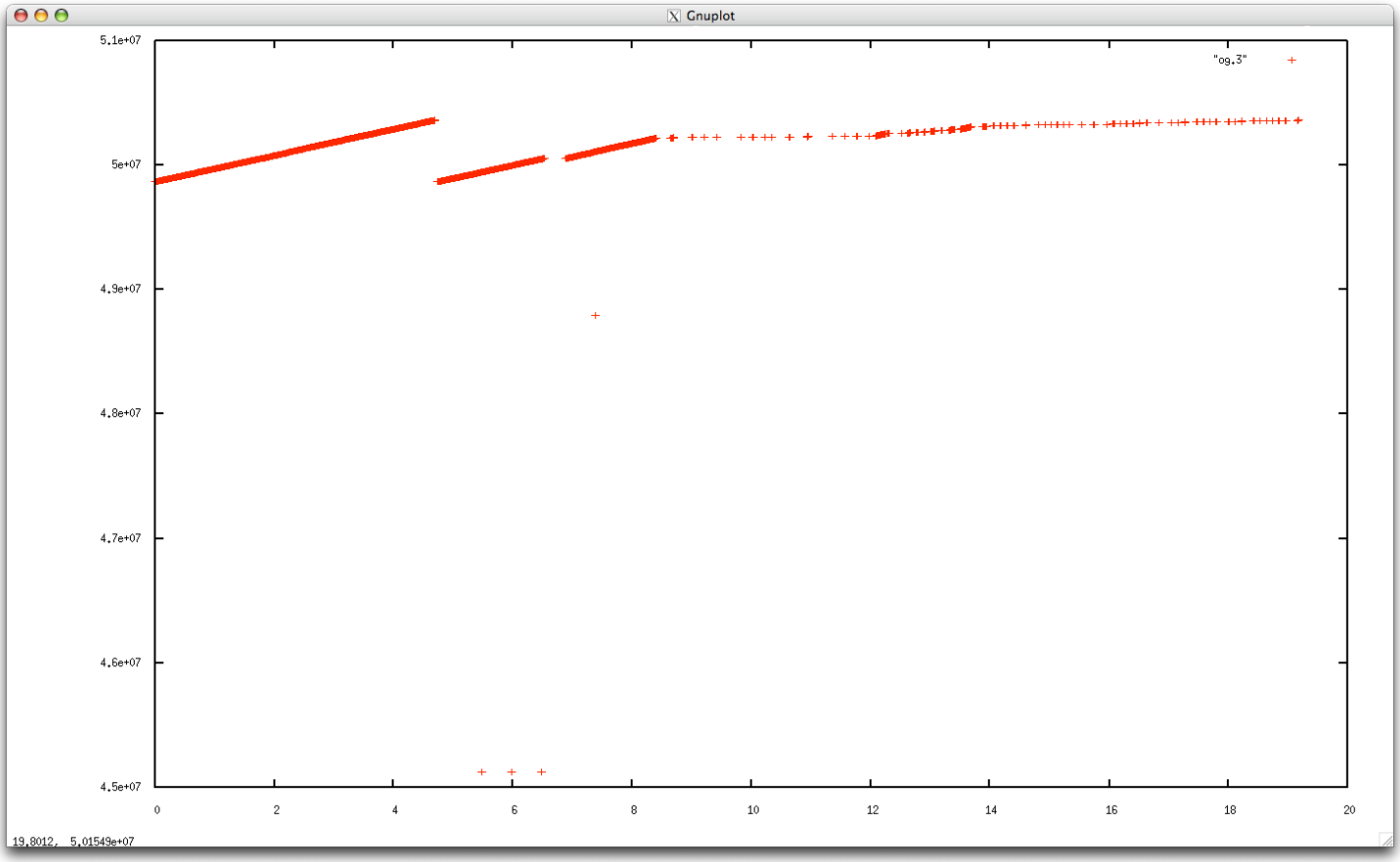


	pread	mmap
avg	19.51153098	14.03146308
min	19.379135	12.714086
max	19.645147	14.999039
stdev	0.048188733	0.518356679

This graph shows a remarkable change when compared to the first two graphs. As usual, pread has a consistent time throughout all iterations, with a standard deviation of only .25%. The interesting part about this graph is that mmap is consistently faster than pread. By reading the file backwards, the effects of readahead of canceled out. Therefore, any speed advantage that pread had because of better readahead mechanisms are ruled out. That mean mmap is able to pull the same file from disk faster than pread.

We modified the kernel by slowing down the mmap system call where it does its readahead calls. This slowdown helped the requests to disk come in at more consistent intervals. This probably allowed the disk buffer or the disk scheduler to more easily handle the repeated requests to disk.

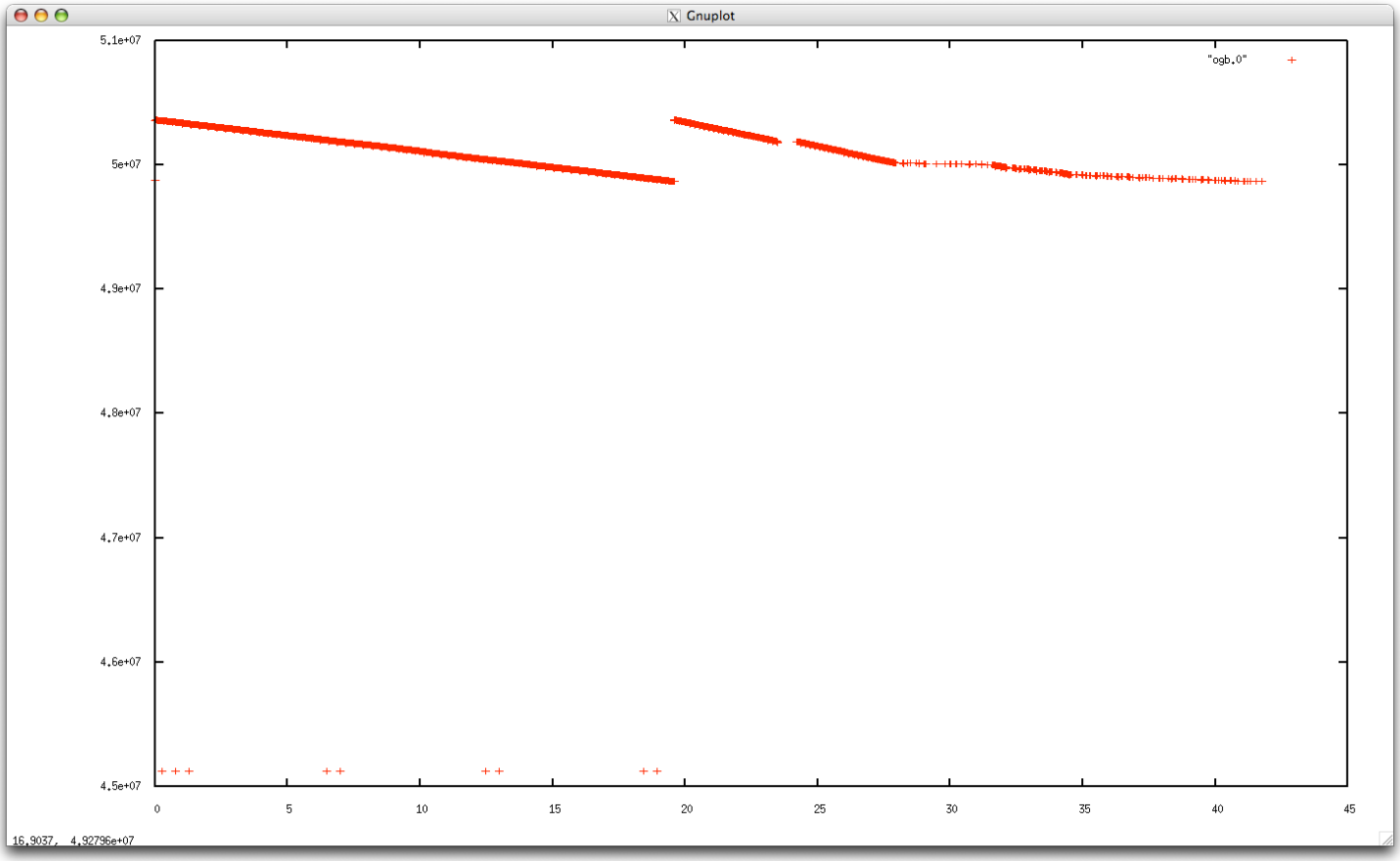
We also ran blocktraces on the same test program. The next graph shows the blocktrace from the first graph above. This trace if from a forward file read on the unmodified kernel.



It is immediately apparent where the slowdown and inconsistencies are. The first solid positive sloping line is when pread is accessing the file. The rest of the graph is when mmap is accessing the file. Two important things to note are the gap in the second line, and the trailing tail. The first gap in mmap is about half a second in length. The second gap right before tail is a quarter of a second.

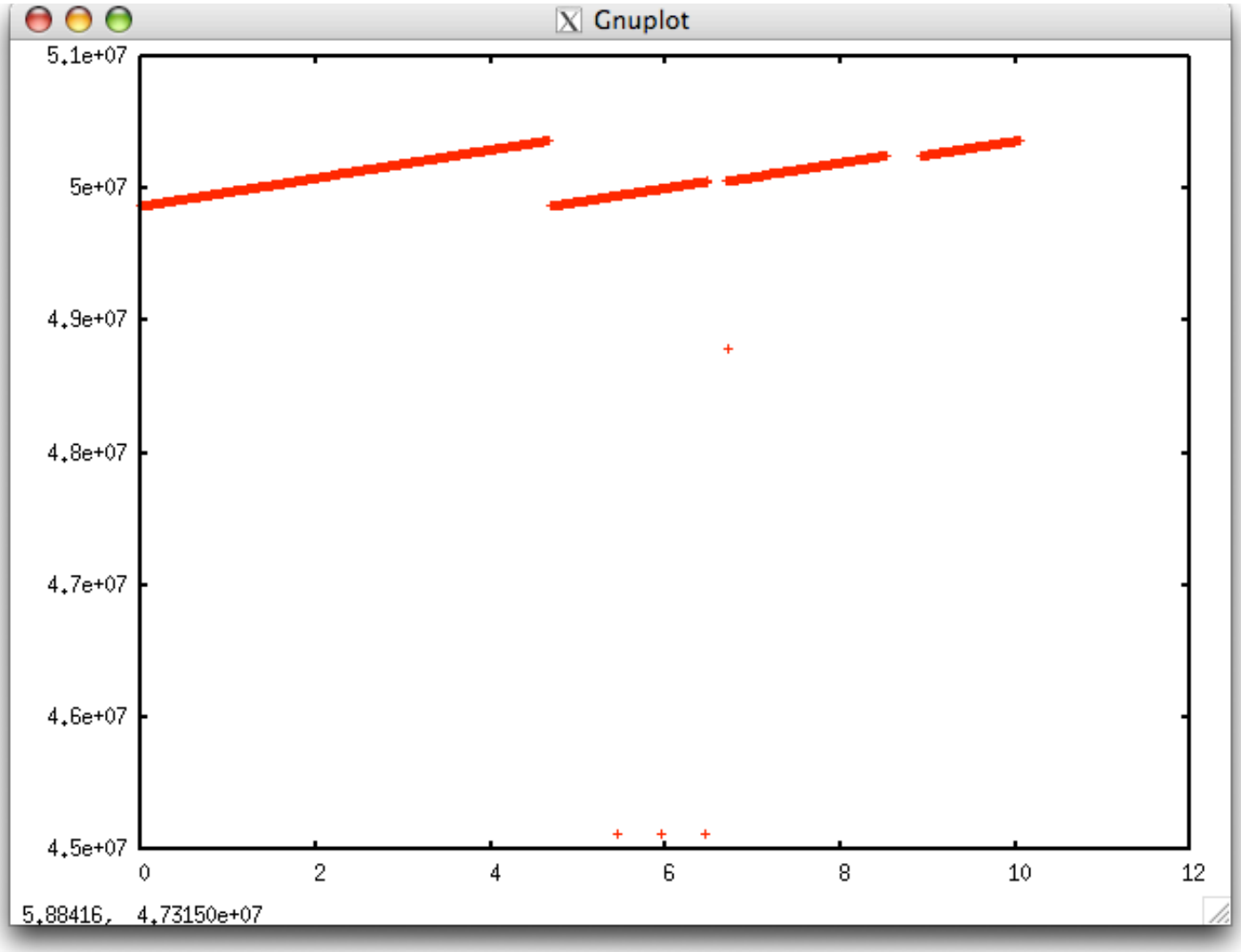
From various different block traces, it became apparent that the trailing tail of the mmap read was the cause for the inconsistencies in timing. All the block traces had similar formats, with a compact solid line for read, and a segmented line for mmap with a long trailing tail. Depending on the fluctuations of the test, the tail grew and shrank in proportion to the how inconsistent the run was.

Similarly, the next graph shows a block trace of a backwards file read on the unmodified kernel.



This graph is similar to the first block trace. It contains a solid negative sloping line for pread, while mmap has a two gaps and a trailing tail.

More interesting results come from the tests run on the modified kernel. The next graph shows a blocktrace of a forward file read running on the modified kernel.



On this graph, the first solid positive sloping line is for pread, while the segmented line is for mmap. The interesting part is the gaps that are during the mmap read. These gaps account for almost the exact time difference between pread and mmap. Depending on how far off the times where, the gaps in mmap would shrink or grow. Some times there would be more than two gaps, but there were usually just two. The backwards file read on the modified kernel was very similar.

From these graphs, there came two possible reasons for the gaps: either the process scheduler was sleeping the process, or the disk schedule was plugging this processes disk requests to allow other processes to have access to the disk.

We addressed the possibility of the disk scheduler allowing other processes to access the disk by using a separate disk for our tests. In this way, there would be no other processes trying to access the disk, so our test process would have unrestricted access. Even after we moved our tests to a separate disk, the gaps still remained.

To address the possibility of the process scheduler sleeping the process, we recoded our test program to have realtime scheduling priority. This would allow our test program to run unimpeded by the OS's process scheduler. Yet, even with realtime scheduling, out block traces still looked the same.

IV. Conclusion

Overall, further tests are necessary to figure out the true cause of the slowdown. By changing the timing on the mmap system call, we were able to reduce the inconsistencies in runtime of mmap. By reducing the inconsistencies, we were able to see distinct gaps that were the cause of the slowdown. Running the process with realtime scheduling priority eliminated the process scheduler as the culprit for the random stops. What probably remains is some sort of bug or optimization in the disk io scheduler that effects mmap's reading of the disk. Another possibility could be the way the memory subsystem handles various repeated pagefaults.

V. Acknowledgements

Thanks you Prof. Eddie Kohler for teaching me the skill necessary to start tackling interesting systems problems. Thanks to Steve VanDeBogart for his help setting up tests and his various help throughout. Thank you Mike Mammarella for your random quick help. Finally, thank you Prof. Amit Sahai for his input and for making this research program possible.