Marc Beitchman
CSEP 551
Assignment #1
1/29/2012

For this project, I used the Fedora virtual machine that was supplied as part of the assignment. This VM contains kernel version 2.6.38.2. I hosted the VM in VMWare Fusion on a MacBook Pro with an Intel core I7 running Max OS X 10.7.2. I discussed aspects of the project with fellow students, Brad Burkett and Jamie Miyazaki.

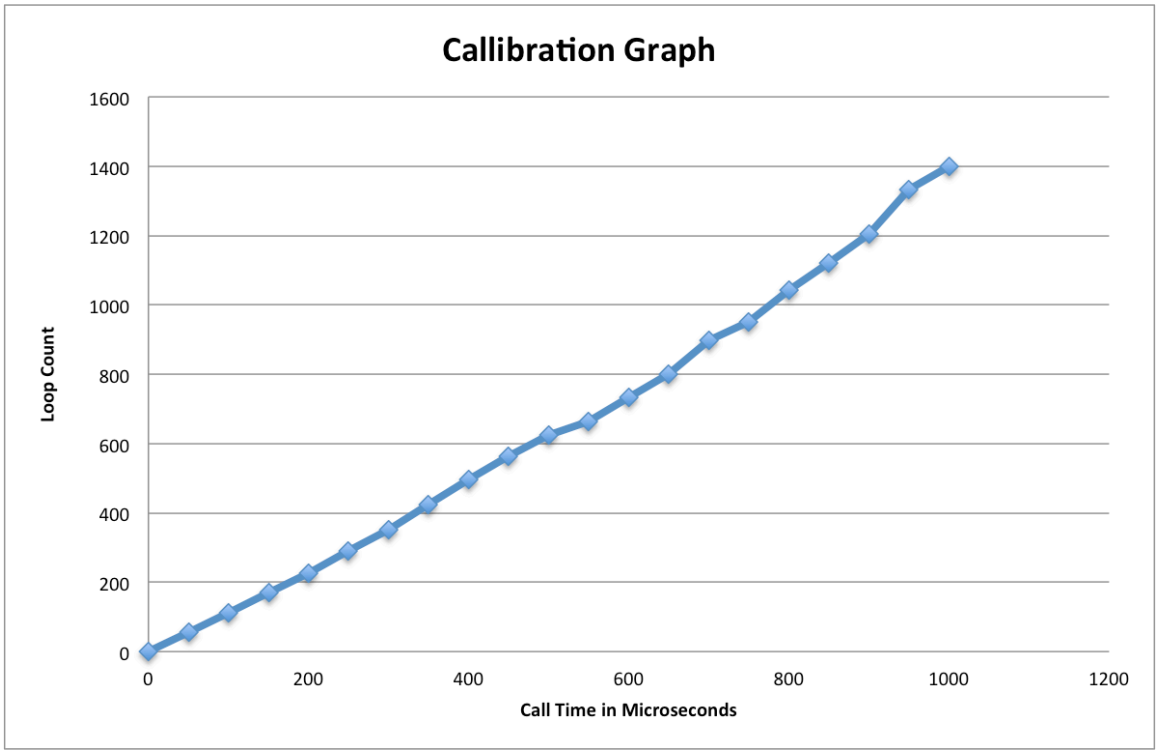
All raw data from my benchmarks is available in the data folder. All code is available in the code folder.

In order to add delay to the kernel, I found the two places in entry_32.S under (/home/451user/project1/kernel/linux-2.6.38.2/arch/x86) where the system call actually happens and inserted a call to a function that I defined in a new c file that I added to the kernel (/home/451user/project1/kernel/linux-2.6.38.2/mydelay.c) just before the call to the system call. I called my function using the *call* instruction and used the *SAVE_ALL* macro before my call and the *RESTORE_REGS* macro after my call to save and restore the registers. The function I defined in mydelay.c returns without doing any work. I exported a function pointer from this file using *EXPORT_SYMBOL* that initially points to the empty function. My kernel module contains my actual delay function. When the kernel module is installed (*init_module* is called), I assign the exported function pointer to a function (*internal_delay)* defined in my module that has the implementation of the delay loop. When the module is removed (*cleanup_module* is called), the exported function pointer is set back to the function that is defined in mydelay.c that does not do any work.

I implemented a kernel module (delay_module.ko) that stores the delay value. This value can be modified by user-level code (by using 'echo') to change the amount of loops that will occur in the delay.

In order to generate the calibration value, I wrote a c-program (perf.c) which calls *close(100)* for ten-thousand iterations. The calibration generation sequence is implemented in the function, *RunCloseMeasurement()* in perf.c. I only call *gettimeofday()* once before and once after the iterations in order to amortize the cost of the delay that occurs when *gettimeofday()* is called.

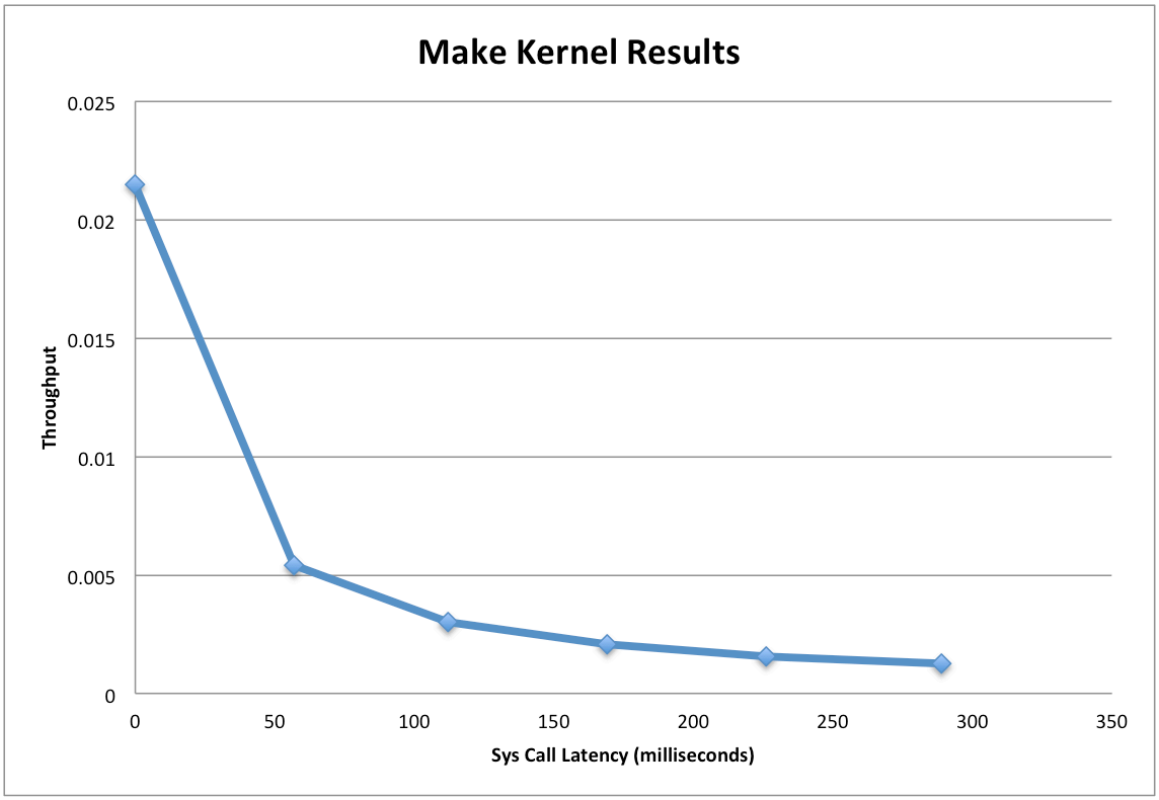
The calibration graph is as follows:



In order to generate the benchmark for compiling the Linux kernel, I added a function (*RunMakeKernelMeasurement()*) in perf.c to measure kernel build time.

I measured in loop increments of 50 from 0 to 250 loops. I ran three timed iterations of each kernel build and averaged the results which report in micrososecnds how long the kernel build took for a given loop count. I ran make clean off the clock and between each build. I also flushed the file system buffer cache between each build. After running this test, I saw build time increase with loop count, which indicated my delay was working as expected.

In order to generate the throughput graph, I converted the build time from microseconds to seconds and then took the inverse. I also replaced loop count with system call latency as calculated in my calibration sequence. The graph is as follows:



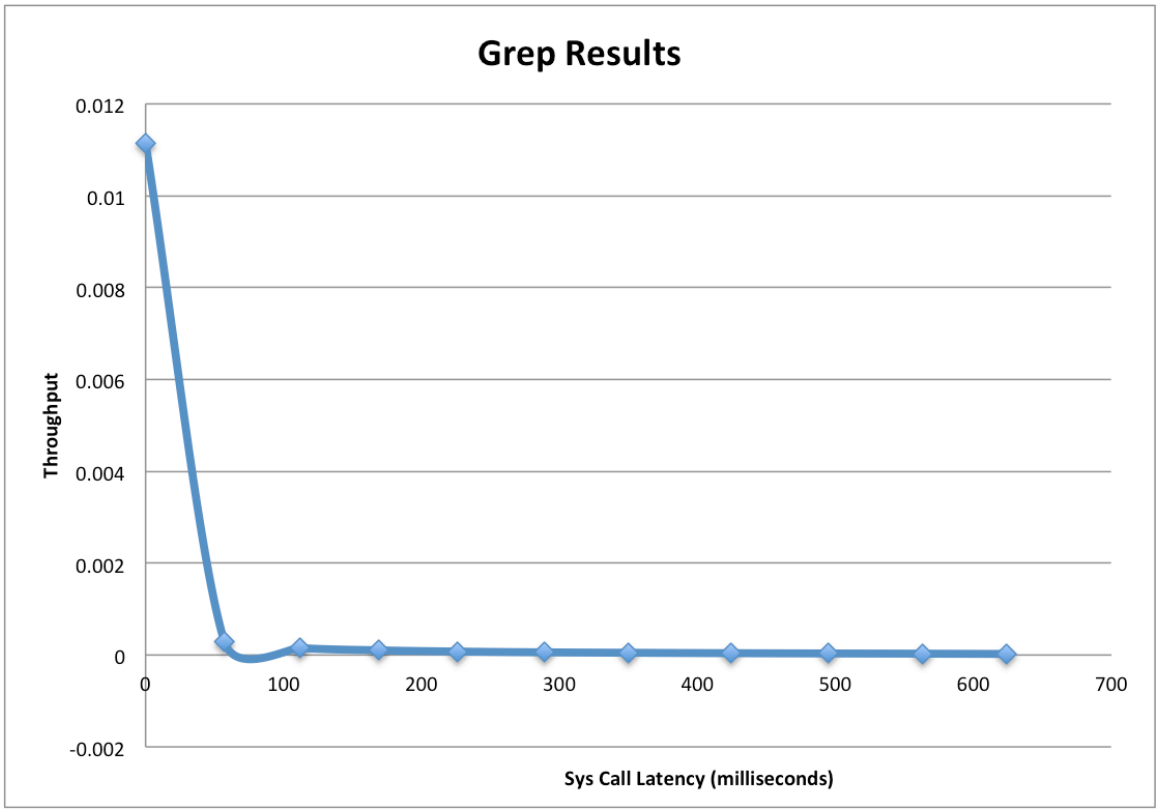
This graph clearly shows that throughput of building the kernel is dramatically influenced by system call latency. I assume this is due to building using the file system copiously. The sharpest decline in throughput comes when latency is initially added to the system call path. As the latency increases, throughput continues to decline but declines by a lesser amount. This graph shows the importance of making system-calls as low-latency and optimized as possible since latency dramatically impacts performance.

For the measurement of my choice, I chose to measure a recursive grep command on the Linux source tree that searches for the system call instruction. I added a function (*RunGrepMeasurements()*) in perf.c to automate taking measurements for this benchmark.

I measured in loop increments of 50 from 0 loops to 500 loops. I ran five timed iterations of each grep execution and averaged the results to report in micrososecnds how long the grep took for a given loop count. I also flushed the file system buffer cache between each group of iterations. After running this test, I saw

execution time increase with loop count, which indicated my delay worked as expected.

In order to generate the throughput graph, I converted the execution time from microseconds to milliseconds and then took the inverse of the execution time. I also replaced loop count with system call latency as calculated in my calibration sequence. The graph is as follows:

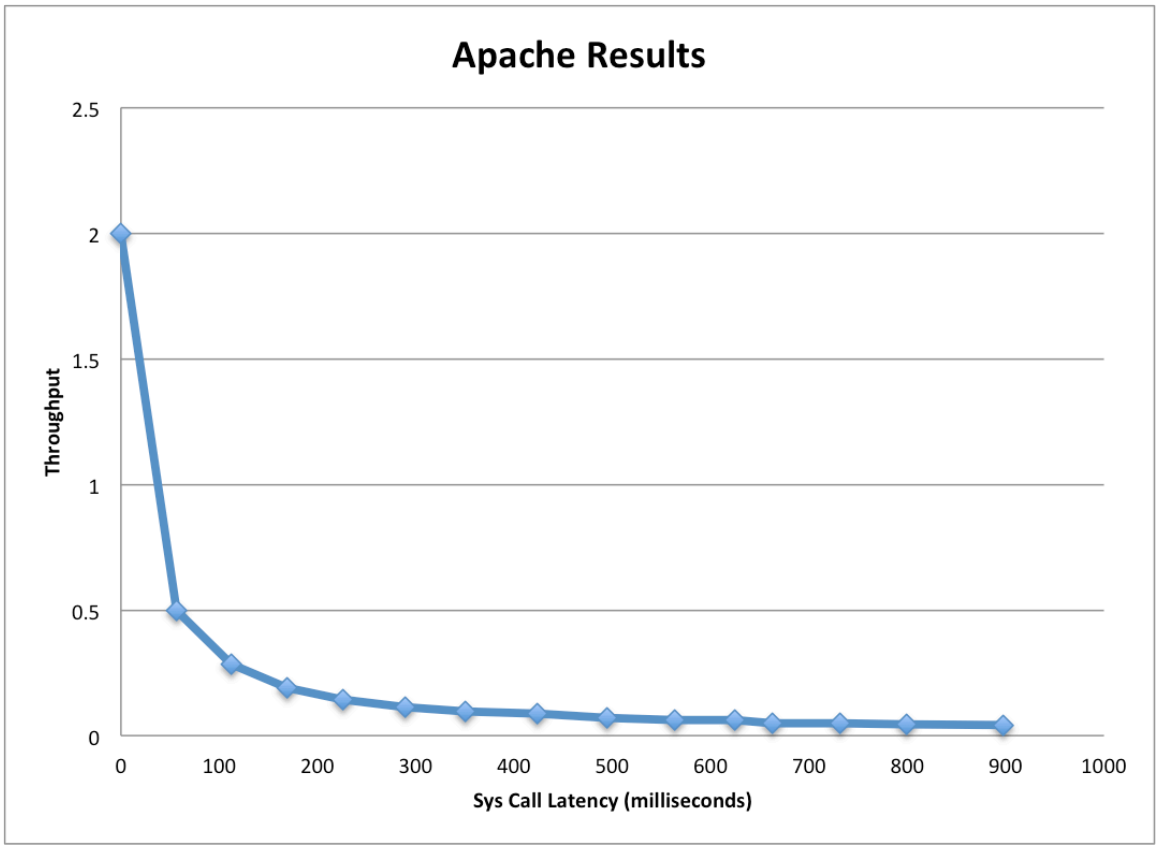


This graph shows that latency has a drastic impact on grep. After approximately 50 loop counts, the throughput reaches a minimum. It seems from this graph that grep does most of its work via system calls. Perhaps, generating more data points at smaller loop intervals would have given a more detailed throughput graph. However, the overall behavior is evident from the above results.

For the maximum throughput of Apache serving a small file, I started Apache on the VM and allowed port 80 through the firewall on the VM. I used the default website that Apache serves as the small static file. On my host, I installed httperf and ran 10 connections at a rate of 1 per second to the Apache server hosted on the VM. I increased loop count on the VM from 0 to 700 in increments of 50. I manually ran httperf from the host and manually adjusted the delay count on the VM for this measurement.

In order to determine throughput, I tracked the reply time reported by httperf, which measures in milliseconds how long it took the server to respond and how long it took to receive the reply.

I took the inverse of the response time to generate throughput. I also replaced loop count with system call latency as calculated in my calibration sequence. The graph is as follows:



Again, similar to building the kernel, there is initially a sharp decline in throughput that eventually levels out. From this data, it also appears that system call latency has a major effect on Apache's ability to respond to requests. It is evident of the importance to design low latency systems and to write applications that minimize the amount of time they need to enter and exit the kernel.