Performance Measurement of Linux Kernel

Abstract— **The kernel performance of an operating system has a significant impact on the overall performance of a computer system. Kernel performance measurement can give us a unique insight into an operating system, provide us useful information in determining the bottlenecks and weaknesses in the system design, and thus enable us to provide alternatives. Understanding the implications of kernel performance data can enable us to write more efficient code for a given operating system by exploiting faster system level methods. Measuring key kernel features (e.g. context switch time) however, is a nontrivial task. Appropriate methodologies are needed to ensure meaningful measurements. This paper discusses the measurement of various important kernel features closely related to kernel performance, which include clock precision, kernel call overhead, context switch time, page allocation latency and file read time. Methods aimed at improving measurement accuracy are discussed. We have done experiments on two computers, both running Linux. Doing experiment on different computers with different hardware configurations is aimed at understanding how hardware differences affect kernel performance. Experiment results show that for most of the kernel features measured, the speedups are roughly proportional to the increase in hardware speed. Even though this paper is not intended to provide a comprehensive kernel benchmarking suite, the methodologies presented here can be extended to build such programs.**

I. Introduction

Performance measurement has always been an exciting research area in computer science. The data obtained from the measurements can not only be used to verify system designs, but to identify the bottleneck that limits the overall performance, and possibly suggest better alternatives as well. Measuring the kernel performance of an operating system can help us better understand the behavior of the system, which in turn can help us avoid writing code with slower system methods when faster versions are available.

The challenges in measuring kernel performance lie in the fact that that the operating system is a very complex system. Cautions must be taken in deriving the appropriate methodologies.

The easiest and most common way to measure the performance of an operation is to measure its execution. This seemingly easy task might turn out to be rather convoluted. Without the right methodology, experiment results might be meaningless.

Before jumping into the rest of the paper, let us consider the following scenario:

Suppose we are given a subroutine that(in the rest of the paper, subroutine and method are used interchangeably) reads system times with sufficiently high accuracy. Without loss of generosity, call this subroutine `gettime`, the returning value of which is the current system time. The goal of the experiment is to determine the execution time[1] of an operation[2]. To reduce the variation of the execution time measurement, the operation is carried out for sufficiently many times. An intuitive approach is shown in Fig. 1.

```
start_time = gettime();
for (i = 0; i < NUM_LOOPS; i++) {
  operation(); //operation to be timed
}
end_time = gettime();
average_time = (end_time - start_time) / NUM_LOOPS;
```

Fig. 1. An intuitive method for measuring execution time

The advantage of this method obvious, we do not need a `gettime` method with high resolution to get reasonably accurate result. With some observations however, we may also find out this method's drawbacks, which are summed up as follows:

- If the execution time of the operation is comparable to that of a single loop, the average time calculated will have a large positive error. An easy solution to correct this problem is to time the loops separately to obtain the loop overhead and compensate the measured time accordingly.
- A more severe problem with this approach is that if the program is executed on a multiprogramming system, the `for` loop might be interrupted half way through, since the operating system might decide to schedule another process. This problem is much trickier than the previous one since it is hard to know how long the loop has been interrupted. If the `for` loop takes a rather long time (e.g. one second) to finish, the proposed method will incur an unpredictable positive error since the process might have been switched on and off for many times. Trying to minimize the numbers of running processes can certainly alleviate the problem but not solve it, since even if it is the only user process running, it could be interrupted by many background system processes. Particularly, I/O operations due to paging might occur virtually anytime.

These observations lead to a revised method shown in Fig. 2. This method, with a high precision time routine, can provide a much higher timing accuracy.

```
for (i = 0; i < NUM_LOOPS; i++) {
  start_time = gettime();
  operation(); //operation to be timed
  end_time = gettime();
  t[i] = end_time - start_time;
}
average_time = average(t , NUM_LOOPS) - gettime_overhead;
```

Fig. 2. The revised version for measuring execution time.

This revised method tackles the first problem in the earlier version by moving the instrumentation calls into the loop. Since an array is used to record the execution time of the operation in each run, the excess execution time due to scheduling can be easily detected. Operating systems almost always schedule time slices much larger than one millisecond, the `average` subroutine can incorporate

[1] Time in this paper should be understood as real time or wall time without special notification.

[2] In this paper, **operation** should be understood as an abstract method. It could be anything (e.g. a system call) needs to be timed

this knowledge and do an average based on a partial sum, discarding elements too large for the operation's execution time. Because the operation's execution time is small, the overhead of the instrumentation calls (e.g. `gettime`) must be taken into consideration. This method also requires `gettime` to have a sufficiently high resolution since the execution times for most of the kernel subroutines are small.

In this paper, the measurement method in Fig. 2 is used whenever applicable.

The next section describes the hardware and software configurations of the two computers on which the experiments were carried out. Section III gives details of how the clock precision, kernel call overhead, context switch time, page allocation latency and file read speed are measured. Experiment data for both system configurations are given for all these measurements. Section IV concludes.

II. System Configuration

Each experiment is done on two different machines both running Linux in order to study how hardware configurations affect the measured kernel features. Table I and Table II show some of the key features of these two machines and the software environments.

TABLE I
Machine 1 configuration

Type	Desktop (Dell Dimension 4100)
CPU	Pentium III 1000 MHz
Bus Speed	133 MHz (FSB)
Phy. Memory	512 Mb (SDRAM)
Hard drive	ATA-66, 5400 rpm
OS	RedHat Linux 7.0
Kernel Ver.	2.2.16-22
gcc Ver.	2.96

TABLE II
Machine 2 configuration

Type	Laptop (Winbook 470E)
CPU	Pentium MMX 175 MHz
Bus Speed	25 MHz
Phy. Memory	128 Mb (EDO)
Hard drive	ATA-33, 4200 rpm
OS	RedHat Linux 6.0
Kernel Ver.	2.2.5-15
gcc Ver.	2.96

All `gcc` flags are turned off (i.e. no debug information, no optimization) for the compiled test programs to ensure predictable program behaviors. It would be ideal if the two machines are running the same version of kernel. Unfortunately, a few attempts trying to install a new kernel on the laptop proved to be unsuccessful. Since modifications to the kernels are usually aimed at adding new features, it is highly unlikely that there are significant differences in the implementations of these two kernels that affect the features we are interested in.

All unnecessary services (e.g. networking, X-server) are turned off to minimize the interference from other processes. X-Window is shutdown during tests. Test programs are run directly from terminals instead.

III. Experiments and Results

The following five subsections discuss the measurements of some kernel features in detail. In each subsection, methodologies are discussed first and then experiment results are discussed.

A. *Clock Precision Measurement*

As discussed in Section I, high accuracy real time measurement is crucial in measuring the execution times of most of the kernel level subroutines. Two methods described in the following two subsections can measure real time with resolutions up to microseconds and nanoseconds respectively[3].

A.1 Clock Precision Measurement Using **gettimeofday**

One way of measuring real time is to use the Linux library call `gettimeofday`. This method has a theoretical maximum resolution of one microsecond.

To determine the actual time resolution on a given system, a method similar to the one in Fig. 2 is used. The following code segment shows how time stamp data are collected.

```
struct timeval t[NUM_LOOPS];
for (i = 0; i < NUM_LOOPS; i++)
  gettimeofday(&t[i] , NULL);
```

Fig. 3. Code segment showing how time data is collected using **gettimeofday**

By walking through `t[]`, maximum clock resolution can be determined (it is the smallest nonzero difference between consecutive array elements). The average overhead for calling `gettimeofday` can also be calculated using array `t[]` by averaging all the nonzero differences. Table III shows the experiment data on both machines.

TABLE III
The resolution and overhead of **gettimeofday**

Machine	1	2
Max. Resolution (μs)	1	1
Average overhead (μs)	0.46	1.59

Theoretical lower bounds for time resolutions are achieved on both machines. The average overhead of the call is affected by CPU speed. On the slower laptop computer, the overhead is much higher than the maximum resolution. To measure kernel features using the method shown in Fig. 2, `gettimeofday` is not suitable to use since the overhead of the call is comparable to the execution time of many kernel operations. A better method is needed to provide more accurate time measurements.

[3]The actual accuracy depends on the CPU clock cycle.

A.2 Clock Precision Measurement Using **rdtsc**

Most of the modern processors provide instructions to measure elapsed machine cycles accurately. Starting from Pentium, Intel processors are equipped with a time-stamp counter. Intel's time-stamp counter is a 64-bit MSR (Model Specific Register) that is incremented every clock cycle. The `rdtsc` instruction (ReaD Time-Stamp Counter)can be used to access the time-stamp counter and provide accurate real time readings. The returned result is stored in `EDX:EAX`.

Since `rdtsc` returns clock cycles rather than time, CPU speed must be known accurately to obtain precise time measurement. CPU speed can be obtained via BIOS information or through experiment. One way to determine CPU speed through experiment is to time a high-precision sleep function (e.g. `nanosleep`), and use the elapsed cycles and the sleep time to calculate the CPU speed. This approach is used in our experiment to determine CPU speed. Verifying the calculated CPU speed against the BIOS reading , we conclude that this approach is very accurate. To minimize instrumentation overhead, inline assembly code is used to perform the `rdtsc` call in our experiments.

The maximum clock resolution is determined by placing two consecutive `rdtsc` calls within a loop, using a method similar to that shown in Fig. 2.

To use `rdtsc` correctly and efficiently, several precautions must be taken. The lower 32-bit of the time-stamp counter (in `EAX`) is periodically reset since it only takes only a few seconds (4 seconds on the 1GHz Pentium computer) to count through 2^{32} bits. Extra code to detect counter overflow is needed to provide correct measurements. Another factor that might affect the accuracy of time measurement using `rdtsc` is the out-of-order execution on modern super scalar computers. Most Pentium processors (After Pentium Pro) support out-of-order execution, in order to measure elapsed time correctly, instructions before `rdtsc` should be completed before the execution of `rdtsc`. Intel suggested calling `cpuid` at least three times before the first `rdtsc` to force all previous instructions to complete.

The first row in Table IV shows the time resolution obtained using the `rdtsc` instruction. Ten thousand measurements are taken and the smallest duration is chosen as the maximum clock resolution. The measured maximum resolution time is much higher than one clock duration. On machine 1 (desktop, Pentium 1GHz), the resolution is 34 clock cycle. This latency is resulted from the fact that instructions are needed to save the registers' contents (`EDX,EAX`), which requires many clock cycles to complete.

Since most of the kernel operations' execution time is well within one second, it is possible to only use the lower 32-bit of the time-stamp counter. A few clock cycles can be saved by only saving `EAX`. The result for this tweaked version is shown in the second row in Table IV. The corresponding assembly code for measuring clock precision using the improved method is small enough to be shown in Fig. 4. Code to correct the counter overflow effect is not shown figure.

The clock resolution on the faster desktop is only slightly better than that on the much slower laptop since Pentium III has a much longer pipeline and the instructions used to measure clock resolution can not fully exploit the parallelism of the architecture and thus require many more clock cycles to complete (29 cycles for machine 1 compared to 14 cycles for machine 2).

TABLE IV
TIME RESOLUTION OF **rdtsc**

Machine	1	2
Max. Resolution (μs)	0.034	0.086
Max. Resolution (μs, tweaked method)	0.029	0.078

```
xor ecx,ecx   # ecx=0
cpuid
cpuid
cpuid
rdtsc
add ecx,eax   # save eax to ecx
              # add is slightly faster than mov
rdtsc
mov start_cycle,ecx
mov end_cycle,eax
```

Fig. 4. The more accurate version of the assembly code for measuring clock precision

For the rest of the experiments, the more accurate `rdtsc` method is used to measure times well below four seconds and the standard `rdtsc` method (using both `EDX` and `EAX`) is used to measure longer durations.

B. Simple Kernel Call Overhead

Kernel calls are invoked frequently from user processes. How efficiently kernel calls are handled affects the overall system performance. The execution times for three simple kernel calls `getpid`, `getuid` and `write` (write one byte to the `null` device: `/dev/null`) are measured in our experiment.

Simple observations show that all the three simple kernel calls are cached when invoked multiple times. To measure cached and uncached execution time separately, two strategies are used:

- To measure cached kernel call performance, each call is invoked a couple of times to warm up the cache before the measurement starts. The rest of the method is similar to the one shown in Fig. 2. Each kernel call is timed ten thousand times and then the execution times are summed and averaged[4].
- To measure uncached kernel call performance, each call is only invoked once in the program (as opposed to put the kernel call within a loop). In order to provide sufficiently large number of measurements, a shell script is used to execute the program multiple times (1000 in our experiments) and average the results. Since each time the program is executed from a shell script, disk I/O activities might affect the accuracy of the time measurement. To avoid this problem, the program is paused for one second (using `sleep`)

[4]As discussed in Section I, it is possible to detect the effect of a context switch. Even without other user processes, occasional context switches can occur due to disk activities,etc. To ensure the accuracy of measurement, we have implemented code to detect such disruptions. Affected execution time data are not used for averaging.

before the kernel call is performed.

Table V shows both the cached and uncached performance measurements for the three system calls.

TABLE V
KERNEL CALL PERFORMANCE

Machine	1	2
`getpid` (μs) (cached,uncached)	0.29,1.69	0.78,14.81
`getuid` (μs) (cached,uncached)	0.32,4.67	0.79,14.95
`write` (μs) (cached,uncached)	0.35,2.93	1.23,22.01

On machine one, the cached kernel calls are six to thirteen times faster than the uncached calls. The speedups due to caching is even more prominent on the slower laptop: the cached calls are almost twenty times faster than the uncached ones due to slower CPU speed.

C. Context Switch Time

One of the easiest ways to measure context switch time is to use one of the interprocess communication (IPC) mechanisms the operating system provides. To measure context switch time accurately, the communication overheads must also be measured. We use three different methods–signals, pipes and shared memory–to measure the context switch time under Linux. Experiment results show that communication using signals has the highest overhead among all three methods. The context switch time measured, after taken into consideration of the communication overheads, is rather consistent.

- **Measuring context switch time using signals:** In this method, signals are exchanged between the parent process and its child process, and the time to complete each round of message exchanging is measured. Singal `SIGUSR1` is used to send between the two processes. The signal handler simply counts how many signals has been received by the current process. When the counter exceeds a predefined value, the process is terminated. To synchronize the send and receive process, the current process waits for the other process's signal to arrive (using `waitpid`) after sending out the signal to the other process. Before the child process forks off, signals are sent within the parent process to time the latency of the signaling mechanism.
- **Measuring context switch time using pipes:** In this method, two pipes are created and shared by both parent and child processes. The two processes communicate by writing a word to one pipe and reading from the other pipe. The descriptors used by the other process for reading and writing are closed in the current process such that each process writes to one pipe and reads from the other. Unlike signals, reading and writing operations on pipes are guaranteed to be synchronized. Latency for using pipes is measured before the child process forks off, by writing and reading from one of the parent process's pipes.
- **Measuring context switch time using shared memory:** Out of curiosity, we measured context switch time using this more primitive method. Shared memory is created using `shmget` and `shmat`. Semaphores are used to avoid race conditions in the shared buffer. We use the producer-consumer model to synchronize the communications between the parent process and the child process. Linux does not provide an easy way to use binary semaphores, the coding is rather convoluted for this method. We have thought that this IPC mechanism should be the fastest among the three. Experiment results suggest that using pipes for IPC is almost as fast as using shared memory and without the hassles of its coding complexity.

Table VI shows the measured context switch time (averaged over ten thousand rounds of communications) using the above three methods.

Table VII shows the measured communication overhead associated with each method.

TABLE VI
CONTEXT SWITCH TIME

Machine	1	2
signal (μs)	1.31	14.92
pipes (μs)	1.32	14.15
shared mem (μs)	1.53	13.28

TABLE VII
IPC OVERHEAD

Machine	1	2
signal (μs)	2.14	15.09
pipes (μs)	1.09	4.93
shared mem (μs)	1.17	4.81

From the data we see that the context switch is much slower on the laptop, which is due to the poor CPU and memory system performance. We conclude that using pipes is ideal for communications between two processes in terms of performance and ease of use.

D. Page Allocation Latency

In this experiment, we measure the latency of a page fault for a zero-filled page. To guarantee the allocated pages are zero-filled, memory is only deallocated after the measurements are done. Fig. 5 shows the skeleton of the code to ensure zero-filled memory allocation.

```
/* number of pages used */
static num_of_pages[]={128,256,512,1024,2048,4096};
         . . .
char *mem[6];
for (i=0;i<6;i++) {
  mem[i]=(char*) malloc(num_of_pages[i]*PAGE_SIZE);
  /* code to measure time used for
     touching each zero-filled page */
         . . .
}
for (i=0;i<6;i++)
   free(mem[i]);
```

Fig. 5. Code to ensure allocating zero-filled page

Page size is operating system dependent. For Linux systems running on Intel's `x86` CPUs, the page size is $4KB$.

Fig. 6 shows the memory allocation time for those tested page sizes (128,256,512,1024,2048,4096). The x axis is in log scale. The memory allocation time using malloc is rather uniform for all page sizes except for small page sizes (i.e. under 1024 pages), in that case the allocation time is slightly shorter. The time used to touch all the pages is shown in Fig. 7. The figure is drawn in log-log scale. The time needed to touch all the pages (by writing one word for each page) is roughly a linear function of the number of pages.

On machine 1, it takes 71113 μs to touch 4096 pages, the average time required to touch a single page is 17.4 μs. On machine 2, it takes 374294 μs to touch 4096 pages, the average time required to touch a single page is 91.4 μs. Again, CPU and memory system have a great influence on page allocation latency for zero-filed pages.

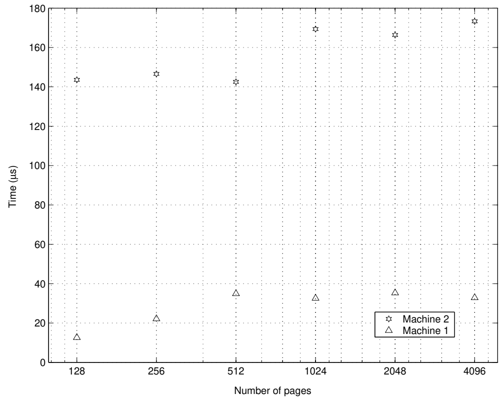

Fig. 6. Allocation time as a function of number of pages

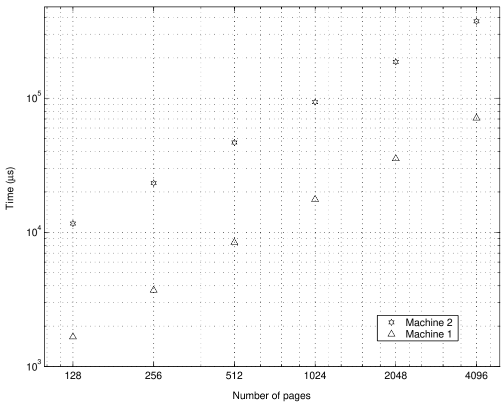

Fig. 7. Time to touch all the pages as a function of number of pages

E. File Read Speed

Our last experiment involves measuring file read speed using two different techniques. We first use the system call read for the test. Then we use memory mapped file to test the read speed for the same file. We first investigate the relationship between the read buffer size and the time required to complete reading the whole file for both techniques. Buffer blocks with sizes of 4,16,64,256,1024,4096,16384,65536 are used for reading. The file used is 64MB in size.

E.1 Cached Read Speed

To ensure the file is in RAM during the entire test, the file is first opened and read (and then rewound) a couple of times before timing the read speed to warm up the cache. The file is not closed until the entire tests are done. For each test, the file is rewound to its beginning using lseek. In this way, the file is ensured to be cached. During the test phase, no disk activities are observed, which further verifies the fact the the file being read is in RAM. On both machines, the file can be held in memory in its entirety.

- **File read using read:** In this experiment, the file is read using the eight different read buffer sizes mentioned above.
- **File read using memory mapped file:** We used mmap in this experiment. Since the whole file is mapped to a memory location, in order to simulate the buffer size effect in read, we use memcpy to copy the memory mapped file from its memory location to a buffer, moving chunks of data to a buffer using the eight read buffer sizes mentioned above. The buffer size equals the size of the chunk being moved (e.g. when using a read size of 4096, a buffer size of 4096 is used for memcpy).

E.2 Uncached Read Speed

Measuring uncached read speed is much trickier than measuring cached read speed. It is guaranteed that when the file is read for the first time, the read is uncached (i.e. file is read directly from the disk). If the file is closed after read, subsequent accesses to the same file may or may not be cached, depending on the current workload of the machine and the virtual memory system's policies. To ensure that the file read activity measured is uncached, the easiest way is to reboot the machine every time before each measurement. While this method works, requiring rebooting the machine takes too much time and is not elegant. We seek an alternative approach. In our method, we page out the file from the physical memory by allocating a significantly large buffer and filling it with random data. For machine 1 (512MB memory), the buffer size is 1024MB, and for machine 2 (128MB memory), the buffer size is 256MB. Since the buffer is much larger than the physical memory, filling it with random data will cause the memory contents to be paged out.

The program for testing uncached read speed is quite similar to that for testing cached read speed. The only differences are that the file is closed after each run and before the next run starts, memory contents are forced to be paged out using the method mentioned above. Experiments show that this method achieves almost identical file read behavior as rebooting the system for each test.

Fig. 8 to Fig. 11 compare the cached and uncached file read performance on both machines. All these figures are plotted in log-log styles. We can see that uncached file read takes significantly longer than cached file read especially when the read buffer is large. We observe the followings from these figures:

- Cached read is faster than uncached read for both methods (`read` or `mmap`) on both machines for all different read buffer sizes. This is especially true for large read buffer sizes. Cached read performance (`read`, for example) can be more than ten times faster than the uncached performance.
- File read using memory mapped file (`mmap`) is faster than `read` on both machines for almost all read buffer sizes. As buffer size increases, the speed difference between `read` and `mmap` decreases dramatically. Increasing buffer size beyond 1024 bytes makes only slight difference in terms of reading speed for both methods.
- Both machines enjoy the speed gain resulted from using larger buffers in the cached file read case more than the uncached case. This is due to the latency of disk operations, which is extremely slow compared to the speed of memory.

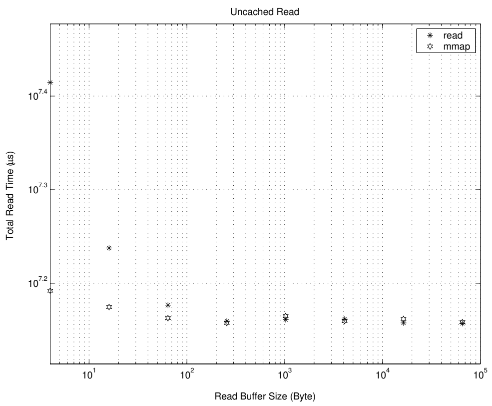

Fig. 8. Uncached file read performance on machine 1

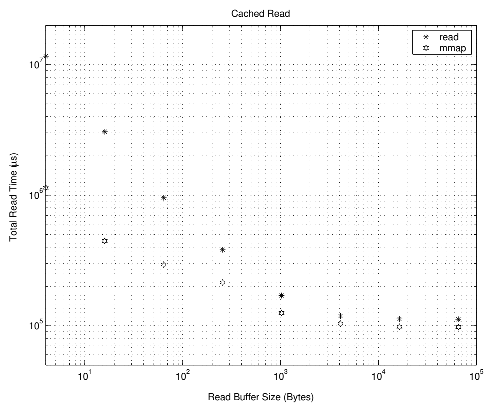

Fig. 9. Cached file read performance on machine 1

We have also done experiment using a fixed size read buffer (4KB) and vary the size of the file to be read (from 1M to 64M). For both cached and uncached file read, the total time required to complete the file read is roughly linear to the file size.

IV. Conclusions

Kernel performance measurement is an important issue. Methods intended to provide accurate measurements are

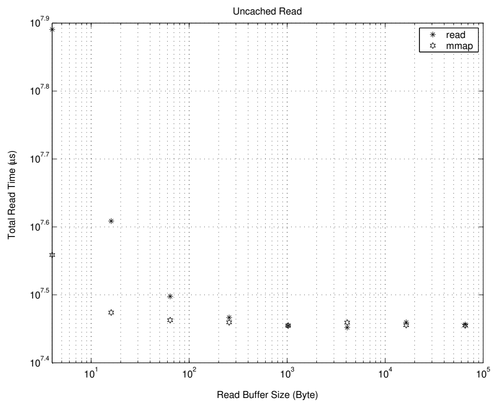

Fig. 10. Uncached file read performance on machine 2

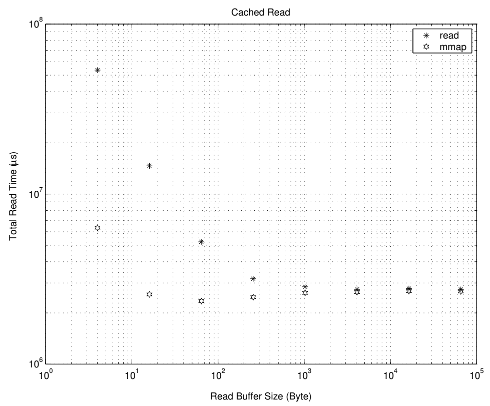

Fig. 11. Cached file read performance on machine 2

discussed in the paper. We conduct our experiments on two different machines with considerably different hardware configurations. Both machines are running Linux operating systems. Various important features related to kernel performance are measured and experiment data are compared. Based on experiment data obtained from both machines, we conclude that hardware performance has significant impact on kernel performance. While the speedups due to faster hardware is not linearly proportional to the hardware speed increase, CPU, memory and disk system have significant impact on kernel performance.