Rethinking Sendfile Support in Unix Kernel

Sharvanath Pathak, Logan Stafman, Wathsala Vithanage
Princeton University

***Abstract*—Traditional web servers, file servers, and proxies transfer files to clients using basic system calls such as UNIX's send/writev. However, these calls incur an overhead due to copying data between user space and kernel space. This can be avoided in certain cases by utilizing UNIX's zero-copy calls, e.g. splice and sendfile. In this paper, we suggest two simple modifications to make the use of sendfile method much more effective. We also show how we can effectively use these modifications to increase performance of existing web servers and proxies.**

Keywords*—*zero-copy, sendfile, Linux networking, TCP

I. Introduction

Traditional web servers used send/writev calls in order to send data to a socket, which often incur the overhead of unnecessary multiple copies and system calls. The most common example where this copy can be avoided is when the program has to send the content of a file to a network socket, the sendfile system call was introduced to Linux in version 2.2[1] to avoid uncessary copying to user buffer in this case. However, there are some aspects of current sendfile implementation which create difficulties for application programmers to use it effectively. There have been other approaches to reduce the overhead of multiple copies in kernel, such as IOLite[2]; however these proposals are not widely accepted by the developer community on the basis that they are too kernel intrusive.

The limitations of the current Linux sendfile method, which are described more thoroughly in section II-A, have also been documented by web servers and proxies in recent past. For instance, NginX, a popular web proxy, includes the following comment in its code base: "we need to disable the use of sendfile() if we use cyclic temp file...", which is one of the two major problems we address in our work. There are several other use cases of our work, some which are discussed thoroughly in section III.

Our goals are to provide enhancements to the current Linux sendfile system call with minimal amounts of changes to the Linux kernel, and minimal changes to applications which choose to take advantage of our modifications. Moreover, applications that use sendfile in its current form should not be required to modify their code. The modifications were implemented in linux kernel 3.6.4, another important goal we heeded to was optimal performance. As we will see in section III, in many cases we can get distinctly better download times by using using the modifications and support we propose.

The rest of paper is organized as following: In the next section, we will discuss the limitations of the current system calls which provide zero-copy behavior in Linux. In section II-B, we will discuss several approaches to resolve these limitations and facilitate effective use of sendfile. In section III, we discuss several use cases for our approach and the results of applying our methodology to these use cases. As this area has received a relatively little amount of attention from the research community, there are many interesting avenues for future work, which have been summarized in section IV. Finally, we present our conclusions in section V.

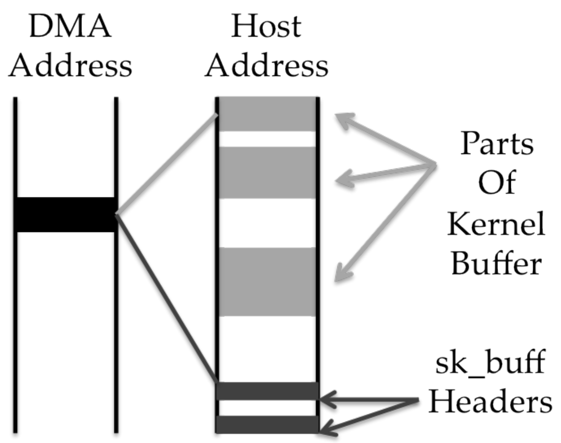

Fig. 1. This describes how DMA uses scatter-gather to allow *sendfile()* to be a true zero-copy function. sk_buff's contain information about the packet used to build headers, (such as packet size), and packet payloads reside in the kernel buffer, both of which are on the host device (main memory). DMA scatter-gather allows these noncontiguous blocks to be combined without use of the CPU.

II. Zero-Copy

In traditional web servers and file servers, files were transferred using Linux's send() and read() system calls. Underneath these system calls, the following steps must take place.

1) A Direct Memory Access (DMA) copy of data from disk memory to a kernel buffer is performed.
2) The kernel buffer is now copied to another buffer which lies in user space.
3) The data is copied from the user space buffer into the kernel's socket buffer (sk_buff).
4) A DMA copy from the kernel's socket buffer to the Network Interface Card (NIC) is performed.

These memory copy operations would typically be repeated many times during the transfer of a large file to a client, making transfers of such large files inefficient.
The zero-copy paradigm was introduced as a solution to this problem. Modern operating systems like Linux introduced implementations of this paradigm, utilizing the power of modern hardware to further reduce the number of excess copies made during the transfer of a file. When Linux's sendfile is called, the following steps occur.

1) A DMA copy from disk memory to a kernel-level buffer is performed.
2) Socket buffers are allocated, but not populated with data.
3) Data is copied via DMA to the NIC using DMA *scatter-gather*.

DMA scatter-gather, available in most modern hardware, allows DMA to copy from multiple non-contiguous blocks of

memory, as is described in Figure 1.
Because sendfile contains no redundant copies between kernel and user space, we save on CPU cycles spent copying memory between buffers, as all copying is now done by DMA. An added advantage is that sendfile can aggregate several other system calls, reducing the amount of time spent calling system calls. Unfortunately, for older hardware which does not support scatter-gather, our modifications to sendfile will not be helpful, as in such cases the buffer contents sk_buff data structures cannot be allocated non-contiguously in memory, as in Figure 1, and hence there has to be seperate socket buffer.
Sendfile in Linux is implemented using the splice operation. Splice is Linux‘s way of passing data between two arbitrary points in memory within the kernel without copying data to user space. When a call to sendfile is invoked, the processs internal pipe (splice_pipe) is initialized with the starting memory address of the pages of the file we are reading from and will be copied to the NIC directly through DMA. This copying process takes place in *tcp_sendpages()* function inside the kernel. *tcp_sendpages()* allocates sk_buff (socket buffers) data structures and sets pointers in them to locations of the memory we going to transmit. Since modern hardware supports DMA scatter-gather it can copy TCP headers and data pointed by sk_buffs even if they are scattered around in physical memory through DMA copy operation without ever copying them to a separate kernel buffer.

A. Limitations of current Zero-Copy implementation in Unix

Unfortunately, *sendfile()*, in it's current implementation, suffers from several problems which hinder its usefulness. The first is a race condition which allows the user to send corrupted content if the user writes to the kernel buffer immediately after a sendfile call. *sendfile()* provides no straightforward API to avoid this race condition, so the only current solution is to wait an "appropriate" amount of time before rewriting the same portion of file.
The second aspect is *sendfile()* is that it blocks if the send buffer is full, although if the scatter-gather dma is supported by NIC, this blocking is just not needed.

1) Race Condition: As with other network write calls(e.g. send and writev) in linux, a call to sendfile blocks when called on blocking socket if the send buffer is not available. Also these network write system calls, including sendfile, might and in many cases do return before the data sent over TCP by the method call has been acknowledged. These methods return as soon as all data is written into the socket buffers (sk_buff) and is pushed to the TCP write queue, the TCP engine can manage alone from that point on. In other words at the time sendfile returns the last TCP send window is not actually sent to the remote host but queued. In cases where scatter-gather DMA is supported there is no seperate buffer which holds these bytes, rather the buffers(sk_buffs) just hold pointers to the pages of OS buffer cache, where the contents of file is located. This might lead to a race condition if we modify the content of the file corresponding to the data in the last TCP send window as soon as sendfile is returned. As a result TCP engine may send newly written data to the remote host instead of what we originally intended to send.

2) Blocking Behavior: As we stated in last paragraph the sendfile call blocks if the last windows is not acknowledged, this makes sense if the data is actually held in socket buffer. Although in cases where the scatter-gather DMA is supported and only pointers are set in the buffer cache, this blocking is useless. We fixed this by just removing the code which leads to this blocking behavior in the case that scatter-gather DMA is supported. This blocking behaviour of sendfile call can hurt the performance of the event based proxies and server, which do not have different threads for each connection.

B. Possible Solutions

The race condition mentioned in last section can be solved if user space application is able to know how many bytes of the file has been received and acknowledged by the recipient. Exposing that information will allow the application to update that safe region of the file without affecting the area of the file which is not yet acknowledged by the recipient. This is especially important when operating on a cyclic file, this case arises in the Nginx proxy described in section III-B. However even with this support solving race conditions that may arise due to multiple writers to the same file needs some application level locking, we will address these issues later in section II-C. We now look at the possible approaches, to be able to expose to applications when the data o which sendfile was called has been acknowledged.

1) Blocking Sendfile: One naive approach is to block the sendfile call until the last byte of data it sent has been acknowledged. There are several problems with approach the most important one being, in event based proxy servers several connections are handled by the same thread and thus blocking this way will hurt a lot of other connections. This approach will not be considered any further because it's obviously not a viable approach to use in the current scalable server implementations.

2) Polling: There are multiple ways to monitor a set of file descriptors. select and poll have been quite common in the past, however on the Linux platform epoll is available to do the same and is more scalable than these polling interfaces. epoll is more efficient because application does not have to pass the entire list of file descriptors every time an event is signaled, epoll also does not traverse (in Kernel space) the list of file descriptors to raise event. In addition epoll also supports both edge triggered and level triggered events whereas poll only supports level triggered events. These features make epoll more efficient than traditional poll operation. However on Linux platform epoll does not work for regular files, as regular files do not have an event interface. But in this case we are interested in events raised on a regular file descriptor passed to the sendfile system call. To overcome this issue an epoll interface has been implemented for regular files. On the other hand standard epoll does not provide a straight-forward way to send information with the event, which is not quite desirable, as we need to inform the user space the number of bytes the recipient has acknowledged so far with the event. We use the u64 field of the union "epoll_data", to stuff the file descriptor and the number of bytes. Although there are a few limitations of this interface, we believe that in most of the cases this simple approach is sufficient enough.

3) Signaling: Signals are software interrupts identified by a number and contains a small amount of information associated with it. Typically the kernel or an application raises signals to notify an event to a user space application. Signals raised from the user space are also routed to the target process via kernel, but in this work we are only interested in signals generate from the kernel. We did implement signaling to be able to expose how much data has been acknowledged, but because it's not so easy to use it in event based proxy servers, we didn't do any experimentation with this.

C. A Concrete Example

Linux follows the advisory locking approach for concurrent access to a file. The problem we are describing here is when the kernel and applications access the same file a mechanism is needed to synchronize. So we think the extra concurrency due to multi-threading of user applications shall be handled by application level synchronization mechanisms. In fact in the current event based servers, these issues don't practically arise because they don't spawn a thread for each connection. In this we discuss a concrete example of how our approach can be used in one practical situation where multi-threading might be a problem.

We now discuss an efficient buffering scheme, which is evaluated in section 5 on lighttpd server. The buffer in this case is materialized by mmapping a temporary file of the size of the buffer we need. Design of this server is simple, in an event loop we have a nonblocking implementation of sendfile with signal or epoll based event mechanism and a writer that modifies the content of a cyclic memory buffer.

Assume that we do not have any information on number of bytes acknowledged so far by the recipient in this case. The worst case that can happen here is given below assuming we have slow writer.

At state t1, sendfile has received TCP ACKs up to r^th byte of the file. Writer is writing bytes to the memory buffer at position r+i. Writer in this case is updating the version v of the buffer to v+1.

At state t2 sendfile passes the point the writer is currently updating and starts sending the recipient old data from version v.

It is now obvious that lack of synchronization is disastrous to such applications. In case of single threaded application we can just add the event each time we call a sendfile and keep track of portions which are not safe to write into. Let us now look at how our sendfile implementation augmented with one additional piece of information "the number of bytes acknowledged so far by the recipient could solve this issue in case of multiple-threads.

Assume that the size of the cyclic memory buffer is B bytes and we logically partition this buffer to n partition length of each logical partition would then be B/n. To maintain the state of each of these logical partitions we will use an array of data structures that will contain the version number of each partition and a binary state whether the partition has been received an acknowledged by the recipient. The array would like this.

V: 1	Ack: T	Lock
V: 1	Ack: T	Lock
V: 1	Ack: F	Lock
V: 1	Ack: F	Lock
V: 0	Ack: T	Lock
V: 0	Ack: T	Lock

Initially all the cells have a version number of 0 and false value for Ack. Assume that sendfile starts sending data. Event handler will update the state of Ack to true in the first cell of the array if and only if it is notified that bytes up to or beyond B/n are received and acknowledged by the recipient.

Sendfile in this case does not operate on the entire memory buffer; it functions at partition granularity. Before calling sendfile the value of Ack and the version should be checked, if Ack is set to false and version number is not less than the version number of previously sent partition then sendfile should called as this is the new version of the buffer that has not been sent to the recipient. In all other cases sendfile should not be called, it could either be a partition, which has not been fully acknowledged yet, or an already acknowledged old version.

Writers should also perform similar checks before updating the corresponding area of the buffer. It should only update the memory area if and only if the value of Ack is T and current version is higher than the version of the partition. Also before updating the memory writer should increment the version vector by one and set the value of Ack to false. If something goes wrong writer should revert values to the previous state before exiting.

Any update to values in the cells representing partitions should be done after acquiring the mutex lock Lock corresponding to that partition.

III. Experimental Evaluations

As discussed later, there are cases where the lack of kernel support for solving these race conditions pose performance limitations on the applications.

A. Experimental setup

We compiled our custom implementation, which has the support for sendfile event and also is non-blocking as stated in section II-A2, on linux 3.6.4. The kernel was installed on a virtual machine running on oracle virtual box 4.2.4. The clients and servers were all connected through a virtual host-only network, the measured throughput of the links was higher than 50MBPS. Before we actually start discussing the benchmarks lets look at the gains of using sendfile over send. In this simple experiment we just had server which sends a file to a client on each request. From fig 2 we see that for very small files there are no gains of sendfile, in fact it does get slow for very small

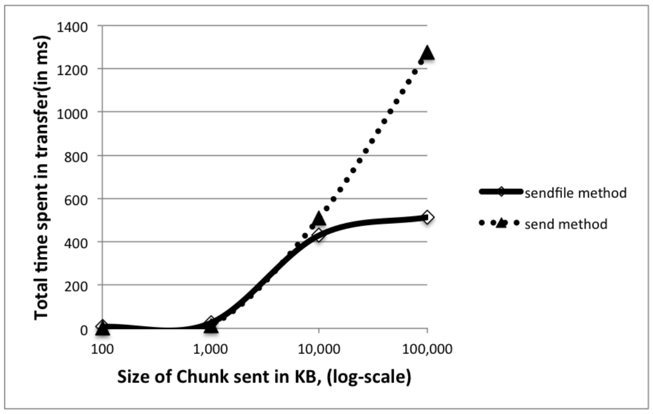

Fig. 2. The variation of download times at client, with increase in file size for sendfile based server and send based server. The server just mmaps the file and sends it to the client by sendfile or send calls depending upon the situation.

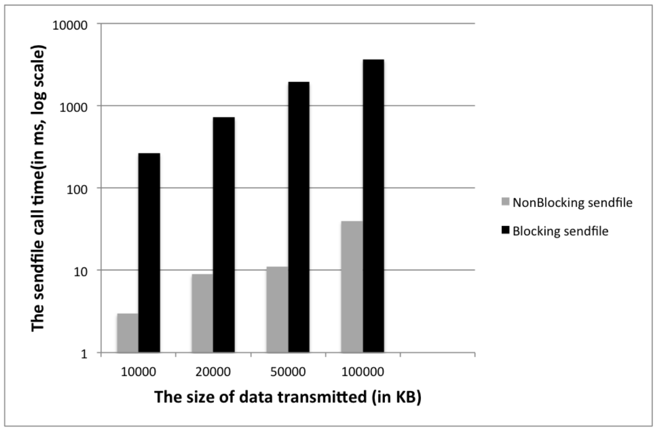

Fig. 3. The comparison of sendfile call times at server, with increase in file size for blocking sendfile based server and non-blocking sendfile based server. Note that Y-axis is in log scale, and hence the non-blocking sendfile has essentially negligible call duration in comparison to the blocking case.

chunks due to some overheads (one example is, because DMA is not on contiguous memory), but for larger chunks sendfile is much better in terms of total time spent. Fig 3 shows the sendfile call duration at client becomes almost negligible when we use non-blocking sendfile(notice that the y axis is in log scale for comparison).

B. Nginx Benchmark

As mentioned in section I, the Nginx proxy modules faces the difficulty of race condition with cyclic temporary files and has clearly commented about the difficulty. The proxy module of Nginx writes the response to a memory buffer, and in case of overflow it writes the response to a temp file[3]. To send that file to a socket it uses sendfile, when ran on Linux. The problem in this case is that they cannot use a cyclic temp file, so if the size of get request from the client downstream is large enough the temp file might overflow as well. In which case it just doesn't read the data from the upstream socket buffer unless the downstream socket buffer is writable. This hurts the download time at the client end. We changed the Nginx code to use a cyclic temp file and the epoll event for sendfile is added to know when it is safe to reuse a portion of the cyclic file. In our case it is much more unlikely that there is no space left in the file, hence the download time for large files get better. We also verified that, if you use a cyclic buffer without adding the events, the file you receive at the client end is different from the file at the server end. This diff easily arises for large files, for instance a 4GB download with 1MB as the size of temp file, always gives the race condition when run on the host-only network(which gives a measured link speed of around 50MBPS), and this is with our custom non-blocking sendfile implementation as discussed in section II-A2. The added benefit of non-blocking implementation as we discussed is that the server is expected to return from the request handling quickly. Given that the major motivation behind the design of Nginx was to solve the C10K problem [4], this is expected to be a good optimization for increasing the capacity to handle more clients.

We compare our approach to the actual Nginx implementation, and find that we perform considerably better of large files. The comparison was done against the standard Nginx, with caching disabled(because in case of caching there is no point of having a a temp file), and was running on the undisturbed linux kernel. Apart from caching no other changes were made for benchmarking. As we discussed the temp file is not cyclic, hence the performance starts getting affected when the temp file overflows. In the customized implementation we used the epoll event to avoid the race condition and used a temp file as small as 1MB, also we had our custom kernel with non-blocking sendfile calls. The size 1MB for temp file was enough to get the desired speed up with cyclic temp file. There were a few cyclic temp file specific optimizations which we performed, firstly we changed the IOVS(the unit of data which is sent and received on each I/O) is increased from 8KB to 100KB, secondly we did a minor code modification to be able to use the temp file more vigorously. Although we verified that Nginx was not any faster with these modifications. As the fig 4 suggests the speed up in download times is considerable for any response which has size greater than a GB, mainly because of the overflow of temp file. Although we didn't actually measured what are the most important factors leading to speedup, apparently the non-blocking part is crucial to the speed-up. With non-blocking sendfile implementation although the probability of race condition becomes considerably large, which means the support we have added to kernel is much more useful this way. Another important benefit which we get using this approach is the disk pressure decreases heavily, i.e. from a GB to a MB of temporary file size. This might not be considered a significant gain in many cases, but with widespread use of flash disks this cost of disk storage is also something that might matter.

C. Lighttpd Benchmark

Another optimization that is possible with using this approach is to have a portion of memory-mapped file act as a

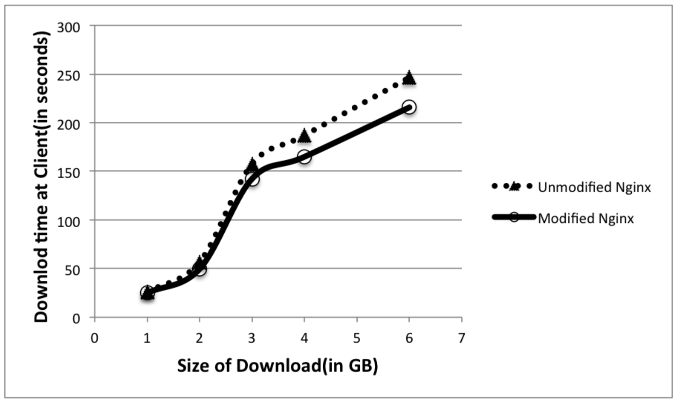

Fig. 4. The variation of download times at client, with increase in file size for both modified Nginx and unmodified Nginx implementations. The comparison is made for when Nginx is set to act as an http proxy. The unmodified Nginx implementation is the untouched versin of Nginx, which runs on unmodified kernel. The modified version uses a cyclic temp file of size 1MB with sendfile enabled, race conditions discussed in section II-A1 are handled using our epoll implementation.

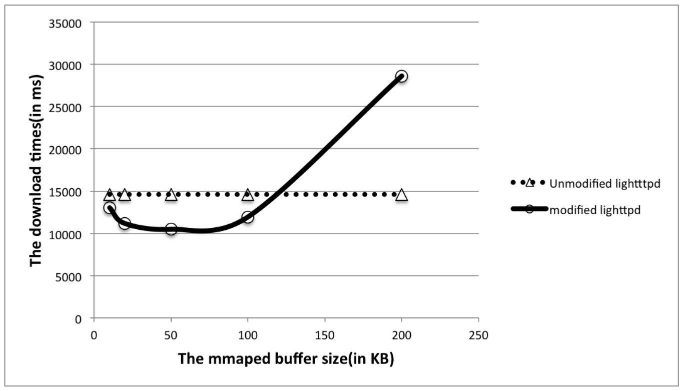

Fig. 5. The variation of download times of response of the modified lighttpd fastcgi module with the variation in size of mmaped temp file.

buffer for the efficient zero copy. We modified the lighttpd server to have a set of files which it memory-maps on initialization and uses to buffer the response of the cgi module. As there is no direct way to do a zero copy send to socket in linux this approach is sometimes beneficial. Conceptually this seems like really good optimization, but the copy cost is really not very significant for small chunks and with large chunk copies we might sometimes get into other issues. We used the download time here as well as the performance metric. We just modified lighttpd to use the above stated buffering scheme with the fastcgi module, we noticed that the speedup is considerable when the size of each unit chunk from cgi module is in a certain range. The problem with small chunks is, it the gain due to copy overhead is overshadowed by the extra-system call overheads. In case of very large chunks, the pipeline starts clogging, i.e. you have to wait for receiving a large chunk of response from cgi before you can start transmitting it. The fig 5 shows the variation of response time of out custom approach and the standard approach with the size of temp file that is mmaped and used as buffer. We call sendfile when the response completes or the temp file is full, hence this acts as the chunk here. Although the question that whether it's possible to go around this clogging and have better buffering scheme with zero copy, is still something which needs exploration.

D. Apache Benchmark

One set of programs that we thought might also benefit from our changes to *sendfile()* is web servers. Apache is a web server which is often used to serve templates created by php. Templates gain several advantages from our enhancements to *sendfile()*. Since templates are only partially dynamic by nature, and our API allows the programmer to determine how many bytes in a file have been sent, a server can begin to reuse memory earlier.

We compared Apache to a custom program which used our sendfile to increase performance. It utilizes the zero-copy behavior of sendfile to reduce time spent copying data. In addition, knowledge of which parts of the template are dynamic allows our program to begin to write over data before we are completely finished sending the file.

In our experiments, we were able to see a noticeable gain for larger templates: for one gigabyte templates, our custom program using sendfile was able to complete the transfer in 10 seconds, while apache took about 14 seconds. However, most templates on the web are orders of magnitude smaller than one gigabyte. Because all of the gain of zero-copy calls is actually in the copying, the gains seen really only apply to larger files. We concluded that our modified sendfile does not actually provide gains for real-world web templates. Programmability gains will be seen for the servers, due to the non-blocking nature of the sendfile call. Note that while Apache does use sendfile currently, its slowdown is due to the fact that it cannot reuse memory because of the current limitations of sendfile discussed earlier.

IV. Future Work

Due to the relatively sparse research until now, there are many unexplored avenues remaining in this area.

From an experimentation standpoint, we would like to extend our work into real world networks in order to evaluate real machines. We would also like to further explore the effects that an implementation of sendfile which blocks would have on current servers and proxies.

We noticed strange behavior during our experimentation when observing loopback connections. Even after all data has been acknowledged by the receiver, the sender can still modify its data and the receiver's data will be modified as well, implying that sendfile actually causes programs to share memory. We discovered that the operating system uses a pipe over the loopback interface, leading to a problem in implementation. A question is raised here as to whether it is more important for the loopback interface to behave as a normal interface, or if the speed of the loopback interface is more important.

From a more general perspective, *sendfile()* is a relatively recent addition to the kernel, leaving several opportunities for improvement. While our work has focused largely on linux, other operating systems may have different approaches, which will have their own unique hurdles to jump. We also believe that many parts of the linux networking stack have remained largely untouched by the research community due to the boundary between operating systems and networking research, leaving many more areas into which we could expand.

V. Conclusions

We have demonstrated that some modifications to the existing sendfile interface will allow applications to use it more effectively. The use of epoll interface in our work made it relatively easy to incorporate this in the existing event based web proxies and servers for linux, moreover the programs which don't want to use our additions can run unmodified. We demonstrate the performance improvements in terms of download times, with the help of several optimization which become possible by our additional support in kernel. Another aspect which we gain on is a large reduction in disk pressure, which might be important with introduction of relatively costly SSDs. We have summarized our future work in previous section, as our finals words we would like to emphasize that the networking stack of OS might have scope for several such useful modifications.

References

[1] (2012, December 29). sendfile(2) - Linux manual page. Retrieved from http://www.kernel.org/doc/man-pages/online/pages/man2/sendfile.2.html

[2] Pai, Vivek S., Peter Druschel, and Willy Zwaenepoel. "IO-Lite: A unified I/O buffering and caching system." Operating systems review 33 (1998): 15-28.

[3] (2012). Http Core Module. in *NginX wiki*. Retrieved from http://wiki.nginx.org/HttpCoreModule

[4] (1999, July 21). The C10K Problem. Retrieved from http://www.kegel.com/c10k.html

[5] Maltz, D. A., & Bhagwat, P. (1999). TCP Splice for application layer proxy performance. Journal of High Speed Networks, 8(3), 225.