A Measurement Study of the Linux TCP/IP Stack Performance and Scalability on SMP systems*

Shourya P. Bhattacharya
Kanwal Rekhi School of Information Technology
Indian Institute of Technology, Bombay
Email: shourya@it.iitb.ac.in

Varsha Apte
Computer Science and Engineering Department
Indian Institute of Technology, Bombay
Email: varsha@cse.iitb.ac.in

***Abstract*— The performance of the protocol stack implementation of an operating system can greatly impact the performance of networked applications that run on it. In this paper, we present a thorough measurement study and comparison of the network stack performance of the two popular Linux kernels: 2.4 and 2.6, with a special focus on their performance on SMP architectures. Our findings reveal that interrupt processing costs, device driver overheads, checksumming and buffer copying are dominant overheads of protocol processing. We find that although raw CPU costs are not very different between the two kernels, Linux 2.6 shows vastly improved scalability, attributed to better scheduling and kernel locking mechanisms. We also uncover an anomalous behaviour in which Linux 2.6 performance degrades when packet processing for a single connection is distributed over multiple processors. This, however, verifies the superiority of the "processor per connection" model for parallel processing.**

I. Introduction

The phenomenal growth of networked applications and their users has created a compelling need for very high speed and highly scalable communications software. At the heart of any application written for use over a network, is the lower layer protocol stack implementation of the underlying operating system. For any application to run at high speed and achieve high scalability, the protocol stack implementation that it runs on must also be high speed, and must not become a bottleneck. Thus it is important to study and understand the performance of the protocol stack implementation which the application will use.

*This work was sponsored by UNM Systems (India) Pvt. Ltd.

Recent trends in technology are showing that although the raw transmission speeds used in networks are increasing rapidly, the rate of increase in processor speeds has slowed down over the last couple of years. While the network backbone speed has increased in orders of magnitude from the 100Mbps Ethernet to Gigabit Ethernet and 10 Gigabit Ethernet, CPU clock frequency has increased linearly [22].

A consequence of this is that network protocol processing overheads have risen sharply in comparison with the time spend in packet transmission [25]. In the case of Internet-based application servers, the protocol processing, i.e. TCP/IP processing, has to be carried out on the general purpose hardware that the server runs on. At high load conditions, the network protocol processing can consume a large fraction of the available computing resources, which can degrade the throughput of the higher layer application.

Several approaches have been employed to scale up the protocol stack implementations. Since parallel processing architectures are becoming increasingly available [16], protocol stack implementations can be optimized to exploit these architectures. In the context of TCP/IP, offloading the protocol processing to dedicated hardware in the NIC, has also been proposed [17], [24].

For the purpose of determining the TCP/IP components that have the highest processing requirements, or to determine how the implementation scales to SMP architectures, a careful performance study of the TCP/IP stack implementation of the operating system in question must be done. In this paper, we discuss the results of such a study done

for the Linux Operating System. The Linux OS has been a popular choice for server class systems due to its stability and security features, and it is now used even by large-scale system operators such as Amazon and Google [11], [13]. In this paper, we have focused on the network stack performance of Linux kernel 2.4 and kernel 2.6. Until recently, kernel 2.4 was the most stable Linux kernel and was used extensively. Kernel 2.6 is the latest stable Linux kernel and is fast replacing kernel 2.4.

Although several performance studies of the TCP protocol stack [1], [6], [9], [10] have been done, this is the first time a thorough comparison of Linux 2.4 and Linux 2.6 TCP/IP stack performance has been carried out. We have compared the performance of these two Linux versions along various metrics: bulk data throughput, connection throughput and scalability across multiple processors. We also present a fine-grained profiling of resource usage by the TCP/IP stack functions, thereby identifying the bottlenecks.

In most of the experiments, kernel 2.6 performed better than kernel 2.4. Although this is to be expected, we have identified specific changes in Linux 2.6 which contribute to the improved performance. We also discuss some unexpected results such as the degraded performance of kernel 2.6 on SMP architecture when processing a single connection on an SMP system. We present fine grained kernel profiling results which explain the performance characteristics observed in the experiments.

The rest of the paper is organised as follows. In Section II, we review previous work in TCP/IP profiling, and discuss some approaches for protocol processing improvement. Section III discusses the improvements made in Linux kernel 2.6 which affect the network performance of the system. Section IV presents results of performance measurement on uniprocessor systems, while Section V discusses results of performance measurement on multiprocessor systems. In Section VI we discuss the kernel profiling results and inferences drawn from it. In Section VII we conclude with our main observations.

II. Background

Several studies have been done earlier on the performance of the TCP/IP stack processing [1], [5], [6], [9], [10]. Copying and checksumming, among others, have usually been identified as expensive operations. Thus, zero copy networking, integrated checksum and copying, header prediction [6], jumbo frame size [1] etc are improvements that have been explored earlier. Another approach has been to offload the TCP/IP stack processing to a NIC with dedicated hardware [24].

Efforts have also been made to exploit the parallelism available in general purpose machines itself, by modifying the protocol stack implementation appropriately. Parallelizing approaches usually deal with trade-offs between balancing load between multiple processors and the overhead due to maintenance of shared data among these processors [3], [4], [19]. Some of the approaches that are known to work well include “processor per message” and “processor per connection” [23]. In the processor-per-message paradigm each processor executes the whole protocol stack for one message (i.e. packet). With this approach, heavily used connections can be efficiently served, however the connection state has to be shared between the processors. In the processor-per-connection paradigm, one processor handles all the messages belonging to a particular connection. This eliminates the connection state sharing problem, but can suffer from uneven distribution of load. Other approaches include “processor per protocol” (each layer of the protocol stack is processed by a particular processor) and “processor per task” (each processor performs a specific task or function within a protocol). Both these approaches suffer from poor caching efficiency.

III. Improvements in Linux kernel 2.6

The Linux kernel 2.6 was a major upgrade from the earlier default kernel 2.4 with many performance improvements. In this section we discuss some of the changes made in kernel 2.6, which can have an impact on the performance of the networking subsystem.

A. Kernel Locking Improvements

The Linux kernel 2.4 uses a lock, termed as the Big Kernel Lock (BKL), which is a global kernel lock, which allows only one processor to be running kernel code at any given time, to make the kernel safe for concurrent access from multiple CPUs [12].

The BKL makes SMP Linux possible, but it does not scale very well. Kernel 2.6 is not completely free of the BKL, however, its usage has been greatly reduced. Scanning the kernel source code revealed that the kernel 2.6 networking stack has only one reference of the BKL.

B. New API - NAPI

One of the most significant changes in kernel 2.6 network stack, is the addition of NAPI ("New API"), which is designed to improve the performance of high-speed networking with two main tricks: *interrupt mitigation* and *packet throttling* [14]. During high traffic, interrupt mitigation allows interrupts to be disabled, while packet throttling allows NAPI compliant drivers to drop packets at the network adaptor itself. Both techniques reduce CPU load.

C. Efficient copy routines

The Linux kernel maintains separate address space for the kernel and user processes for protection against misbehaving programs. Due to the two separate address spaces, when a packet is sent or received over the network, an additional step of copying the network buffer from the user space to the kernel space or vice versa is required. Kernel 2.6 copy routines have therefore been optimised, for the x86 architecture, by using the technique of hand unrolled loop with integer registers [7], [21], instead of the less efficient "movsd" instruction used in kernel 2.4.

D. Scheduling Algorithm

The kernel 2.4 scheduler, while being widely used and quite reliable, has a major drawback: it contains $O(n)$ algorithms where n is the number of processes in the system. This severely impedes its scalability [15]. The new scheduler in kernel 2.6 on the other hand does not contain any algorithms that run in worse than $O(1)$ time. This is extremely important in multi-threaded applications such as Web servers as it allows them to handle large number of concurrent connections, without dropping requests.

IV. Performance on uniprocessor systems

As a first step of the study, we measured "high-level" performance of the two OS versions - that is, without fine-grained profiling of the kernel routines. These tests help us characterise the performance, while the kernel profiling results (Section VI) help us explain those characteristics. Thus, in this section we compare performance measures such as connection throughput and HTTP throughput for Linux 2.4 and Linux 2.6. We also carried out high-level profiling to get a basic idea of the processing needs of the socket system calls. The tests were carried out on two platforms - 1)A single CPU 1.6 GHz Pentium IV machine with 256MB RAM henceforth referred to as the "Pentium IV server" and 2) A Dual CPU 3.2 Ghz Xeon(HT) machine with 512MB RAM, henceforth referred to as the "Xeon server".

A. Performance comparison of socket system calls

In this test, the CPU requirement of the socket system calls was measured using *strace* [26], while clients ran a simple loop of opening and closing connections with servers. The tests were run on kernel-2.4.20 and kernel-2.6.3 on the Pentium IV, with clients and servers on the same machine, connecting over the loopback interface.

Kernel ⇒	2.4.20	2.6.3
socket()	18.05	20.56
bind()	2.91	3.37
listen()	32.37	25.97
connect()	98.97	89.19

TABLE I
AVERAGE TIME SPENT (μS) IN SOCKET SYSTEM CALLS.

The results obtained are shown in Table IV-A. It shows that there is not much difference in the bind() and socket() system call overheads between the two kernels, but the listen() and connect() system calls are slightly cheaper in kernel 2.6. Table IV-A does not show *accept* and *close* system calls, as these

were blocking calls - an accurate resource usage of this could not be obtained from *strace*.

A separate experiment was done to profile the *accept* and *close* system calls. In this experiment, the clients were on a separate machine, with the servers on the Pentium IV server. Instead of using *strace* on the server, the CPU time consumed in server system calls (*accept* and *close*) was estimated by the "utilization law" [8] (i.e. utilization was divided by throughput, to give CPU time). The results, shown in Table II, indicate that the processing time in kernel 2.6 for connection setup and tear down is higher than that of kernel 2.4. This can be attributed to the many security hooks that kernel 2.6 socket code contains. For example, the socket system call in kernel 2.6 additionally invokes the functions `security_socket_create()` and `security_socket_post_create()`.

	CPU Time/conn. (μs)
Kernel 2.4	95.02
Kernel 2.6	105.30

TABLE II

TIME SPENT IN *accept* AND *close* CALLS.

B. Connection throughput

The socket system call profiles did not reveal significant differences in performance between the two kernels. However, these measurements were done at a low to medium load on the system (unto 60% utilization). We are, however, also interested in the maximum achievable capacity of the two kernels - specifically, to confirm whether throughput continues to increase proportionally with the offered load coming to the system. Using the numbers shown in Table II, we can estimate maximum connection rate achievable by Kernel 2.4 to be $\frac{1}{95.02\mu s} \approx 10500$ connections per second and that for Kernel 2.6 to be $\frac{1}{105.3\mu s} \approx 9500$ connections per second.

Figure 1 shows the throughput vs number of active connections for an experiment in which multiple clients repeatedly connected and disconnected from the server without transmitting any data. The peak throughput agrees with the projected capacity quite well, for both the kernels; thus in this experiment, where maximum number of open connections were 100, there were no unexpected bottlenecks.

To stress the system further, we carried out the following experiment: the server had 300 threads with open ports out of which at any time a subset of N ports were made active (i.e. with which clients opened and closed connections). Results obtained from the experiments, where N was varied from 10 to 100 are also shown in Figure 2.

Figure 2 shows a completely different throughput curve for kernel 2.4, and only a slightly different curve for kernel 2.6. While kernel 2.6 throughput continued to show a proportional increase in throughput with increasing load, kernel 2.4 reveals a significant slowdown (e.g. when clients double from 20 to 40, throughput increases from 4000 to only about 5200 - far from double). Kernel 2.6 maximum capacity came down to about 8500 (achieved with 30 connections), while kernel 2.4 capacity was about 9000 - achieved at 100 active connections. We believe this is due to the superiority of the kernel 2.6 scheduler. The kernel 2.4 scheduler has to cycle through all the processes listening on the open ports in the system irrespective of the fact that they are active or not. On the other hand the kernel 2.6 scheduler is relatively less affected by the number of open ports in the system and its performance remains comparable to the earlier case.

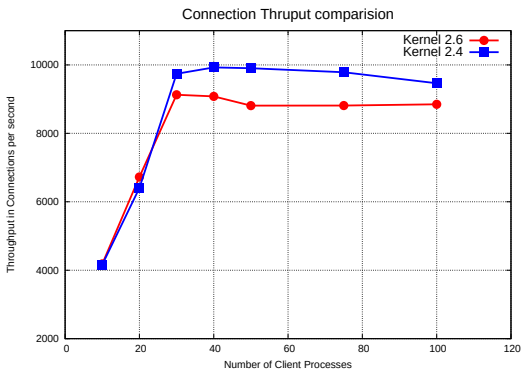

Fig. 1. Connection throughput comparison with varying number of client connection threads.

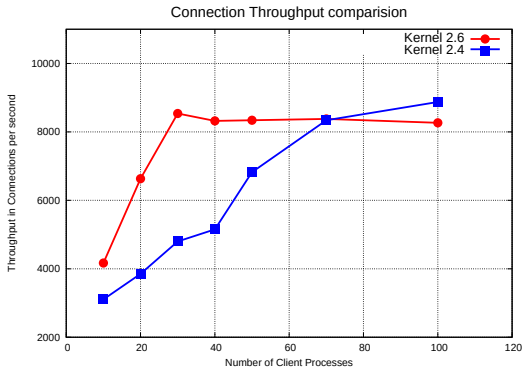

Fig. 2. Connection throughput comparison with 300 open ports and varying number of client connection threads.

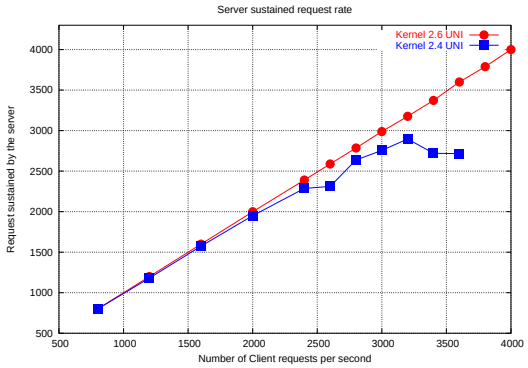

Fig. 3. Request rate sustained by kernel 2.4 and 2.6

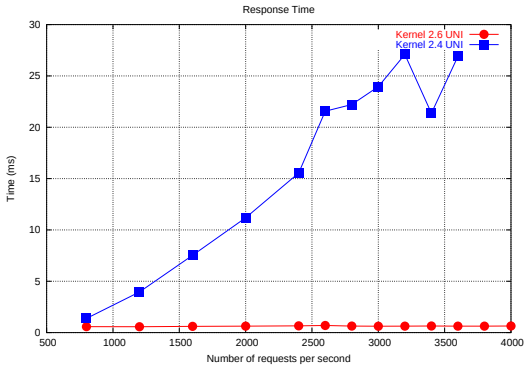

Fig. 4. Response time comparisons of kernel 2.4 and 2.6

C. Web server performance

The previous set of experiments revealed that although raw CPU consumptions of the two kernels were comparable, their scalability characteristics are quite different. In this section, we take our investigation further by observing how these differences impact application layer performance. HTTP was the obvious choice for the application layer protocol, whose performance on the two kernels we wanted to study.

The Apache [2] Web server was run on the Xeon machine (in single processor mode), and load was generated using *httperf* [18]. The following two changes were made to the default Apache configuration file: `MaxClients` was set to the maximum value of 4096 and `MaxRequestsPerChild` was set to zero (unlimited). The clients were made to request a static text page of only 6 Bytes in size (this ensured that the network would not be the bottleneck). The clients generated requests according to a specified rate, using a number of active connections (up to the maximum allowed by Apache).

The maximum client connection request rate sustained by the server and response time for the requests reported by the two kernels are shown in the Figures 3 and 4 respectively.

These graphs show dramatically better HTTP performance on kernel 2.6 as compared with that on kernel 2.4. Kernel 2.4 struggled to handle more than 2800 simultaneous connections and started reporting errors beyond that point. Its connection time and response time also started rising sharply. In contrast kernel 2.6 could easily handle 4000 simultaneous connections and there was no sign of any increase in connection time or response time, suggesting that kernel 2.6 would be able to handle even higher number of simultaneous connections than could be tested. Note that these tests were performed on a faster machine than the previous tests, so the raw CPU consumption differences between the kernels matter even lesser. What dominates performance, are factors that improve *scalability* - e.g. the scheduler.

V. Performance on SMP systems

In the previous section, we focussed on a uniprocessor platform and studied the speed and scala-

bility of the kernels on a single CPU. We would now like to investigate the scalability of the kernels on parallel processing hardware. The basic question that needs to be answered is, if the number of CPUs increases, does the kernel capacity increase proportionally? We answer this question in the case of going from single to dual processors.

We did this by carrying out two types of experiments on the Xeon dual processor machine: bulk data throughput, and connection throughput. We carried out tests with SMP enabled and disabled, and observed the scale up obtained.

A. Bulk data throughput experiments

Bulk data throughput experiments were done using *iperf*, which is a TCP traffic generator. The experiments were run over the loopback interface. The tests were run for a duration of 30 seconds with the TCP window size set to the maximum of 255KB. The buffer size was set to the default value of 8KB. Three sets of experiments were run for each kernel: 1) single TCP connection, single processor, 2) single TCP connection, dual processor, 3) multiple TCP connections, dual processor For kernel 2.6, an additional experiment with a single TCP connection on dual processor with *hyper-threading disabled* was also carried out.

Figure 5 shows the results of the experiments for one and two TCP connections. Figure 6 shows the full set of results with number of TCP connections varying from 1 to 12.

First, consider the scale up achieved by kernel 2.4, as seen in Figure 5. With a single TCP connection on a single processor, kernel 2.4 achieves data throughput of 4.6Gbps, which increases only marginally to 5.1 Gbps with two processors. With 2 TCP connections kernel 2.4 is able to achieve 6.75 Gbps throughput, which amounts to a scale-up factor of 1.5.

Kernel 2.6, as one might expect, was faster than kernel 2.4 in uniprocessor mode, with a throughput of 5.5 Gbps. However its throughput in the SMP mode oscillated between 3.4 Gbits/sec and 7.8 Gbits/sec. Since (as discussed later) this variation seemed to be due to hyper-threading, it was disabled. The throughput then remained consistently at 3.4 Gbps - almost 40% *lesser* than throughput achieved with a single processor. With two TCP connections on two processors, however, kernel 2.6 achieved a throughput of 9.5 Gbps - a factor 1.8 scale-up over that achieved by one TCP connection on uniprocessor.

The higher data throughput of kernel 2.6 in uniprocessor mode is due to its more efficient copy routines as discussed in Section III.

The degradation in throughput of the kernel 2.6 with a single connection on dual processors, can be attributed to “cache bouncing”. In kernel 2.6 because of its better scheduling logic and smaller kernel locks, packet processing can be distributed on all available processors. In our tests, *iperf* creates a single TCP connection and sends data over that connection, but when incoming packets of a connection were processed on different CPUs it would lead to frequent cache misses, as the network buffers cannot be cached effectively in the separate data caches. This results in poorer performance in comparison to the uniprocessor kernel[1].

This also explains the fluctuating high performance (3.4-7.5Gbits/sec) on 2.6 SMP kernel when hyper-threading is enabled. Since the Intel Xeon processors are hyper-threaded, the SMP scheduler randomly schedules the packet processing on two logical processors of the same physical processor. In such a situation there will not be any cache penalty as the logical processors will have access to the same cache. The results with HT disabled verify this explanation. Later, in section VI we discuss kernel profiling tests which further confirm these ideas.

The graph that shows throughput with increasing number of TCP connections (Figure 6) confirms the expectation that with two processors, going beyond two TCP connections (each sending bulk data), does not achieve further scale up. In fact, in SMP

[1]It is not entirely clear why Kernel 2.6 does not show this behaviour for two TCP connections. It is possible that in the case of the single connection, the effect of distributing the work between the two processors is more pronounced, since the processors are more ‘available’. In the two TCP connections case, both processors will be busy with a connection each, and the Kernel will usually not find the other processor “available” to work on a connection that one processor is currently working on, thus leading to processor affinity. However, these are conjectures, and we do not have a definite explanation.

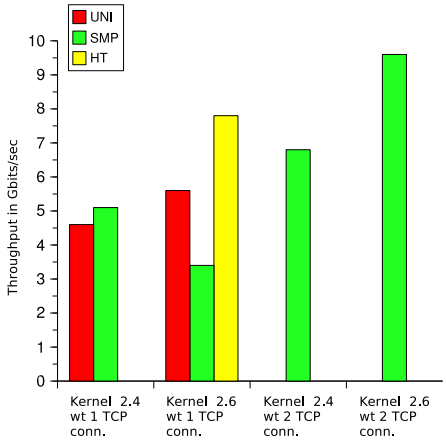

Fig. 5. Graphical representation of the data transfer rates achieved in different test cases

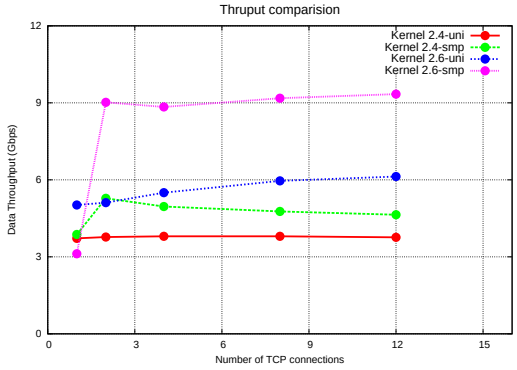

Fig. 6. Data throughput with varying number of TCP connections

kernel 2.4, the data throughput rises initially with two simultaneous connections but drops slightly as the number of parallel TCP connections increased, implying that kernel 2.4 SMP incurs some penalties while multiplexing multiple TCP streams on a physical processor.

B. Connection throughput on SMP systems

We carried out similar experiments as above, focussing this time on connection throughput, to study the SMP scalability of connection set-up and teardown operations. Figure 7 plots the connection rate achieved vs number of clients. Note that throughput

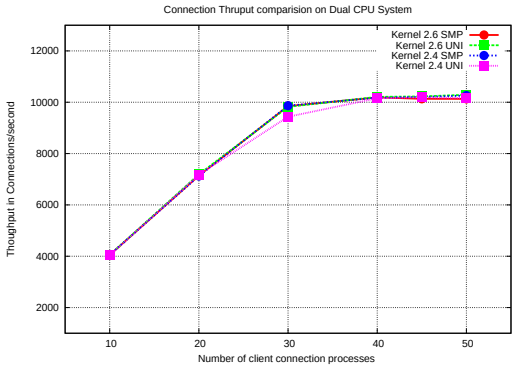

Fig. 7. Measured Connection throughput on Dual CPU system with 300 open server ports

peaks at 10000 connections per second for all cases, *due to client bottleneck*. Since the server could not be pushed to its maximum capacity with the available hardware, we used a different approach to characterize scalability: we measured utilisation of the server CPUs and drew conclusions from that.

Table III shows the utilisation measured for the experiment with 30 clients. First consider kernel 2.4. We can see that for the same throughput of about 10000 connections per second, the utilisation of the dual processor is almost the same as that of the uniprocessor - the ideal would be half of the utilisation of the single processor. For kernel 2.6 the results are significantly better. For the same throughput, utilisation of the dual processor system is two-thirds of the uniprocessor utilisation. These results again demonstrate the superior scalability of the Linux kernel 2.6. The kernel 2.4 results show that there would be almost no benefit in adding hardware resources, if a kernel 2.4 system shows a connection throughput bottleneck. However, adding CPUs will definitely relieve similar bottlenecks in the case of kernel 2.6.

VI. Kernel profiling results

In addition to the above high-level experiments, we did detailed profiling of both the Linux kernels using OProfile [20]. Oprofile is a statistical profiler that uses hardware performance counters available on modern processors to collect information on executing processes. The profiling results provide

	Server Utilisation
Kernel 2.4 UNI	20.08 %
Kernel 2.4 SMP	21.09 %
Kernel 2.6 UNI	21.00 %
Kernel 2.6 SMP	15.58 %

TABLE III

SERVER UTILISATION FOR NUMBER OF CLIENTS = 30.

valuable insight and concrete explanation of the performance characteristics observed in the previous experiments- specifically, the observed anomalous behaviour in section V of SMP kernel 2.6, processing a single TCP connection on a dual CPU system.

A. Breakup of TCP packet processing overheads

The breakup of TCP packet processing overheads are shown in Table IV. It lists the kernel functions that took more than 1% of the overall TCP packet processing time. The function `boomerang_interrupt` function is the interrupt service routine for the 3COM 3c59x series NIC, which was used in our experiments. The other `boomerang_*` functions are also part of the NIC driver involved in packet transmission and reception. `__copy_from_user_ll` copies a block of memory from the user space to kernel space. `csum_partial` is the kernel checksumming routine.

Thus we can see that the NIC driver code, interrupt processing, buffer copying, checksumming are the most CPU intensive operations during TCP packet processing. In comparison TCP functions take up only a small part of the overall CPU time.

B. Analysis of kernel 2.6 SMP anomaly

In Section V we had observed that there was a sharp drop in the performance of SMP kernel 2.6 when a single TCP connection was setup on a dual CPU system, as compared with the uniprocessor system, but as the number of TCP flows were increased to 2 and more, kernel 2.6 performed extremely well.

To analyse this anomalous behaviour, we re-ran the data throughput experiments for kernel 2.6 in both SMP and Uni-Processor mode, and profiled

CPU Samples	%	Function Name
24551	12.0273	boomerang_interrupt
15615	7.6496	boomerang_start_xmit
14559	7.1323	__copy_from_user_ll
12037	5.8968	issue_and_wait
8904	4.3620	csum_partial
6811	3.3366	mark_offset_tsc
5442	2.6660	ipt_do_table
5389	2.6400	csum_partial
4913	2.4068	boomerang_rx
4806	2.3544	ipt_do_table
3654	1.7901	tcp_sendmsg
3426	1.6784	irq_entries_start
2832	1.3874	default_idle
2382	1.1669	skb_release_data
2052	1.0053	ip_queue_xmit
2039	0.9989	tcp_v4_rcv
2013	0.9862	timer_interrupt

TABLE IV

BREAKUP OF TCP PACKET PROCESSING OVERHEADS IN THE KERNEL

CPU Samples	%	Function Name
122519	13.1518	__copy_from_user_ll
94653	10.1605	__copy_to_user_ll
45455	4.8794	system_call
41397	4.4438	(no symbols)
35921	3.8559	schedule
35829	3.8461	tcp_sendmsg
31186	3.3477	__switch_to

TABLE V

TCP PACKET PROCESSING OVERHEADS IN KERNEL 2.6 UNI WITH A SINGLE TCP CONNECTION

the kernel during that period. In these experiments, each TCP connection sent and received exactly 2GB of data. This allowed us to directly compare the samples collected in both the situations.

The most striking fact emerging from Table VI and V is the large increase in time spent in the kernel copy routines. The functions `__copy_from_user_ll()` and `__copy_to_user_ll()` are used for copying buffers from user space to kernel space and from kernel space to user space respectively. There is a very sharp increase in the time spent by these two functions of the SMP Kernel with a single TCP connection. More than 50% of the

CPU 0 Samples	%	CPU 1 Samples	%	Total	%	Function Name
373138	28.54	417998	31.72	791136	30.1354	__copy_from_user_ll
293169	22.42	264998	20.1076	558167	21.2613	__copy_to_user_ll
74732	5.71	82923	6.2921	157655	6.0053	tcp_sendmsg
26537	2.02	24047	1.8246	54153	2.0628	schedule
25327	1.93	28826	2.1873	50584	1.9268	(no symbols)
23410	1.79	23275	1.7661	46685	1.7783	system_call
21441	1.64	21757	1.6509	43198	1.6455	tcp_v4_rcv

TABLE VI

TCP PACKET PROCESSING COSTS IN KERNEL 2.6 SMP WITH SINGLE TCP CONNECTION

CPU 0 Samples	%	CPU 1 Samples	%	Total	%	Function Name
129034	11.215	127949	11.1399	256983	11.1775	__copy_from_user_ll
121712	10.5786	127182	11.0731	248894	10.8256	__copy_to_user_ll
59891	5.2054	56946	4.958	116837	5.0818	schedule
47992	4.1712	46888	4.0823	94880	4.1268	tcp_sendmsg
44767	3.8909	46023	4.007	90790	3.9489	system_call
32413	2.8172	30421	2.6486	62834	2.733	__switch_to
26090	2.2676	25822	2.2482	51912	2.2579	tcp_v4_rcv

TABLE VII

TCP PACKET PROCESSING COSTS IN KERNEL 2.6 SMP WITH TWO TCP CONNECTION

total time is spent in these functions. Such a sharp increase in the cost of copy routines can be attributed to a high miss rate of processor cache. To verify this, the `__copy_from_user_ll()` and `__copy_to_user_ll()` routines were further analysed and it was found that more than 95% time in these routines were spent on the assembly instruction

```
repz movsl %ds:(%esi),%es:(%edi)
```

The above instruction copies data between the memory locations pointed by the registers in a loop. The performance of the `movsl` instruction is heavily dependent on the processor data cache hits or misses. The significantly higher number of clocks required by the `movsl` instruction in the case of SMP kernel 2.6, for copying the same amount of data can only be explained by an increase in the data cache misses of the processor.

If you further compare the Tables VII and V you will observe that the times spent by the kernel functions are very similar, i.e. when two TCP connections are run on the dual CPU system, both CPU's are utilised in a similar pattern as when a single TCP connection running on a uniprocessor system is utilised. This is unlike the case of a single TCP connection running on dual CPU system.

VII. CONCLUSION

Our study of the performance and scalability of the two Linux kernels primarily confirms that performance of software used for communications can be impacted greatly by the underlying operating system. E.g. the HTTP results show dramatic difference in throughput and response time measures when operating over the two different kernels.

The improvements made in kernel 2.6 seem to have had a great impact on its performance. Kernel 2.6 could handle extremely large number of simultaneous connections and sustained higher data transfer rates. We were able to correlate these observations with the architectural changes in kernel 2.6, specifically its $O(1)$ scheduler, efficient copy routines and finer kernel locks.

The experiments offer valuable insights into the SMP behaviour of the TCP stack. In kernel 2.6 we were able to get a scale up of more than 1.8x in the data throughput tests, on a dual CPU system

with two or more TCP connections, while kernel 2.4 showed a scale up of less than 1.5x.

We also identified the most dominant overheads involved in packet processing, namely, the interrupt costs, device driver overheads, checksumming and buffer copying. TCP layer overheads were comparatively insignificant.

One of the most significant conclusions that can be drawn from our SMP experiments is that the data throughput of a TCP connection is heavily dependent on the processor cache. An inference that can be drawn from this is that in traditional SMP systems where each processor has a separate cache, the OS scheduler should follow the "processor per connection" paradigm. One the other hand, if the data cache gets shared by the processing cores like in HT technology, then "processor per message" approach can be effective.

REFERENCES

[1] V. Anand and B. Hartner. *TCP/IP Network Stack Performance in Linux Kernel 2.4 and 2.5*. World Wide Web, http://www-124.ibm.com/developerworks/opensource/linuxperf/netperf/ols2002/tcpip.pdf, 2002.

[2] *The Apache Software Foundation*. World Wide Web, www.apache.org.

[3] M. Björkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Conference proceedings on Communications architectures, protocols and applications*, pages 74–83. ACM Press, 1993.

[4] M. Björkman and P. Gunningberg. Performance modeling of multiprocessor implementations of protocols. *IEEE/ACM Trans. Netw.*, 6(3):262–273, 1998.

[5] J. Chase, A. Gallatin, and K. Yocum. End systems optimizations for high speed TCP. *Communications Magazine, IEEE*, 39(4):68–74, April 2001.

[6] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. pages 23–29. IEEE Communications, 1989.

[7] J. W. Davidson and S. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 125–132. IEEE Computer Society Press, 1995.

[8] R. Jain. *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. Wiley-Interscience, New York, April 1991.

[9] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in tcp/ip. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, pages 259–268, New York, NY, USA, 1993. ACM Press.

[10] J. Kay and J. Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Trans. Netw.*, 4(6):817–828, 1996.

[11] *How Linux saved Amazon millions*. World Wide Web, http://news.com.com/2100-1001-275155.html?legacy=cnet.

[12] *Big Kernel Lock lives on*. World Wide Web, https://lwn.net/Articles/86859/.

[13] *Google Relies Exclusively on Linux Platform to Chug Along*. World Wide Web, http://www.hpworld.com/hpworldnews/hpw0 09/02nt.html.

[14] *Network Drivers*. World Wide Web, http://lwn.net/Articles/30107/.

[15] *Linux Kernel 2.6 Scheduler Documentation*. World Wide Web, http://josh.trancesoftware.com/linux/.

[16] D. Marr, F. Binns, D. Hill, G. Hinton, and D. Koufaty. *Hyper-Threading Technology Architecture and Microarchitecture*. Intel Technology Journal, http://www.intel.com/technology/itj/2002/volume06issue01/art01_hyper/p15_authors.htm, 2002.

[17] J. C. Mogul. TCP offbad is a dumb idea whose time has come. In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[18] D. Mosberger and T. Jin. httperf—a tool for measuring Web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[19] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. F. Towsley. Performance issues in parallelized network protocols. In *Operating Systems Design and Implementation*, pages 125–137, 1994.

[20] *OProfile profiling system for Linux 2.2/2.4/2.6*. World Wide Web, http://oprofile.sourceforge.net.

[21] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 147–155. IEEE Computer Society, 1999.

[22] K. Quinn, V. Turner, and J. Yang. The next evolution in enterprise computing: The convergence of multicore x86 processing and 64-bit operating systems. Technical report, IDC, April 2005.

[23] D. C. Schmidt and T. Suda. Measuring the impact of alternative parallel process architecture on communication subsystem performance. In *Protocols for High-Speed Networks IV*, pages 123–138. Chapman & Hall, Ltd., 1995.

[24] P. Shivam and J. S. Chase. On the elusive benefits of protocol offbad. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 179–184. ACM Press, 2003.

[25] J. M. Smith. Programmable networks: Selected challenges in computer networking. *Computer*, 32(1):40–42, 1999.

[26] *Strace Homepage*. World Wide Web, http://www.liacs.nl/ wichert/strace/.