Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

TCPIP Network Stack Performance in Linux Kernel 2.4 and 2.5

Vaijayanthimala Anand, Bill Hartner
IBM Linux Technology Center
manand@us.ibm.com, bhartner@us.ibm.com

Abstract

We discuss our findings on how well the Linux 2.4 and 2.5 TCPIP stack scales with multiple network interfaces and with the SMP network workloads on 100/1000 Mb Ethernet networks. We identify three hotspots in the Linux TCPIP stack: 1) inter-processor cache disruption on SMP environments, 2) inefficient copy routines, and 3) poor TCPIP stack scaling as network bandwidth increases.

Our analysis shows that the L2 cache_lines_out rate (thereby memory cycles per instruction-mCPI) is high in the TCPIP code path leading to poor SMP Network Scalability. We examine a solution that enhances data cache effectiveness and therefore improves the SMP scalability. Next the paper concentrates on is improving the "Copy_To_User" and "Copy_From_User" routines used by the TCPIP stack. We propose using the "hand unrolled loop" instead of the "movsd" instruction on the IA32 architecture and also discuss the effects of aligning the data buffers. The gigabit network interface scalability workload clearly shows that the Linux TCPIP stack is not efficient in handling high bandwidth network traffic. The Linux TCPIP stack needs to mimic the "Interrupt Mitigation" that network interfaces adopt. We explore the techniques that would accomplish this effect in the TCPIP stack. This paper also touches on the system hardware limitation that affects the gigabit NIC's scalability. We show that three or more gigabit NICs do not scale in the hardware environment used for the workloads.

1 Introduction

Linux is widely deployed in the web server arena and it has been claimed that Linux networking is optimized to a great extent to perform well in the server network loads such as file serving and web serving, and in packet forwarding services such as firewalls and routers. Linux scales well horizontally in cluster environments which are used for web servers, file servers etc.; however, our studies on IA32 architecture show that the TCPIP network stack in the Linux Kernel 2.4 and 2.5 lack SMP network scalability as more CPUs are added and lack NIC scalability on high bandwidth network interfaces when more NICs are added.

1.1 SMP Network Scalability

Cache memory behavior is a central issue in contemporary computer system

performance[6]. Much work has been done to examine the memory reference behavior of application code, operating system and network protocols. Most of this kind of work on network protocols concentrates on uniprocessor systems. This paper discusses the data and instruction memory reference effects of Linux TCPIP stack in a multi-processor system. We examine the L2_cache_line_out rate and instructions retired rate in the receive path of TCPIP and Ethernet driver to understand memory latency.

In IA32 Linux, interrupts from different network interfaces (NICs) are routed to CPUs in a dynamic fashion. The received data, its associated structures that deal with a particular connection, and its activities get processed in different CPUs due to the dynamic routing, which results in non-locality of TCPIP code and data structures, which increases the memory access latency leading to poor performance. This effect is eliminated when the application process, and the interrupt for the particular network interface are aligned to run on the same CPU. By binding the process and interrupt to a CPU, a given connection and its associated activities including the data processing during the duration of that connection are guarenteed to process on the same CPU. This binding results in better locality of data and instructions, and improving the cache effectiveness. Affinitizing process and interrupt to a CPU may be feasible in a single service dedicated server environment, but may not be desirable in all situations. Therefore, reducing the number of L2 lines that bounce between the caches in the TCPIP stack code path is a critical factor in improving the SMP scalability of the TCPIP stack. We use affinity as a tool to understand how TCPIP SMP scalability can be improved.

The Inter-processor cache line bouncing problem can be generally addressed by improving the data memory references and instruction memory references. Instruction cache behavior in a network protocol such as TCPIP has a larger impact on performance in most scenarios than the data cache behavior [6, 2]. Instruction memory refereces may be solely important in scenarios where zero copy [1] is used or scenarios where less data is used. When zero copy is not used, reducing the time spent on the data memory reference considerably improves performance.

In the TCPIP stack, under numerous conditions, the received data is check summed in the interrupt (softirq) handler and is copied to user buffer in the process context. These two contexts, interrupt and process, are frequently executed on different processors due to the dynamic interrupt routing and how processes are scheduled. We proto-typed a patch that forces the csum and copy to happen on the same processor for all conditions resulting in performance improvement. Linux TCPIP does have routines that fold csum into copy; however, these routines are not used in all the code paths. We also show profiling data that supports the need to improve both data and instruction memory references in TCPIP stack.

1.2 Efficient Copy Routines

Not only did we consider reducing the memory reference cycles for data in SMP environment but we also considered it for uniprocessor by improving temporal and spatial locality for data copy. Copying data between user and kernel memory takes a big chunk of network protocol processing time. Zero copy again is the mechanism to reduce this processing time, and improving the packet latency. Even with zero copy, the copy is eliminated only on the send side; improving the copy routines in TCPIP stack would help the receive side processing. We found that the copy routines used in IA32

Linux TCPIP stack can be made more efficient by using "unrolled integer copy" instead of the string operation "movsd." This paper presents measurements to show that "movsd" is more expensive than "unrolled integer mov." We also look at the effects of alignment on Pentium III systems. Measurements presented in section 3 show that "movsd" performs better if both the source and the destination buffers are aligned.

1.3 TCPIP Scalability on Gigabit Network

Finally this paper looks at the performance of the TCPIP stack using gigabit bandwidth network interfaces (NICs). We use NIC hereafter in this paper to mean Network adapters. In this paper we concentrate on receive side processing to discuss how the TCPIP stack may be improved for handling high bandwidth traffic. The techniques discussed in this paper for receive side processing are also applicable to transmit side processing but we have not evaluated transmit side performance. This paper presents analysis data to show why TCPIP should process packets in bunches at each layer rather than one packet at a time. High bandwidth network interface cards implement a technique called "interrupt mitigation" to interrupt the processor for a bunch of packets instead of interrupting for each received packet. This paper suggests that this "interrupt mitigation" should be mimicked in higher layer protocols as "packet mitigation." We also look at how the hardware that we used has limitation that leads to poor gigabit NIC scaling.

The main contribution of this paper is to bring to light the Linux TCPIP SMP scalability problem caused by cache line bouncing among multiple processors in the Linux 2.4.x and 2.5.x kernels. Second this paper points out that the Linux TCPIP stack needs to be tailored to more efficient protocol processing for high bandwidth network traffic.

The rest of the Introduction Section discusses the benchmark, hardware and software environments used. Each section includes the benchmark results, analysis data, description of the problem if one exists, and a technique or a patch that could alleviate the bottleneck. Section 2 deals with the SMP Scalability problem that in Linux 2.4 kernel and also shows how the 0(1) scheduler[8] has improved this scalability to some extent in 2.5 kernel. In section 3 we discuss about efficient copy routines in the TCPIP stack followed by section 4 that discusses TCPIP stack scalability on high bandwidth networks and hardware limitation that causes poor gigabit NIC scalability.

BENCHMARK ENVIRONMENT

Netperf [10] is a well-known network benchmark used in the open source community to measure network throughput and packet latency. Netperf is available at www.netperf.org. Netperf3 is an experimental version of Netperf that has multi-thread and multi-process support. We extended Netperf3 to include multi-adapter and synchronized multi-client features to drive 4-way and 8-way SMP servers and clients. Netperf3 supports streaming, request response, and connection request response functions on TCP and UDP. We used TCP_STREAM for our work so far.

In this paper we evaluate SMP scalability and network NIC scalability using the TCP_STREAM feature of Netperf3. We used multi-adpater workloads between a server and a client for SMP scalability study and used multiple clients and a server with multiple NICs for NIC scalability. The TCP_STREAM test establishes a control socket connection and

a data connection then sends streams of data using the data connection. At the end of a fixed time period, the test ends and the total throughput is reported in Mbits per second. The principal metric we used for TCP_STREAM is the throughput and "throughput scaled to 100% CPU Utlization." Vmstat is used to measure the CPU Utlization at the server. "Throughput scaled to 100% CPU Utlization" is derived using the throughput and the cpu Utlization. The "throughput scaled to CPU" is the throughput which would result if all CPUs could be devoted to the throughput. We refer "throughput scaled to 100% CPU Utlization" as "throughput scaled to CPU" in the rest of the paper.

We used isolated Ethernet 100Mbit and 1000Mbit network for our workloads. Open source profiling tools such as SGI kernprof[7], and SGI lockmeter[3] were used for analysis. Where necessary, additional tools were developed to gather profiling data. In addition to time based profiling of SGI Kernprof, we used Kernprof's Pentium performance-counter based profiling. In performance-counter based profiling, a profile observation is set to occur after a certain number of Pentium performance counter events[4]. We used a frequency of 1000 for our profiling, i.e., for every 1000 cache_lines_out or 1000 instructions retired, the profile records the instruction location. Thus the profile shows where the kernel takes most of its cache_lines_out or where the kernel executes most of its instructions.

HARDWARE AND SOFTWARE ENVIRONMENT

The 100Mb Ethernet test was run on an 8-way Profusion chipset, using 700 Mhz Pentium III Xeon with 4 GB memory and 1 MB L2 cache as our client and a 500 Mhz Pentium III 4-way SMP system with 2.5 GB memory, and 2 MB L2 cache as our server. The NIC cards used were Intel Ethernet PRO/100 Server PCI adapters. For 1000Mb Ethernet, we used the same 8-way system as our server and Pentium III 2-way clients with 1 GB memory (4 of them). The NIC cards used were Intel Ethernet PRO 1000 XF Server PCI adapters. We used both 2.4.x and 2.5.x Linux kernels. In both cases the server and client(s) are connected point-to-point using cross over cables.

2 TCPIP SMP Scalability Using Ethernet 100Mb

2.1 Netperf3 TCP_STREAM results

We measured the SMP scalability of the 2.4.17 and 2.5.3 TCPIP stack using the Netperf3 multi-adapter TCP_STREAM test. The server used was a 4-way 500 Mhz PIII with four 100 Mb ethernet NICs operating in full duplex mode with a MTU size of 1500 bytes. The server NICs (receive side) were connected to the client NIC (send side) point-to-point. We used 8-way, 4-way and 2-way clients to drive 4-way, 2-way and 1-way servers respectively.

A single TCP connection was created between the server and the client on each of the networks for a total of four connections. A process was created for each of the NICs for a total of (4) server processes. The test was run using the application message sizes of 1024, 2048, 4096, 8192, 16384, 32768 and 65536 bytes. The tcp socket buffer size was 65536 bytes and TCP NODELAY OPTION was set on the socket. Both the network throughput (Mb/sec) and CPU utilization were measured on the server. The throughput scaled to CPU utilization is derived from these two measurements. Especially for the 100 Mb Ethernet workload, the "throughput scaled to CPU" is

the important measure since the throughput of each of the 4 adapters reaches maximum media speed in all tests.

We ran the test with 1, 2, and 4 CPUs enabled on the server. The 2P and 4P SMP scalablity factor was calculated by dividing the respective "throughput scaled to CPU utilization" by the UP "throughput scaled to CPU utilization."

In Figure 1 and Figure 2, we report the 2P and 4P SMP scalability factor of the 2.4.17 and the 2.5.3 kernel for the Netperf3 TCP_STREAM test using the different Netperf3 message sizes. Note that the 4P scalability factor for the 2.4.17 kernel using the 65536 message size barely achieves 2/4 scaling. The 4P scalability factor of the 2.5.3 kernel is much improved reaching 2/4 in most of the cases. The better 4P scalability of the 2.5.3 kernel may be attributed to the 0(1) scheduler which provides better process affinity to a CPU than the 2.4.17 scheduler.

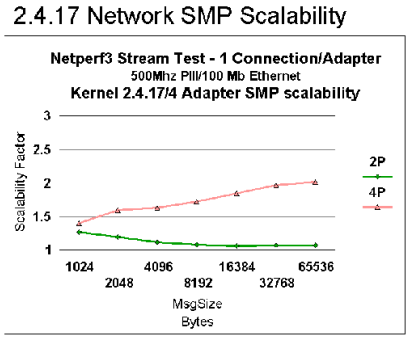

Figure 1: SMP Network Scalability on 2.4.17 Kernel

The SMP TCPIP scalability for 4P and 2P as shown in Figures 1 and 2 is poor. To understand this problem we ran some tests. We choose 4P and 4 adapter cases for our experiment. First we examined the effects

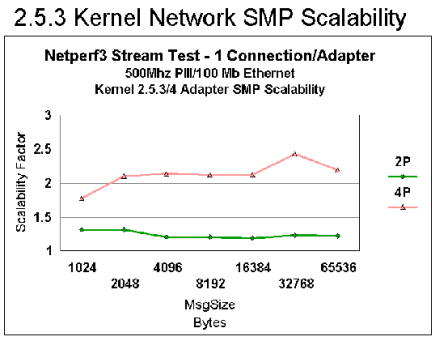

Figure 2: SMP Network Scalability on 2.5.3 Kernel

of IRQ and process affinity on the Netperf3 TCP_STREAM test for the 2.4.17 kernel using the 4-way server. IRQ affinity involves binding each of the four network adapter IRQs to one of the (4) CPUs. For example, the interrupt assigned to NIC 1 to CPU 0, NIC 2 to CPU 1 etc. PROCESS affinity involves binding each of the four Netperf server processes to one the four CPUs. Both PROCESS and IRQ affinity can be combined by affinitizing the IRQ for NIC and netperf3 server process servicing the tcp connection to the same CPU. In table 1, we report the percent improvement of the Netperf3 TCP_STREAM throughput scaled to CPU when IRQ affinity, PROCESS affinity and BOTH affinities were applied to the server. The percentage improvement is relative to the throughput scaled to CPU utilization when no affinity was applied. The higher the percentage, better the performance.

For the case of IRQ affinity, throughput scaled to CPU utilization improved 4 to 21% for the various message sizes. For the case of PROCESS affinity, throughput scaled to CPU utilization improved 6 to 28%. For the case of both IRQ and PROCESS affinity applied, throughput scaled to CPU utilization improved

42 to 74%. The gain in the third case, where both IRQ and PROCESS affinity are applied, is much greater than the sum of the results of these two affinities applied separately.

Msgsize (Bytes)	IRQ Affinity (%)	PROCESS Affinity (%)	BOTH Affinities (%)
1024	4.34	12.71	74.53
2048	21.45	21.43	66.56
4096	19.01	28.73	74.68
8192	19.18	25.25	68.03
16384	16.36	14.37	55.22
32768	11.38	11.38	47.45
65536	11.31	6.34	42.09

Table 1: Affinity Comparison using 2.4.17 Kernel ON 4P using 4 NICS

Figure 3 shows the comparison of the TCP_STREAM throughput scaled to CPU utilization for the 2.4.17 kernel with 1) no affinity 2) IRQ affinity 3) PROCESS affinity 4) BOTH affinities, and 5) the 2.5.3 kernel with no affinity applied.

The TCP_STREAM results for the 2.5.3 base kernel and the 2.4.17 kernel with PROCESS affinity are comparable. Again this is probably attributable to the fact that the 2.5's 0(1)scheduler achieves better process affinity to CPU than the 2.4.17 kernel. The best performed test run in figure 3 is the 2.4.17 kernel with both Affinities applied.

The throughput for the results presented in the Figure 3 is mostly constant for most of the tests, and the difference reflected in CPU idle time.

2.2 Analysis of IRQ and PROCESS Affinity

L2_CACHE_LINES_OUT

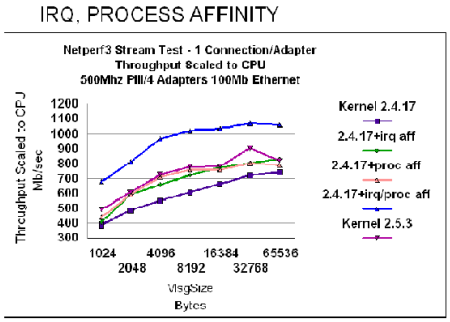

Figure 3: IRQ and PROCESS AFFINITY

Next, we analyzed the 4P TCP_STREAM test to understand why IRQ and PROCESS affinity together improves throughput and CPU utilization to a great extent (up to 74%). We used profiling based on Intel Performance counters [4]. We profiled the server during the TCP_STREAM test using the 32K message size on 2.4.17 kernel using the L2_cache_lines_out and total instructions retired events.

By using IRQ and PROCESS affinity, each TCPIPconnection and its activities are bound to happen in the affinitized CPU. This binding leads to lower L2 cache lines out as the data and instructions do not float between CPUs. The gain we achieved through affinity would reflect in this event counter. So we decided to measure L2_cache_lines_out. Because the throughput and throughput scaled to CPU increased in the IRQ and PROCESS affinity case, we decided to measure the total instruction retired count also.

In Table 2 we show the results of the performance counter profiling taken for the event "L2_cache_lines_out" on base Kernel 2.4.17 and Kernel 2.4.17+IRQ and PROCESS affin-

Kernel function	2.4.17 Kernel Baseline Frequency	IRQPROCESS AFFINITIES Frequency
poll_idle	121743	72241
csum_partial_copy_generic	27951	13838
schedule	24853	9036
do_softirq	9130	3922
mod_timer	6997	1551
TCP_v4_rcv	6449	629
speedo_interrupt	6262	5066
__wake_up	6143	1779
TCP_recvmsg	5199	2154
USER	5081	2573
speedo_start_xmit	4349	1654
TCP_rcv_established	3724	1336
TCP_data_wait	3610	748

Table 2: L2 Cache Lines Out on 2.4.17

ity. The kernel routines with the highest L2_cache_line_out are listed here. The results in Table 2 show that using IRQ and PROCESS affinity has reduced the number of L2 cache lines out in all of the listed kernel routines. The kernel routines tcp_v4_rcv, mod_timer, and tcp_data_wait are the major benefectors of affinity. Overall the whole of TCPIP receive code path has less L2_cache_lines_out resulting in better performance.

This profile is taken using TCP_STREAM test on 4P server and 8P client. The test ran on 4 adapters using 4 server processes with the configuration of 64K socket buffer, TCP NODELAY ON using 32k message size on 2.4.17 kernel:

INSTRUCTIONS RETIRED

Next we measured the total instructions retired for the same workload on both baseline 2.4.17 kernel and 2.4.17+IRQ+PROCESS affinity applied kernel. Figure 3 shows the total instructions retired for a short period of time running the TCP_STREAM 4P/4adapter test.

Kernel Function	Instruc. Retired Frequency
2.4.17 Kernel base	32554892
poll_idle on base	19877569
2.4.17 +IRQ+PROCESS	51607249
poll_idle on IRQ+PROCESS	39326958

Table 3: Instructions Retired Count on 2.4.17

The number of instructions executed is high in the affinity case which also supports the fact that lining up process/irq with a CPU brings memory locality and improves instruction memory references. We also presented the number of instructions retired in idle loop. The instructions retired in the idle-loop is doubled in the affinity case. We gained CPU in Affinity case as the memory latency for instructions is decreased. The number of instructions retired excluding the idle-loop case has not improved in the affinity case.

The above analysis clearly indicates that the TCPIP stack SMP scalability can be improved

by fixing the inter-processor cache line bouncing by reducing L2_cache_lines_out.

2.3 Combine_csum_copy Patch to reduce the cache_lines_out

Affinitizing both the IRQ and PROCESS to a CPU results in better locality of data and instructions for the TCPIP send and receive path and thus better performance. Because affinity is not feasible in all situations, we analyzed the code to determine if there are code optimizations that could provide better cache effectiveness. It was observed that most of the time, the incoming frames were checksummed in the interrupt context and then copied to the applicatoin buffer in the process context. Often, the interrupt and process context were on two different CPUs. A proto-type patch, csum_copy_patch, was developed to force the checksum and copy operations to execute more often in the process context.

Figure 4 shows the results of the TCP_STREAM test for 1) 2.4.0 kernel baseline, 2) 2.4.0 +IRQ and PROCESS affinity and 3) 2.4.0+csum_copy_patch. The csum_copy_patch improved throughput scaled to CPU utilization by up to 14%. There is additional work to be done in order to bridge the gap between the baseline and the IRQ and PROCESS affinity case. We will continue our work to see how we could close the gap between the non-affinity and affinity case through code improvement.

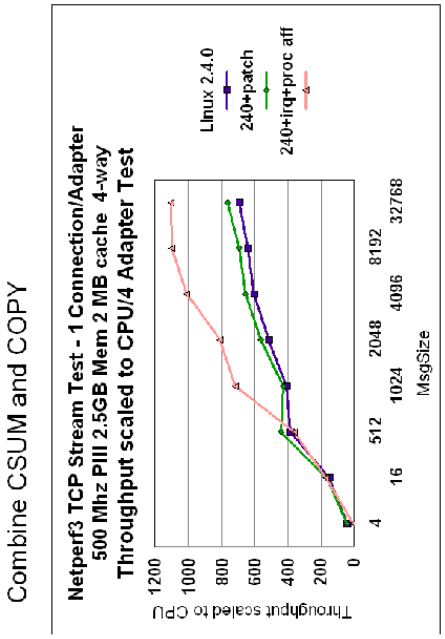

Figure 4: Combine CSUM and COPY on 2.4.0

3 Copy Routines in TCPIP

3.1 One gigabit NIC's TCP_STREAM Results

We measured the Netperf3 TCP_STREAM throughput for a single connection over a gigabit ethernet network using a 2.4.17 UNI kernel. The Netperf3 message size was 4096 bytes, MTU size was 1500 bytes, and the server (receive side) was an 8-way processor

Kernel function	Frequency
__generic_copy_to_user:	3127
e1000_intr:	540
alloc_skb :	313
TOTAL_SAMPLES	6001

Table 4: Gigabit PC Sampling of 2.4.17 UNI Kernel

700 Mhz PIII using a UNI kernel. We observed that the resulting throughput did not achieve maximum media throughput. The CPU was 19% utilized. A time based profile of the kernel revealed that 30–50% of the total ticks were spent in __generic_copy_to_user routine in the receive path of the TCPIP code. The __generic_copy_to_user routine is used to copy data from the network buffers to the application buffers. The profiling data for the TCPIP receive path is given in Table 4

3.2 Copy Routine Analysis

We analyzed the __generic_copy_to_user routine looking for ways to improve the code. The copy code was using the move string (MOVSD) instruction and had no special case code for handling mis-aligned data buffers. We looked at some of the fast memory copy work done previously and in particular the work done by University of Berkeley during the P5 time frame. The Berkeley study [9] compares three types of copies

- STRING: MOVSD string copy.
- INTEGER: unrolled/prefetched using integer registers.
- FLOATING POINT: unrolled/prefetched using floating point registers.

According to their results on P5 machines, the floating point copy method yielded 100% more throughput when compared to the string copy method. The integer copy method yielded 50% better throughput than the string copy method. We adopted both integer copy and floating point copy methods from this technical paper for improving the copy routines in the TCPIP stack.

We developed a user level tool to test these copy methods and found that the integer copy performs better than the other two methods if the source and destination buffers are not aligned on 8-byte boundaries. As shown in Table 5, if the source and the destination buffers are aligned on a 8-byte boundary, the string copy performed better than the “unrolled integer copy.” We used a Pentium III system for this test and each test copied 1448 million bytes. In Table 5, the MBytes copied is the throughput and higher the number the copy method is more efficient.

3.3 CopyPatch for Efficient Copy

We created a copy patch using “unrolled integer copy” for the Linux Kernel and tested further with and without alignment to further understand the impact of this patch and buffer alignment on our workload. We decided to test the alignment in the receive path of the TCPIP stack. The gigabit driver was modified to align the receive buffer (which is the source buffer for the copy routine). The destination buffer allocated by netperf3 was already aligned. We instrumented __generic_copy_to_user to measure the CPU cycles spent in this routine. We read the Intel’s TSC counter before and after execution of the copy routine with HW interrupts disabled. A user level program was written to retrieve the value of the cycle counter during the test run several times and also after the test is completed.

The results showed in Table 6 are the average cycles (rounded) spent to copy a buffer size of 1448 with and without the patch and with alignment. Lower the cycles better the performance of the copy routine.

The data in Table 6 suggests that the MOVSD instruction has the best performance when the

Method	time taken (sec)	MBytes copied	dst address	src address	aligned
MOVSD	0.609	2378	804c000	804f000	YES
Integer	0.898	1613	8051000	8053000	YES
MOVSD	1.172	1235	804c000	804f004	NO
Integer	0.851	1703	8051000	8053004	NO

Table 5: Comparison of movsd, integer copy, alignment

Method Used	Cycles Spent
movsd copy routine without alignment	7000
movsd copy routine with 8 byte alignment	3000
IntegerCopyPatch without alignment	4000

Table 6: Measurement of cycles spent in copy methods on 2.4.17 Kernel

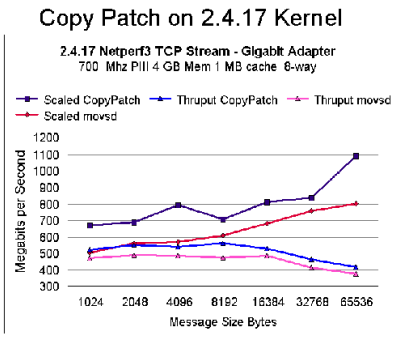

Figure 5: Copy Patch on 2.4.17 Kernel

source and the destination buffer addresses are aligned on an 8-byte boundary. However, an 8-byte source and destination alignment may not be possible in the receive path of the TCPIP stack for all general purposes and in all hetrogenous networks. The TCPIP frame header size is variable due to the TCP and IP options in the header. For our analysis purpose we were able to align this as we had a controlled and isolated homogenous network environment. So we decided to implement the "unrolled integer copy" replacing the "movsd" string copy in the copy routines used in this workload as aligning the buffers is out of question in the TCPIP receive path.

Figure 5 and Figure 6 show the baseline and CopyPatch throughput and "throughput scaled to CPU" results on 2.4.17 and 2.5.7 kernels respectively. It is obvious that the unrolled integer copy routine has improved throughput scaled to CPU for all the message sizes on both kernels. On 2.4.17, the raw throughput improved for all message sizes with the copypatch, however on 2.5.7 kernel, copypatch improved raw throughput on messages greater than 8k.

There is other copy routines combined with checksum in the TCPIP stack, we have not modified those routines yet. See appendix for integer copy patch. Since there are other methods such as sendfile (only for sendside) and mmx copies that are applicable to cover some situations, the scope of this work may look limited but this kind of string copy is used in other places of the kernel, glibc etc., So we think this work is important to improve the Linux performance in IA32 architecture and for the future "string copy instruction - movsd" implementation in IA32 architecture.

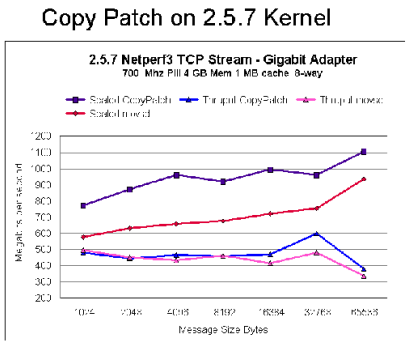

Figure 6: Copy Patch on 2.5.7 Kernel

3.4 Future Work

As part of our future work we will look in to improving the following:

- Extend this work to other memcopy routines in the kernel.
- Extend this work to glibc routines.

4 TCPIP GIGABIT NIC SCALABILITY

4.1 Gigabit NIC Scalability Results

The gigabit Ethernet NIC scalability test measures how well multiple gigabit Ethernet NICs perform on an 8-way server. We measured Gigabit Ethernet NIC scalability on the 2.4.7 kernel using the Netperf3 multi-adapter TCP_STREAM test. The server used was an 8-way 700 Mhz Pentium III Xeon CPUs with up to seven Gigabit Ethernet NICs operating in full duplex mode with a MTU size of 1500 bytes. Four 2-way 900 Mhz clients (send side) with two Gigabit Ethernet NICs on each connected to the server (receive side) Ethernet NICs.

The test was first run with only one gigabit NIC, then two NICs, and so on, up to a total of seven gigabit Ethernet NICs. A single TCP connection was created between the server and the client on each of the Ethernet NICs. The test was run with application message sizes of 1024, 2048, 4096, 8196, 16384, 32768 and 65536. The TCP socket buffer size was set to 64K with TCP NODELAY ON. The Ethernet NICs default options are used for the configuration parameters; although, tuning these options did not yield any better results on our hardware. We used SMP kernels enabling 8 processors on the server and 2 processors on each of the client.

The throughput results of the NIC scalability test are found in Figure 7. A single Ethernet NIC achieved only 604 Mb per second. Furthermore, adding a second Ethernet NIC achieved only a total of 699Mb per second for the pair of NICs (yielding a scaling factor of only 58% = 699/604*2). Adding successive Ethernet NIC added minimal throughput. The results of the gigabit Ethernet NIC scalability test inidcates that the gigabit Ethernet NICs tests do not scale on this 8-way system.

4.2 Analysis of TCPIP Scalability on Gigabit Network

We profiled the kernel to understand what could be done in the software to improve the gigabit NICs Scalability. From the TCPIP stack and kernel perspective, our analysis points out that the network stack does not efficiently handle the high rate of incoming frames. One of the main problems we noticed was that the high

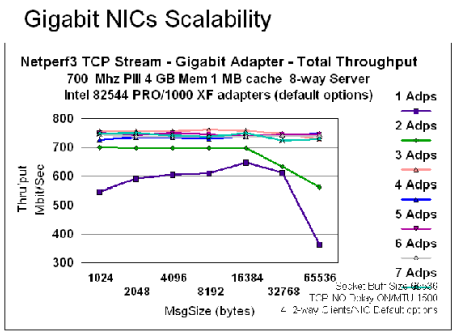

Figure 7: Gigabit NIC Scalability on 2.4.7 Kernel

rate of incoming frames was being processed at the protocol level one at a time even though the NIC mitigates interrupts and causes interrupt once per configurable interrupt delay. Therefore the NIC causes interrupts for a bunch of frames instead of interrupting the processor for each frame. This interrupt mitigation is not mimicked in higher layer protocol processing.

Kernprof time based annotated call graph profiling taken with MTU=1500 and MTU=9000

Kernel Function	Times invoked
e1000_intr	275543
. ProcessReceiveInterrupts	312522
. . netif_rx	2291591
. . . get_fast_time	2291591
. . . . do_gettimeofday	2291591
. . . get_sample_stats	2291591
. . . cpu_raise_softirq	2291591
. . eth_type_trans	2291591
. . RxChecksum	2291591
. . _tasklet_schedule	32369
. ProcessTransmitInterrupts	312522
. . cpu_raise_softirq	189394

Table 7: 2.4.7 Kernel Annotated Callgraph for MTU 1500

By setting the MTU=9000 (jumbo size), on a gigabit adapter/TCP_STREAM, we could reach the max media limit, whereas with MTU set to 1500, we did not reach the maximum media limit on our hardware. We used profiling tools to see the difference. Tables 7 and 8 show the annotated call graph for a three adapter case using 1500 and 9000 MTU sizes. In MTU=1500 case we received less interrupts around 275543 times. But the softirq handler was invoked around 2 million times. Therefore, we received around 8 frames per interrupt in an average. Whereas in the MTU=9000 case, we received 860970 time interrupts and softirq handler was invoked for 758774 times. We received one frame per interrupt on an average. Therefore, the upper layer protocols are not invoked more than the interrupts and the protocol processing latency was less so we received 3X more interrupts/frames in case of jumbo size frame. Therefore, we suspect that the higher layer protocol processing latency is causing the throughput to go down in the case of MTU=1500 as each layer of the protocol is processing one frame at a time. We propose using a mechanism such as “gather-receive” where the bunch of the frames received are gathered and sent to upper layer protocol for processing. We have not done any proto-type yet to prove that this will help.

Kernel Function	Times invoked
ReceiveBufferFill	32369
. alloc_skb	2291645
. . kmalloc	2481043
. . . kmem_cache_alloc_batch	63866
. . kmem_cache_alloc	1178633
. . . kmem_cache_alloc_batch	7110

Table 9: 2.4.7 Kernel’s Kernprof Annotated Callgraph for MTU 1500

We also found a similar problem with the allocation and deallocation of skbs (socket buffers) and data frame buffers in the receive path. The Linux gigabit network device driver has a re-

Kernel function	Times Invoked
e1000_intr	840790
. ProcessReceiveInterrupts	951848
. . netif_rx	758774
. . . get_fast_time	758774
. . . . do_gettimeofday	758774
. . . get_sample_stats	758774
. . . cpu_raise_softirq	758774
. . eth_type_trans	758774
. . RxChecksum	758774
. . _tasklet_schedule	12339
. . . wake_up_process	3
. . . . reschedule_idle	3367
. ProcessTransmitInterrupts	951848
. . cpu_raise_softirq	368286

Table 8: 2.4.7 Kernel Kernprof's Annotated callgraph for MTU 9000

Kernel function	Times Invoked
ReceiveBufferFill	12339
. alloc_skb	758748
. . kmalloc	1127037
. . . kmem_cache_alloc_batch	170
. . kmem_cache_alloc	178716
. . . kmem_cache_alloc_batch	792

Table 10: 2.4.7 Kernel Kernprof's Annotated callgraph for MTU 9000

ceive pool of buffers called "receive ring." Gigabit driver replenishes this pool by allocating skb one at a time. Tables 9 and 10 show that alloc_skb is called 2 million times when ReceiveBufferFill is called only 32 thousand times in the case where MTU was set to 1500. ReceiveBufferFill is the routine that replenishes the receive buffer pool. TCPIP stack should provide a way to allocate these skbs bunch at a time. To take this one step further, we think that the allocated buffers (receive ring in the driver) should be recycled instead of freeing and reallocating them again. We have started looking into creating a proto-type patch and this is our on-going effort.

4.3 Analysis of Hardware for Gigabit NIC Scalability

We performed additional tests in order to understand why multiple NICs do not scale well on this 8-way system. We first investigated the PCI bus. The 8-way server has two 66 Mhz 64-bit PCI buses (A and B) and two 33 Mhz 64-bit PCI buses (C and D). We used gigabit Ethernet NICs on different combinations of PCI busses and reran the tests.

Tables 4.3 and 4.3 indicate that when two gigabit Ethernet NICs are used, placing one on bus A and the other on bus B improved throughput by 10% compared to having both adapters on

Table 11: Two NIC using separate PCI buses

Bus		Throughput
NIC 1	Bus A (66 MHz)	385 Mb/sec
NIC 2	Bus B (66 MHz)	387 Mb/sec
Total		782 Mb/sec

Table 12: Two NIC sharing the PCI bus

Bus		Throughput
NIC 1	Bus A (66 MHz)	355 Mb/sec
NIC 2	Bus A (66 MHz)	342 Mb/sec
Total		697 Mb/sec

bus A.

Running three adapters on three different buses yields almost the same throughput as the case where 3 NICs share the same 66 Mhz bus. The PCI bus capacity is 532 MByte per second on 66 Mhz and 266 MByte/sec on 33 Mhz [5] But we are not even getting 800 Mbits/sec total in the above cases. We are far from hitting the PCI bus capacity; therefore, we concluded that the PCI bus is not the bottleneck.

Next we looked at the effects of IRQ and Process affinity on the workload. We tested affinity on 2 NICs. Since we have 8 CPUs on our server system we chose different combinations of CPUs to see if selecting one CPU from each

Table 13: Three Adapters, Separate vs. Shared

Three adapters using separate PCI bus		
NIC	Bus	Throughput
1	Bus A (66 MHz)	266 Mb/sec
2	Bus B (66 MHz)	257 Mb/sec
3	Bus C (33 MHz)	259 Mb/sec
Total		782 Mb/sec
Three adapters sharing PCI bus		
1	Bus A (66 MHz)	197 Mb/sec
2	Bus A (66 MHz)	197 Mb/sec
3	Bus B (66 MHz)	356 Mb/sec
Total		750 Mb/sec

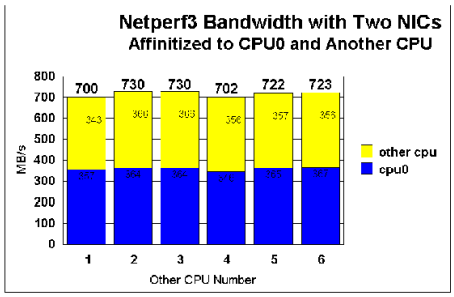

Figure 8: Results of IRQ and Process Affinity with 2 NICs

side of the Profusion chip set would make a difference. (The 8-way is composed of two 4-ways with independent 800 MBytes/sec front side buses, connected by the Profusion chip to 2 memory cards providing 1600 MBytes/sec and to the PCI buses via a 800 MByte/sec connection.) We affinitize Netperf3 server processes to a combination of CPU sets and the Ethernet NIC's IRQs to a combination of CPU sets. The system is not rebooted between the tests, so the numbers assigned to CPUs by the operating system stayed intact between the tests. The server process was restarted before each test and the IRQs for the NICs were reset after each test. Both the NICs were placed in Bus A (a 66 Mhz/64 bit bus) and interrupts 23 and 24 were assigned to these two NICs. Interrupt 23 was bounded to CPU0 for all the tests and interrupt 24 was affinitized to CPU1, CPU2 etc. Netperf's server process 1 is always affinitized to CPU0 and process 2 is bound to CPU 1, CPU 2 and so on. The results of the IRQ and PROCESS affinity test are in Figure 8. The throughput has not improved that much compared to the baseline. The max throughput that we get with both affinity is around 730 Mbits/sec and our baseline is 699 Mbits/sec.

4.4 Future Work

Neither the IRQ and PROCESS affinity nor selecting CPUs from different sides of the bus improved the throughput much. These 2 and 3 adapter cases are neither CPU bound nor network media limited. So we must contnue our analysis to find the bottleneck. We also did some preliminary investigation using Intel Performance counter profiling; but, found no conclusive leads. We will continue this work further. However, we have shown that the PCI bus is not the limiting factor, and IRQ and PROCESS affinity do not help improve the throughput and scalability. It is not acceptable that even one adapter does not achieve close to media speed using MTU 1500 size and that adding NICs do not scale well. Gigabit NIC scalability will be a focus of our future investigation.

5 Concluding Remarks

In this paper we have highlighted a few potential areas for improvement in the Linux TCPIP protocol.

- SMP Network Scalability: We presented results showing the SMP network scalability problem and provided analysis to associate the poor scalability to inter-processor cache line bouncing due to high L2_cache_lines_out problem. We believe that this SMP network scalability is one of the areas where TCPIP stack needs to be fixed to improve scalability. We also showed a proto-type to improve the data cache reference in the TCPIP stack. Additionally we examined efficient copy routines for the IA32 Linux TCPIP stack and expressed the belief that this may be a potential area to further investigate to gain performance improvement in the IA32 kernel itself and glibc routines.

- TCPIP Scalability on Gigabit: We examined the effects of using gigabit network on the LINUX TCPIP stack. We presented various test results and analysis data to show that the hardware that we used is not good enough to handle more than 2 NICs. We also emphasized that the implementation of the Linux TCPIP stack itself needs modifications to handle high bandwidth network traffic. As we move to other types of high bandwidth networks such as InfiniBand, we need to keep in mind that the software network stack should also need to scale to utilize fully the high network bandwidth. We conclude that both the system hardware and the software need improvement to take advantage of the high network bandwidth.

We will be working on fixing the above mentioned problems. We look forward to working with the members of the Linux community to discuss, design, and implement solutions to improve the LINUX TCPIP SMP scalability and gigabit network scalability.

6 Acknowledgments

The authors would like to acknowledge the assistance of Fadi Sibai of Intel in interpreting the Pentium Performance counter data. We would like to acknowledge Bill Brantley of IBM and Patricia Goubil-Gambrell of IBM for improving the quality of the paper. We would also like to thank Nivedita Singhvi of IBM and Bruce Alan of IBM for sharing their gigabit NICs scalability test results. This helped us to verify our results.

7 About the authors

Vaijayanthimala (Mala) K Anand works in the IBM Linux Technology center as a member of the Linux Performance team. Mala has worked on the design and development of network protocols, network device drivers and thin clients. Mala is currently working on Linux TCPIP stack performance analysis. She can be reached at manand@us.ibm.com.

Bill Hartner is the technical lead for IBM's Linux Technology Center performance team. Bill has worked in software development for 18 years. For the past 8 years, Bill has worked in kernel development and performance. For the past 3 years Bill has worked on Linux kernel performance. Bill can be reached at bhartner@us.ibm.com.

References

[1] Jeff Chase Andrew Gallatin and Ken Yocum. Trapeze IP:TCPIP at Near Gigabit speeds. http://www.cs.duke.edu/ari/trapeze /freenix/paper.html.

[2] Trevor Blackwell. Speeding up protocols of small messages. ACM SIGCOMM Symposium on Communications Architectures and Protocols,Aug 1996.

[3] R. Bryant and J. Hawkes. Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel. In *Proc. Fourth Annual Linux Showcase and Conference, Atlanta*, Oct 2000.

[4] Intel Corp. Intel Architecture Software Developer's Manual Volume 3: Sysytem Programming. http://www.intel.com.

[5] Adaptec corp's White Paper. PCI, 64-Bit and 66 MHz Benefits. http://www.adaptec.com/worldwide /product/markeditorial.html.

[6] Jim Jurose Erich Nahum, David Yates and Don Towsleyr. Cache Behavior of Network Protocols, June 1997. http://cs-www.bu.edu/faculty/djy.

[7] John Hawkes et. al (Silicon Graphics Inc.). Kernprof. Available at http://oss.sgi.com/projects/kernprof /index.html.

[8] Ingo Molnar. 0(1) scheduler patch. http://www.kernel.org/pub/linux/kernel/ people/mingo.

[9] University of Berkeley. Fast Memory Copy. http://now.cs.berkeley.edu/ Td/bcopy.html.

[10] Hewlett Packard Inc. Rick Jones. Network Benchmarking Netperf. http://www.netperf.org.

8 Appendix

8.1 COPY Patch

This patch changes the copy routines used in tcpip stack.

```
diff -Naur linux-417/arch/i386/lib/usercopy.c \
           linux-417a/arch/i386/lib/usercopy.c
--- linux-417/arch/i386/lib/usercopy.c  Tue Jan 22 21:29:05 2002
+++ linux-417a/arch/i386/lib/usercopy.c Fri Jan 18 12:50:38 2002
@@ -44,7 +44,6 @@
 unsigned long
 __generic_copy_to_user(void *to, const void *from, unsigned long n)
 {
-       prefetch(from);
        if (access_ok(VERIFY_WRITE, to, n))
                __copy_user(to,from,n);
        return n;
diff -Naur linux-417/include/asm-i386/string.h \
           linux-417a/include/asm-i386/string.h
--- linux-417/include/asm-i386/string.h Tue Jan 22 21:29:48 2002
+++ linux-417a/include/asm-i386/string.h        Wed Jan 23 00:11:29 2002
@@ -196,21 +196,65 @@
 return __res;
 }

-static inline void * __memcpy(void * to, const void * from, size_t n)
+static inline void * __memcpy(void * to, const void * from, size_t  size)
 {
-int d0, d1, d2;
+  int __d0, __d1;
 __asm__ __volatile__(
-       "rep ; movsl\n\t"
-       "testb $2,%b4\n\t"
-       "je 1f\n\t"
-       "movsw\n"
-       "1:\ttestb $1,%b4\n\t"
-       "je 2f\n\t"
-       "movsb\n"
-       "2:"
-       : "=&c" (d0), "=&D" (d1), "=&S" (d2)
-       :"0" (n/4), "q" (n),"1" ((long) to),"2" ((long) from)
-       : "memory");
+                "       cmpl $63, %0\n\t"
+                "       jbe  2f\n\t"
+                "       .align 2, 0x90\n\t"
+                "0:     movl 32(%5), %%eax\n\t"
+                "       cmpl $67, %0\n\t"
+                "       jbe 1f\n\t"
+                "       movl 64(%5), %%eax\n\t"
```

```
+                "       .align 2, 0x90\n\t"
+               "1:     movl 0(%5), %%eax\n\t"
+                "       movl 4(%5), %%edx\n\t"
+                "       movl %%eax, 0(%4)\n\t"
+               "       movl %%edx, 4(%4)\n\t"
+               "       movl 8(%5), %%eax\n\t"
+               "       movl 12(%5),%%edx\n\t"
+               "       movl %%eax, 8(%4)\n\t"
+               "       movl %%edx, 12(%4)\n\t"
+               "       movl 16(%5), %%eax\n\t"
+               "       movl 20(%5), %%edx\n\t"
+                "       movl %%eax, 16(%4)\n\t"
+                "       movl %%edx, 20(%4)\n\t"
+                "       movl 24(%5), %%eax\n\t"
+                "       movl 28(%5), %%edx\n\t"
+                "       movl %%eax, 24(%4)\n\t"
+                "       movl %%edx, 28(%4)\n\t"
+                "       movl 32(%5), %%eax\n\t"
+                "       movl 36(%5), %%edx\n\t"
+                "       movl %%eax, 32(%4)\n\t"
+                "       movl %%edx, 36(%4)\n\t"
+                "       movl 40(%5), %%eax\n\t"
+                "       movl 44(%5), %%edx\n\t"
+                "       movl %%eax, 40(%4)\n\t"
+                "       movl %%edx, 44(%4)\n\t"
+                "       movl 48(%5), %%eax\n\t"
+                "       movl 52(%5), %%edx\n\t"
+                "       movl %%eax, 48(%4)\n\t"
+                "       movl %%edx, 52(%4)\n\t"
+                "       movl 56(%5), %%eax\n\t"
+                "       movl 60(%5), %%edx\n\t"
+                "       movl %%eax, 56(%4)\n\t"
+                "       movl %%edx, 60(%4)\n\t"
+                "       addl $-64, %0\n\t"
+                "       addl $64, %5\n\t"
+                "       addl $64, %4\n\t"
+                "       cmpl $63, %0\n\t"
+                "       ja  0b\n\t"
+               "2:     movl  %0, %%eax\n\t"
+                "       shrl  $2, %0\n\t"
+                "       andl  $3, %%eax\n\t"
+                "       cld\n\t"
+                "       rep; movsl\n\t"
+                "       movl %%eax, %0\n\t"
+               "       rep; movsb\n\t"
+               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)
+               :  "0"(size), "1"(to), "2"(from)
+               : "eax", "edx","memory");
 return (to);
 }

diff -Naur linux-417/include/asm-i386/uaccess.h \
           linux-417a/include/asm-i386/uaccess.h
--- linux-417/include/asm-i386/uaccess.h        Tue Jan 22 21:29:43 2002
```

```
+++ linux-417a/include/asm-i386/uaccess.h      Tue Jan 22 21:58:11 2002
@@ -256,50 +256,186 @@
 do {                                                                    \
        int __d0, __d1;                                                 \
        __asm__ __volatile__(                                           \
-               "0:     rep; movsl\n"                                   \
-               "       movl %3,%0\n"                                   \
-               "1:     rep; movsb\n"                                   \
-               "2:\n"                                                  \
-               ".section .fixup,\"ax\"\n"                              \
-               "3:     lea 0(%3,%0,4),%0\n"                            \
-               "       jmp 2b\n"                                       \
-               ".previous\n"                                           \
-               ".section __ex_table,\"a\"\n"                           \
-               "       .align 4\n"                                     \
-               "       .long 0b,3b\n"                                  \
-               "       .long 1b,2b\n"                                  \
-               ".previous"                                             \
-               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)               \
-               : "r"(size & 3), "0"(size / 4), "1"(to), "2"(from)      \
-               : "memory");                                            \
+               "       cmpl $63, %0\n"                                 \
+               "       jbe  5f\n"                                      \
+               "       .align 2,0x90\n"                                \
+               "0:     movl 32(%4), %%eax\n"                           \
+               "       cmpl $67, %0\n"                                 \
+               "       jbe 1f\n"                                       \
+               "       movl 64(%4), %%eax\n"                           \
+               "       .align 2,0x90\n"                                \
+               "1:     movl 0(%4), %%eax\n"                            \
+               "       movl 4(%4), %%edx\n"                            \
+               "2:     movl %%eax, 0(%3)\n"                            \
+               "21:    movl %%edx, 4(%3)\n"                            \
+               "       movl 8(%4), %%eax\n"                            \
+               "       movl 12(%4),%%edx\n"                            \
+               "3:     movl %%eax, 8(%3)\n"                            \
+               "31:    movl %%edx, 12(%3)\n"                           \
+               "       movl 16(%4), %%eax\n"                           \
+               "       movl 20(%4), %%edx\n"                           \
+               "4:     movl %%eax, 16(%3)\n"                           \
+               "41:    movl %%edx, 20(%3)\n"                           \
+               "       movl 24(%4), %%eax\n"                           \
+               "       movl 28(%4), %%edx\n"                           \
+               "10:    movl %%eax, 24(%3)\n"                           \
+               "51:    movl %%edx, 28(%3)\n"                           \
+               "       movl 32(%4), %%eax\n"                           \
+               "       movl 36(%4), %%edx\n"                           \
+               "11:    movl %%eax, 32(%3)\n"                           \
+               "61:    movl %%edx, 36(%3)\n"                           \
+               "       movl 40(%4), %%eax\n"                           \
+               "       movl 44(%4), %%edx\n"                           \
+               "12:    movl %%eax, 40(%3)\n"                           \
+               "71:    movl %%edx, 44(%3)\n"                           \
+               "       movl 48(%4), %%eax\n"                           \
```

```
+                "       movl 52(%4), %%edx\n"                          \
+                "13:    movl %%eax, 48(%3)\n"                          \
+                "81:    movl %%edx, 52(%3)\n"                          \
+                "       movl 56(%4), %%eax\n"                          \
+                "       movl 60(%4), %%edx\n"                          \
+                "14:    movl %%eax, 56(%3)\n"                          \
+                "91:    movl %%edx, 60(%3)\n"                          \
+                "       addl $-64, %0\n"                               \
+                "       addl $64, %4\n"                                \
+                "       addl $64, %3\n"                                \
+                "       cmpl $63, %0\n"                                \
+                "       ja  0b\n"                                      \
+               "5:     movl  %0, %%eax\n"                             \
+                "       shrl  $2, %0\n"                                \
+                "       andl  $3, %%eax\n"                             \
+                "       cld\n"                                         \
+                "6:     rep; movsl\n"                                  \
+                "       movl %%eax, %0\n"                              \
+               "7:     rep; movsb\n"                                  \
+               "8:\n"                                                 \
+               ".section .fixup,\"ax\"\n"                             \
+               "9:     lea 0(%%eax,%0,4),%0\n"                        \
+               "       jmp 8b\n"                                      \
+                "15:    movl %6, %0\n"                                \
+                "       jmp 8b\n"                                      \
+               ".previous\n"                                          \
+               ".section __ex_table,\"a\"\n"                          \
+               "       .align 4\n"                                    \
+               "       .long 2b,15b\n"                                \
+               "       .long 21b,15b\n"                               \
+               "       .long 3b,15b\n"                                \
+               "       .long 31b,15b\n"                               \
+               "       .long 4b,15b\n"                                \
+               "       .long 41b,15b\n"                               \
+               "       .long 10b,15b\n"                               \
+               "       .long 51b,15b\n"                               \
+               "       .long 11b,15b\n"                               \
+               "       .long 61b,15b\n"                               \
+               "       .long 12b,15b\n"                               \
+               "       .long 71b,15b\n"                               \
+               "       .long 13b,15b\n"                               \
+               "       .long 81b,15b\n"                               \
+               "       .long 14b,15b\n"                               \
+               "       .long 91b,15b\n"                               \
+               "       .long 6b,9b\n"                                 \
+                "       .long 7b,8b\n"                                \
+               ".previous"                                            \
+               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)               \
+               :  "1"(to), "2"(from), "0"(size),"i"(-EFAULT)           \
+               : "eax", "edx", "memory");                             \
 } while (0)

 #define __copy_user_zeroing(to,from,size)                               \
 do {                                                                    \
```

```
        int __d0, __d1;                                                 \
        __asm__ __volatile__(                                           \
-               "0:     rep; movsl\n"                                   \
-               "       movl %3,%0\n"                                   \
-               "1:     rep; movsb\n"                                   \
-               "2:\n"                                                  \
-               ".section .fixup,\"ax\"\n"                              \
-               "3:     lea 0(%3,%0,4),%0\n"                            \
-               "4:     pushl %0\n"                                     \
-               "       pushl %%eax\n"                                  \
-               "       xorl %%eax,%%eax\n"                             \
-               "       rep; stosb\n"                                   \
-               "       popl %%eax\n"                                   \
-               "       popl %0\n"                                      \
-               "       jmp 2b\n"                                       \
-               ".previous\n"                                           \
-               ".section __ex_table,\"a\"\n"                           \
-               "       .align 4\n"                                     \
-               "       .long 0b,3b\n"                                  \
-               "       .long 1b,4b\n"                                  \
-               ".previous"                                             \
-               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)               \
-               : "r"(size & 3), "0"(size / 4), "1"(to), "2"(from)      \
-               : "memory");                                            \
+                "       cmpl $63, %0\n"                                \
+                "       jbe  5f\n"                                     \
+                "       .align 2,0x90\n"                               \
+                "0:     movl 32(%4), %%eax\n"                          \
+               "       cmpl $67, %0\n"                                \
+                "       jbe 2f\n"                                      \
+                "1:     movl 64(%4), %%eax\n"                          \
+                "       .align 2,0x90\n"                               \
+               "2:     movl 0(%4), %%eax\n"                           \
+                "21:    movl 4(%4), %%edx\n"                           \
+                "       movl %%eax, 0(%3)\n"                           \
+               "       movl %%edx, 4(%3)\n"                           \
+               "3:     movl 8(%4), %%eax\n"                           \
+               "31:    movl 12(%4),%%edx\n"                           \
+               "       movl %%eax, 8(%3)\n"                           \
+               "       movl %%edx, 12(%3)\n"                          \
+               "4:     movl 16(%4), %%eax\n"                          \
+               "41:    movl 20(%4), %%edx\n"                          \
+                "       movl %%eax, 16(%3)\n"                          \
+                "       movl %%edx, 20(%3)\n"                          \
+                "10:    movl 24(%4), %%eax\n"                          \
+                "51:    movl 28(%4), %%edx\n"                          \
+                "       movl %%eax, 24(%3)\n"                          \
+                "       movl %%edx, 28(%3)\n"                          \
+                "11:    movl 32(%4), %%eax\n"                          \
+                "61:    movl 36(%4), %%edx\n"                          \
+                "       movl %%eax, 32(%3)\n"                          \
+                "       movl %%edx, 36(%3)\n"                          \
+                "12:    movl 40(%4), %%eax\n"                          \
+                "71:    movl 44(%4), %%edx\n"                          \
```

```
+                "       movl %%eax, 40(%3)\n"                          \
+                "       movl %%edx, 44(%3)\n"                          \
+                "13:    movl 48(%4), %%eax\n"                          \
+                "81:    movl 52(%4), %%edx\n"                          \
+                "       movl %%eax, 48(%3)\n"                          \
+                "       movl %%edx, 52(%3)\n"                          \
+                "14:    movl 56(%4), %%eax\n"                          \
+                "91:    movl 60(%4), %%edx\n"                          \
+                "       movl %%eax, 56(%3)\n"                          \
+                "       movl %%edx, 60(%3)\n"                          \
+                "       addl $-64, %0\n"                               \
+                "       addl $64, %4\n"                                \
+                "       addl $64, %3\n"                                \
+                "       cmpl $63, %0\n"                                \
+                "       ja  0b\n"                                      \
+               "5:     movl  %0, %%eax\n"                             \
+                "       shrl  $2, %0\n"                                \
+                "       andl $3, %%eax\n"                              \
+                "       cld\n"                                         \
+                "6:     rep; movsl\n"                                  \
+                "       movl %%eax,%0\n"                               \
+               "7:     rep; movsb\n"                                  \
+               "8:\n"                                                 \
+               ".section .fixup,\"ax\"\n"                             \
+               "9:     lea 0(%%eax,%0,4),%0\n"                        \
+               "16:    pushl %0\n"                                    \
+               "       pushl %%eax\n"                                 \
+               "       xorl %%eax,%%eax\n"                            \
+               "       rep; stosb\n"                                  \
+               "       popl %%eax\n"                                  \
+               "       popl %0\n"                                     \
+               "       jmp 8b\n"                                      \
+                "15:    movl %6, %0\n"                                 \
+                "       jmp 8b\n"                                      \
+               ".previous\n"                                          \
+               ".section __ex_table,\"a\"\n"                          \
+               "       .align 4\n"                                    \
+               "       .long 0b,16b\n"                                \
+               "       .long 1b,16b\n"                                \
+               "       .long 2b,16b\n"                                \
+               "       .long 21b,16b\n"                               \
+               "       .long 3b,16b\n"                                \
+               "       .long 31b,16b\n"                               \
+               "       .long 4b,16b\n"                                \
+               "       .long 41b,16b\n"                               \
+               "       .long 10b,16b\n"                               \
+               "       .long 51b,16b\n"                               \
+               "       .long 11b,16b\n"                               \
+               "       .long 61b,16b\n"                               \
+               "       .long 12b,16b\n"                               \
+               "       .long 71b,16b\n"                               \
+               "       .long 13b,16b\n"                               \
+               "       .long 81b,16b\n"                               \
+               "       .long 14b,16b\n"                               \
```

```
+               "       .long 91b,16b\n"                                \
+               "       .long 6b,9b\n"                                  \
+                "       .long 7b,16b\n"                                 \
+               ".previous"                                             \
+               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)               \
+               :  "1"(to), "2"(from), "0"(size),"i"(-EFAULT)           \
+               : "eax", "edx", "memory");                              \
 } while (0)

 /* We let the __ versions of copy_from/to_user inline, because they're often
@@ -577,24 +713,16 @@
 }

 #define copy_to_user(to,from,n)                                 \
-       (__builtin_constant_p(n) ?                       \
-        __constant_copy_to_user((to),(from),(n)) :      \
-        __generic_copy_to_user((to),(from),(n)))
+        __generic_copy_to_user((to),(from),(n))

 #define copy_from_user(to,from,n)                       \
-       (__builtin_constant_p(n) ?                       \
-        __constant_copy_from_user((to),(from),(n)) :    \
-        __generic_copy_from_user((to),(from),(n)))
+        __generic_copy_from_user((to),(from),(n))

 #define __copy_to_user(to,from,n)                       \
-       (__builtin_constant_p(n) ?                       \
-        __constant_copy_to_user_nocheck((to),(from),(n)) :      \
-        __generic_copy_to_user_nocheck((to),(from),(n)))
+        __generic_copy_to_user_nocheck((to),(from),(n))

 #define __copy_from_user(to,from,n)                     \
-       (__builtin_constant_p(n) ?                       \
-        __constant_copy_from_user_nocheck((to),(from),(n)) :    \
-        __generic_copy_from_user_nocheck((to),(from),(n)))
+        __generic_copy_from_user_nocheck((to),(from),(n))

 long strncpy_from_user(char *dst, const char *src, long count);
 long __strncpy_from_user(char *dst, const char *src, long count);
```