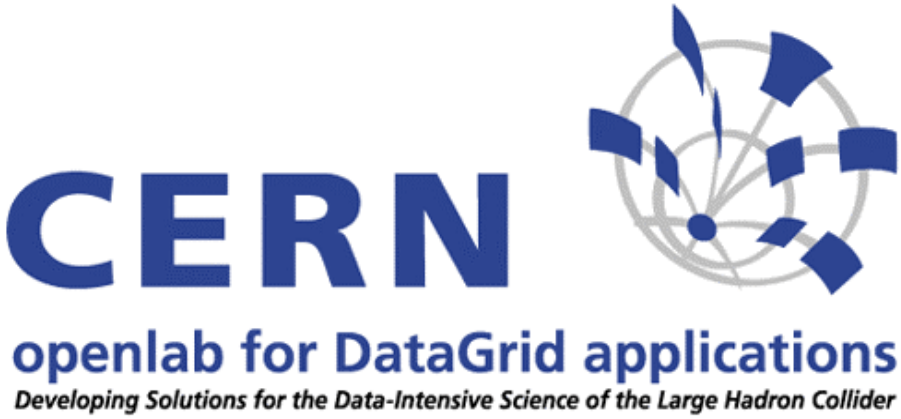


10Gb Ethernet Back-To-Back tests

Glenn Hisdal

June - August 2003

Table of Contents

1 Introduction

This document describes the 10Gb Ethernet back-to-back test done as part of the Opencluster project summer 2003. This document starts by describing the hardware and software systems used in the test, and then continues with the results from the various tests. The last section identifies some of the issues that appeared while these tests were done.

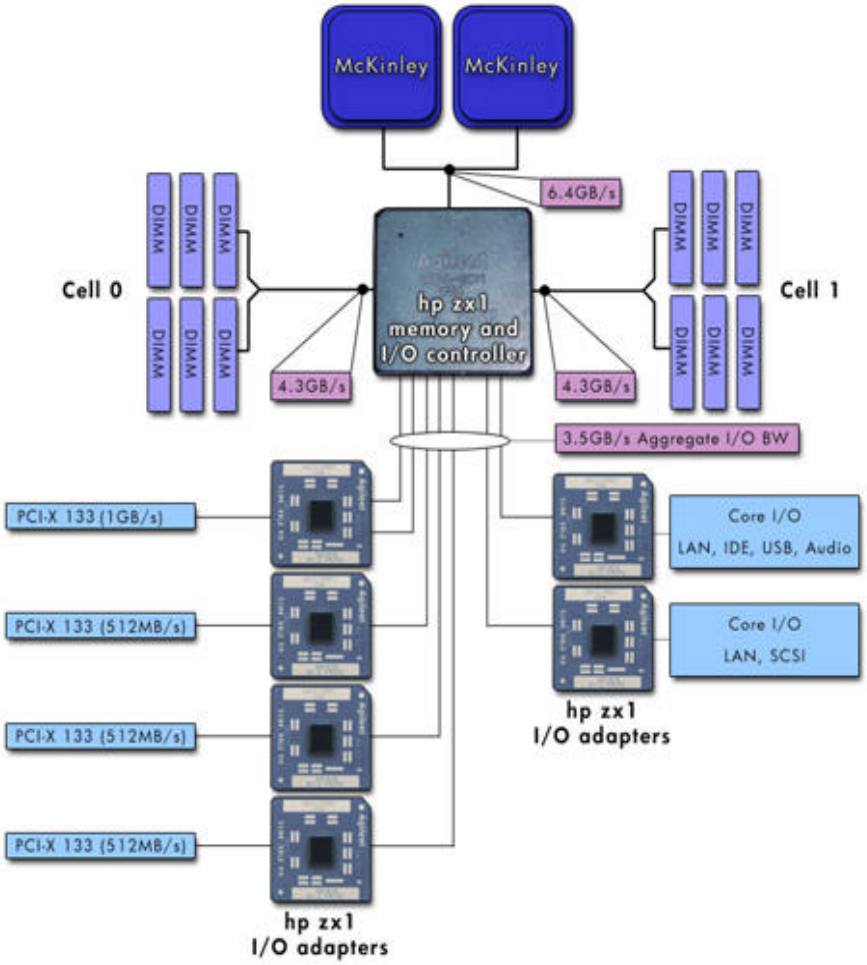
The goal of this project was to measure the throughput of a 10Gb Ethernet connection between two IA64 computers and between an IA32 computer and an IA64 computer, all of which were running Linux. In this document I refer to the systems as "Linux32" and "Linux64" respectively. I wanted to get as close to the limit of the PCI-X system as possible. The bandwidth of the PCI-X bus is about 1GB/s. The hope was to get to at least 80% of this speed for the Linux64 to Linux64 transfer, which was the practical limit according to system specialists. I also wanted to see if a Linux32 system is capable of getting the same speed as the Linux64 systems.

~~2~~ 2 Hardware

The hardware used in these test were two IA64 computers and one IA32 computer.

~~2.1~~ 2.1 IA64 Hardware

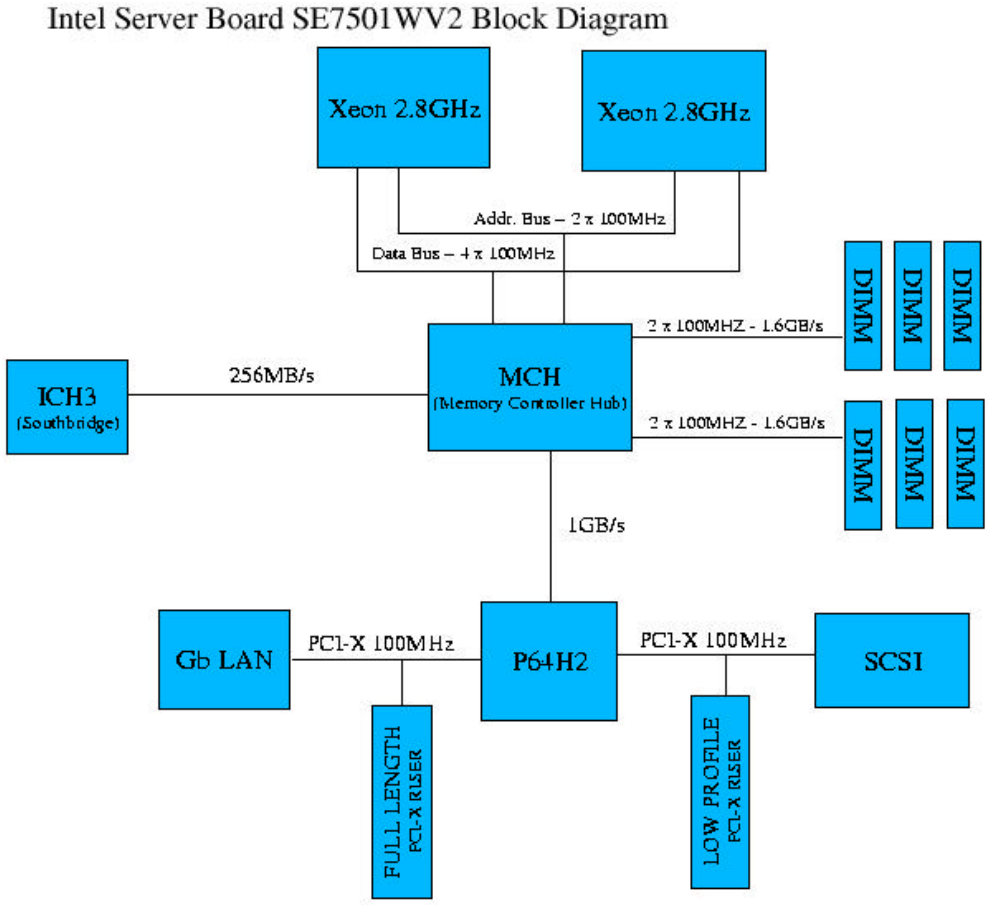
The IA64 computers used are HP RX2600 servers. The specification for these systems can be found at HP's web page[1]. The systems in use here are each configured with two Intel Itanium2 CPUs running at 1.5GHz, 4GB of RAM and 73GB hard disks. An Intel PRO/10GbE network card is connected to the fast (~1GB/s) PCI-X slot. The system use asymmetric IO, i.e. all interrupts goes to one CPU.



[1] http://www.hp.com/products1/servers/integrity/entry_level/rx2600/index.html

~~2.2~~ 2.2 IA32 Hardware

The IA32 computer in use is an Intel Server Board SE7501WV2 based system. It has two Intel Xeon CPUs running at 2.8 GHz. The system is equipped with 1GB RAM, 73GB disk space and an Intel 10Gb network card. This card is the same as used in the IA64 computers. Below is a simple block diagram over the server board.



More information about this system can be found at Intel's homepage[2]. A more detailed block diagram is included in the Technical Product Specification[3] document which can be found in the Technical Information section of the same page.

[2] http://www.intel.com/design/servers/SE7501WV2/
[3] http://support.intel.com/support/motherboards/server/se7501wv2/tps.htm

~~2.3~~ 2.3 PCI-X

The PCI-X protocol is a high-performance extension to the existing PCI Local Bus. It is a 64bit bus running at up to 133MHz. PCI-X is compatible with the old PCI standard. It is possible to use PCI-X cards on a PCI bus. The cards will then run at the speed of the bus (33 or 66MHz). You can also use PCI cards on a PCI-X bus, but this will force the bus to lower its speed to the speed of the PCI card. If one mixes PCI and PCI-X cards on the same bus, the bus will operate at the same speed as the slowest card.

A good introduction to PCI-X can be found on this site:
http://h18000.www1.hp.com/products/servers/technology/pci-x-enablement.html

Being 64 bit and with a speed of 133MHz, the PCI-X bus has a bandwidth of 8512Mb/s, or 1064MB/s.

The PCI-X bus in the IA32 computer used in these tests runs at 100MHz, giving it a bandwidth of 800MB/s.

~~3~~ 3 Software

This section will give an overview of the software used when performing the tests. I start with the Operating System, and then try to explain how the benchmark software works.

~~3.1~~ 3.1 Operating System

The operating system in use is GNU/Linux[4], an open source OS available for a number of architectures. The IA64 systems run the Red Hat[5] 2.1 AW (Advanced Workstation) beta distribution. The deployed Linux kernel is version 2.5.72. During the testing period kernel version 2.5.70 was also used.

The IA32 system runs the Red Hat 8.0 distribution, and kernel 2.6.0-test2. Kernel versions 2.5.70, 2.5.72 and 2.6.0-test1 were also used. See the section about problems to see why so many kernels were tried.

~~3.2~~ 3.2 Benchmark Software

For doing the benchmarking, CERN's GenSink software was used. This does memory-to-memory data transfer over TCP/IP. The software is made up of two parts: The generator and the sink. On the destination (sink) machine, you run the program "sink4". After starting this, you can start the program "gen4" on the other machine. This will now start sending data to the sink machine.

While running, GenSink will print the measured throughput to screen at regular intervals. The printout also contains information on the amount of data transferred, time usage and average throughput. The output will look something like this:

When starting gen4 and sink4, you have to specify a number of arguments:

```
#./sink4

Usage: ./sink4 hostname server_port record_length setsockopt
```

The hostname and server port should be set to the hostname/IP-Address and port of the sink machine for both "sink4" and "gen4". The other two arguments should also match. That is, use the same parameters for both "gen4" and "sink4". The record_length argument sets

[4] http://www.linux.org
[5] http://www.redhat.com

the number of bytes the program sends to the TCP/IP stack in one go. TCP may have to split this into smaller packages to transfer over the network. The setsockopt argument is used to specify the socket send/receive buffer size to be used.

Unless stated otherwise, I use the value 65536 for the record_length argument and 262144 for the setsockopt argument. These numbers was found, by experimenting, to give the best throughput.

It is possible to start multiple copies of sink4 and gen4 on each machine, thus being able to run multiple streams of data. To get the total throughput, the results for all the streams have to be added up. This was at first done by redirecting the output to file and run a shell script to find the overall throughput. This required that each output file had a large number of entries to be able to get accurate measurements.

To make it easier to calculate the bandwidth, and speed up the testing, a small program was written to calculate the throughput without having to go through the output generated by GenSink. This program uses the values in /proc/net/dev to calculate the throughput. This way, no special care has to be taken when multiple streams are running. The program that calculates the throughput has to be run while GenSink is running. You can find the source code for the program in Appendix A.

As well as measuring the throughput, I have in some tests used "top" to find the CPU usage while the tests are running.

~~4~~ 4 Test Results

This section contains the results of my tests. The scripts used to switch between different settings can be found in Appendix B-E. I use the notation A -> B to indicate that system A sends data to system B. A <-> B means that traffic runs in both directions. (Both systems send and receive data, i.e. a pair of GenSinks per stream).

The data transfer was done memory-to-memory over TCP/IP. The machines were directly connected to each other. There were no switches or other network hardware between the machines.

Some tests were done more than once. I only include one set of results for each test in this report.

~~4.1~~ 4.1 *Linux64 to Linux64 Tests*

Here are the test results for the Linux64 to Linux64 tests.

~~4.1.1~~ 4.1.1 Original results (Linux64 -> Linux64)

This test was done prior to my work here. My hope was to be able to bypass these numbers, and get closer to the PCI-X limit.

No Tuning			
MTU	**1 Stream**	**4 Streams**	**12 Streams**
1500B	127	375	523
9000B	173	364	698
+ Kernel Tuning			
MTU	**1 Stream**	**4 Streams**	**12 Streams**
1500B	203	415	497
9000B	329	604	662
+ Driver Tuning			
MTU	**1 Stream**	**4 Streams**	**12 Streams**
1500B	275	331	295
9000B	693	685	643
16114B	755	749	698

Table 1: Original Results

In later tests, 20 streams were tested as well.

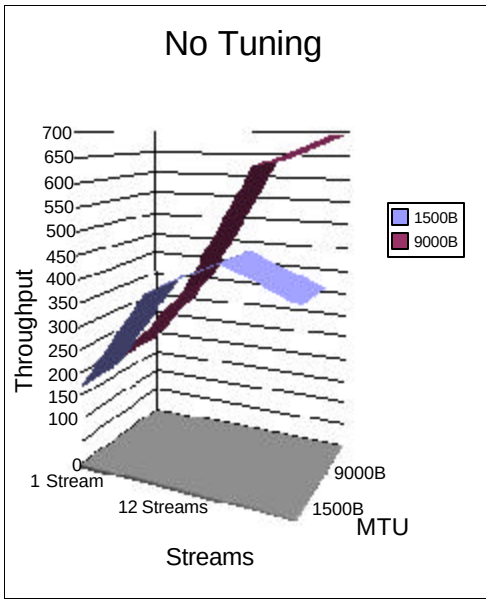
~~4.1.2~~ 4.1.2 Linux64 -> Linux64

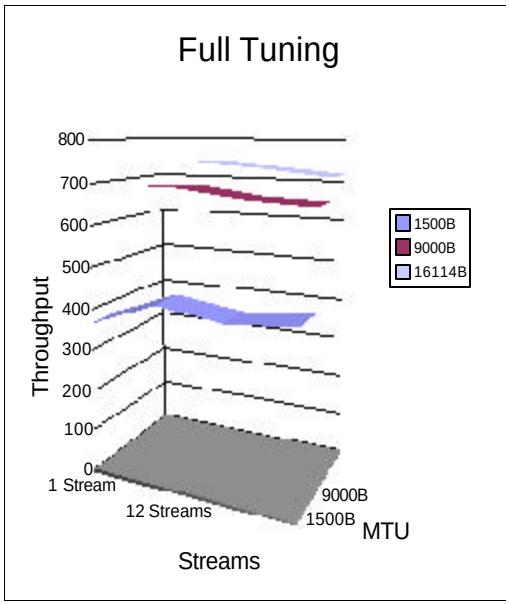
My results for the Linux64 to Linux64 throughput are in some cases lower than the original results. This can have two reasons: First of all, I use my own program to calculate the throughput. This should be more accurate than having to add up a lot of numbers produced by GenSink. Secondly, the GenSink program was changed to print the measured bandwidth more often, so this could cause some speed loss, since the program has to do more work. I reach a maximum throughput of 744MB/s. This is about 70% of the theoretical PCI-X limit. The following table shows my full results.

No Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	170	388	460	398
9000B	182	355	632	694
+ Kernel Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	325	497	470	440
9000B	352	585	672	668
+ Driver Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	366	429	408	421
9000B	686	684	666	658
16114B	744	742	733	717

Table 2: Linux64 -> Linux64 results

We see that for the standard settings and the kernel tuning settings, adding more streams will give higher throughput, up to a certain limit. With an MTU of 1500 bytes, the throughput drops after 12 streams, while with a 9000 bytes MTU the throughput is unchanged or even better with 20 streams. When all tuning parameters are on, only an MTU of 1500 bytes gives higher throughput when adding more streams. 9000 and 16114 bytes MTUs are best with only one stream. The charts below illustrate this.





Only the charts for No Tuning and Full Tuning are shown. The chart for kernel tuning will be similar to the No Tuning chart, but with different values, of course. As you can see, the throughput with full tuning and an MTU of 1500B goes down with 12 streams, and then up again with 20. This is a bit strange. Maybe something was affecting the system when the 12 stream test was run.

When more streams are added, the CPU will have to do more work managing the streams, thus having less computing power left to handle the network traffic. When enough streams are added, the overhead of managing the streams will be so high that the data rate will drop because the CPU can not keep up.

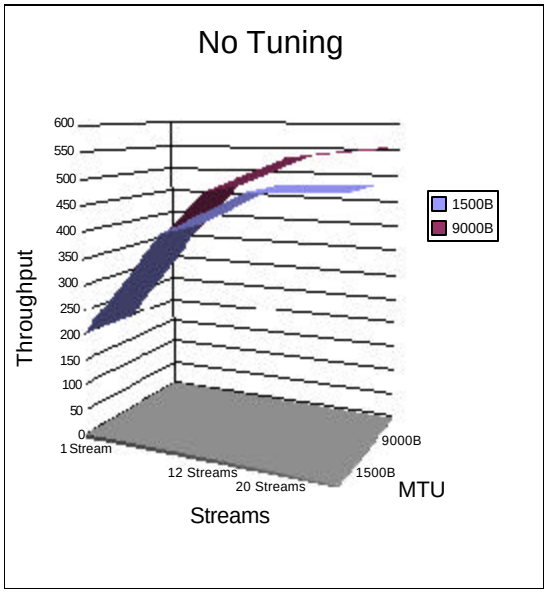
~~4.1.3~~ 4.1.3 Linux64 <-> Linux64

This test ran traffic in both directions. This should give a bit better throughput than running only in one direction. When data is received, an interrupt is sent to one of the CPUs. The other CPU is still free to send data. Note that the number of streams listed in the table is the number of streams in each direction. So, 12 streams means that a total of 24 streams are running (12 in each direction). Before doing this test, I did some experimenting with the GenSink arguments. It looked like setting the record length to 262144 would give better results in some cases. I therefore use that value here. The full results of this test are given in the table below.

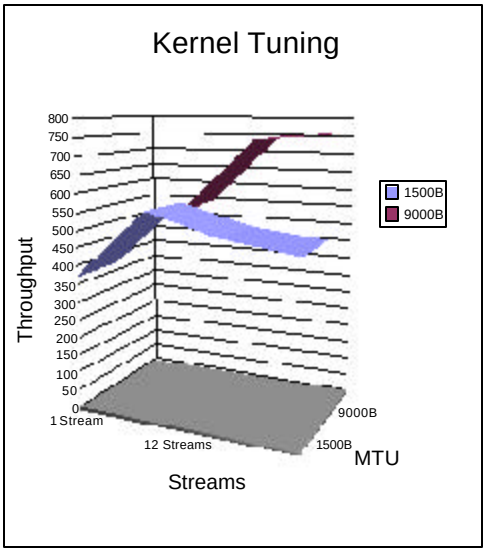
No Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	199	408	491	493
9000B	298	467	540	557
+ Kernel Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	368	563	512	486
9000B	460	544	745	761
+ Driver Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	457	594	535	516
9000B	699	739	719	712
16114B	774	770	751	745

Table 3: Linux64 <-> Linux64 results

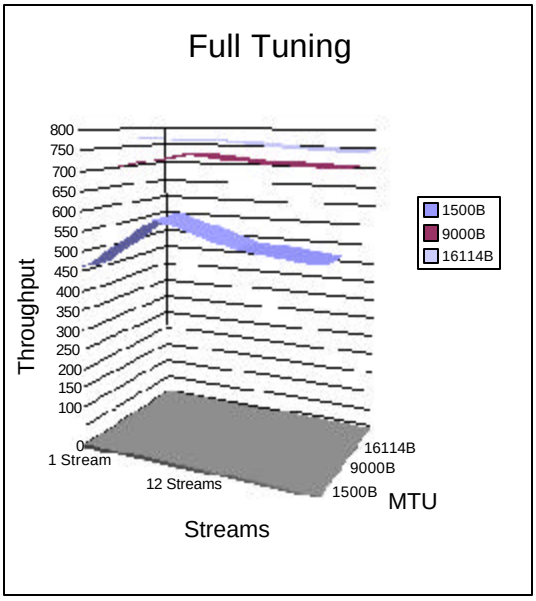
The highest value I get here is 774 MB/s, which is bout 73% of the PCI-X limit.



Looking at this chart, we see that with no tuning, the throughput increases when adding more streams for both MTU values. There is not a big improvement going from 12 to 20 streams. Adding even more streams would most likely make the throughput drop again.



Turning on kernel tuning, the curve for an MTU of 9000B is quite similar to the one given for no tuning. However, the one for a 1500B MTU is quite different. We see here that the throughput clearly drops after 4 streams, while without any tuning it kept improving when adding more streams. The maximum throughput is still higher when the tuning is applied though.



With all tuning parameters applied, the throughput drops after 4 streams. The exception is for an MTU of 16114B where the best throughput is measured when there is only one stream in each direction.

~~4.2~~ 4.2 Linux64 <-> Linux32 Tests

After completing a number of Linux64 to Linux64 tests, I started looking at Linux64 to Linux32. I wanted to know if the Linux32 system can give the same data rate as the Linux64 system.

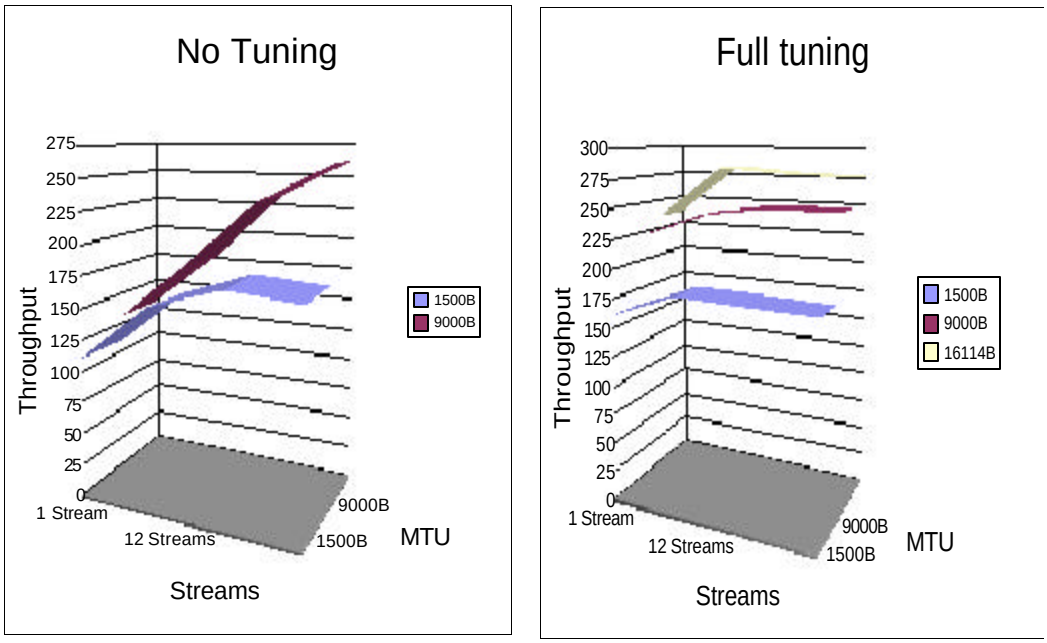
~~4.2.1~~ 4.2.1 Linux32 -> Linux64

In this test, data was transferred from the Linux32 system to the Linux64 system. I experienced several problems with the Linux32 system when sending data. I tried a lot of different kernels, and also tried experimenting with the parameters given to GenSink without luck. For some reason the data transfer would just stop completely from time to time. Sometimes it just paused a long time, while other times it actually stops. The reason for it stopping, seems to be that the Linux network component dies. Since the network stack was so unreliable, getting proper measurements was very hard. Close to the end of my work here, I found that the Linux32 system worked fine when it was running in single user mode. I was therefore able to complete an Linux32 to Linux64 test. The results are given below. (Linux32 in single user mode)

No Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	110	157	176	172
9000B	131	178	233	262
+ Kernel Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	152	211	204	198
9000B	201	236	262	261
+ Driver Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	160	184	182	181
9000B	223	244	252	252
16114B	236	283	278	277

Table 4: Linux32 -> Linux64 results

The charts below show how the throughput changes as the number of streams and MTU is changed.



As seen, the curves for full tuning are much flatter than for no tuning. This means that adding more streams has less effect with all tuning parameters applied. But, the maximum throughput is higher when tuning is applied. The curves for kernel tuning would be somewhere in between the curves shown here.

~~4.2.2~~ 4.2.2 Linux64 -> Linux32

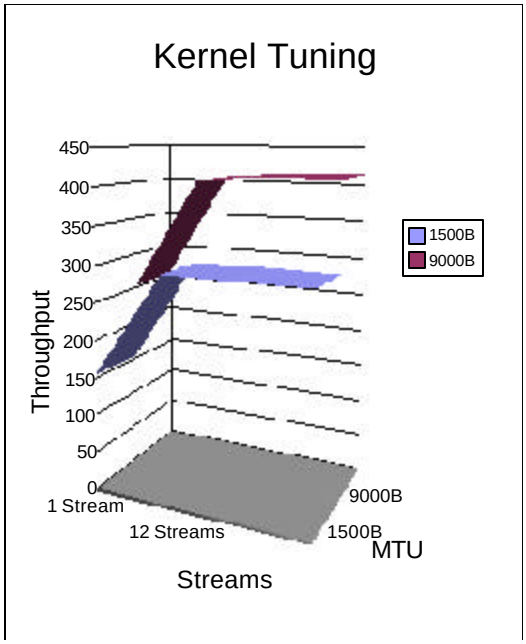
When the Linux32 system were used as sink (receiving data), the network stack was much more reliable. Once in a while the transfer would stop completely, but most of the time it worked. The results given here should be quite accurate. There was no need to run the Linux32 system in single user mode for this test.

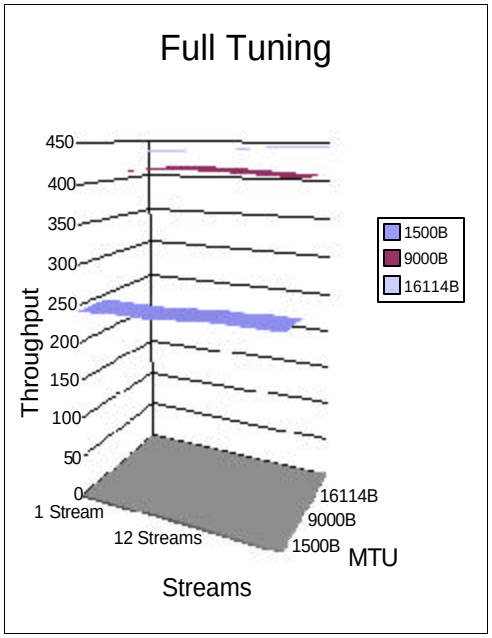
No Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	133	298	313	312
9000B	166	384	416	415
+ Kernel Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	153	295	297	296
9000B	254	407	415	413
+ Driver Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	242	242	248	249
9000B	409	418	416	412
16114B	435	438	444	444

Table 5: Linux64 -> Linux32 results

As seen, transferring from Linux64 to Linux32 gives much higher throughput than the other

way around.





The charts for kernel tuning and no tuning are almost identical. For that reason, only Kernel tuning is included here. The chart for kernel tuning is slightly flatter than the one for no tuning. Looking at the charts for full tuning, you can see that the curves are nearly completely flat. This means that the throughput is almost constant.

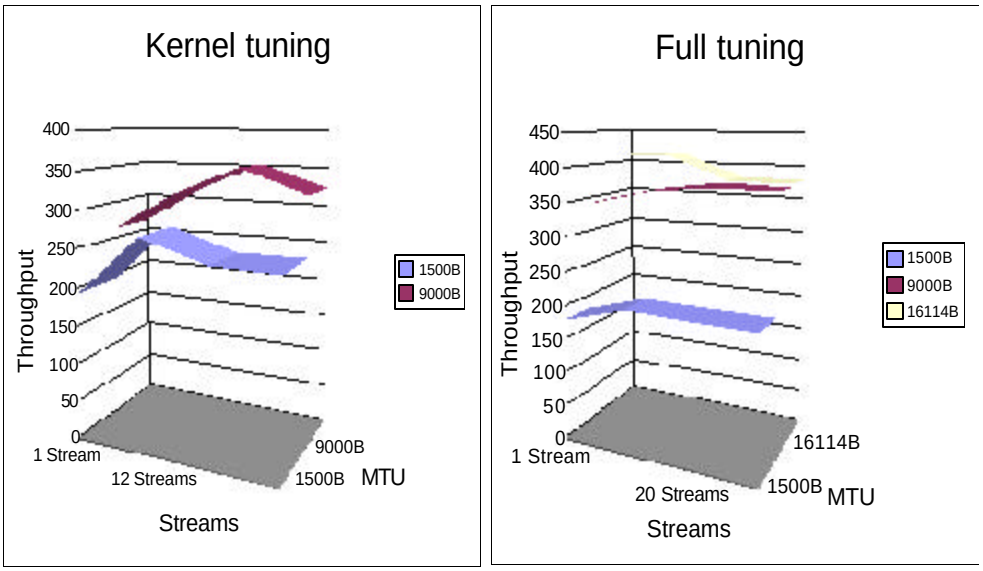
~~4.2.3~~ 4.2.3 Linux32 <-> Linux64

This test ran traffic in both directions. As with the Linux64 <-> Linux64 test, the number of streams given in the table is the number of streams in each direction. Since sending from Linux32 did not work unless the Linux32 system was in single user mode, this was the mode deployed while performing this test.

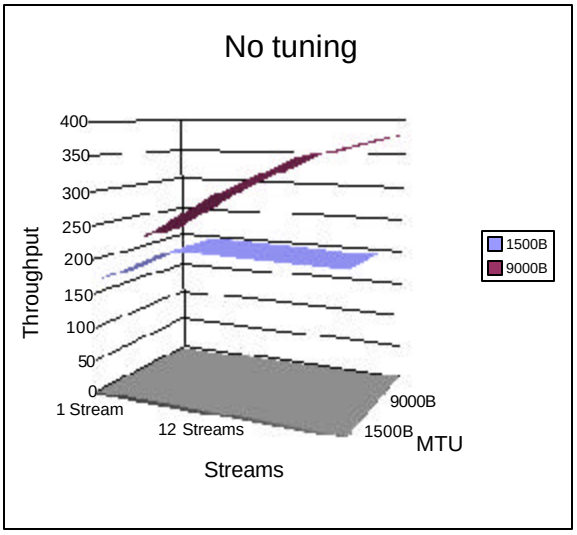
No Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	168	224	220	218
9000B	214	290	346	378
+ Kernel Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	192	273	245	245
9000B	262	314	353	323
+ Driver Tuning				
MTU	**1 Stream**	**4 Streams**	**12 Streams**	**20 Streams**
1500B	179	205	203	203
9000B	343	363	375	373
16114B	408	415	378	377

Table 6: Linux32 <-> Linux64 results

As you can see, the results here are higher than Linux32 to Linux64, but the maximum throughput is somewhat lower than what I got for Linux64 to Linux32. This was a bit disappointing. I was expecting to get higher results when I ran data in both directions, like I did with the Linux64 <-> Linux64 measurements. The reason that it is actually lower, is probably that the Linux32 system was working so hard when sending in one direction that the overhead in adding more streams made it slow down. When running in both directions, twice as many streams are running.



These charts show the throughput with kernel tuning and full tuning. When all tuning parameters are turned on, the throughput is almost constant when adding more streams. With only kernel tuning, we have a clear peak at 4 streams for the 1500B MTU and 12 streams for the 9000B MTU.



With the standard settings, we see that with a 1500B MTU, the throughput increases up to 4 streams, and then start to drop. The system is almost capable of keeping up with the added streams, so the drop in throughput is low. Changing the MTU to 9000B, the throughput keep on increasing even when running 20 streams.

~~5~~ 5 Problems

Here, I try to describe the problems encountered while performing these tests.

~~5.1~~ 5.1 Linux64 Problems

There were not many problems with the Linux64 systems. Most of the time they worked as expected. At one point, the reported bandwidth was very high. I got values of up to 950MB/s. It looks like this was caused by a bug in the Linux kernel. After a kernel upgrade, the results were back to normal. The bug appeared when trying to set the record length argument for GenSink to 262144. It looked as though that value would give higher throughput, but this may have been due to the kernel bug.

~~5.2~~ 5.2 Linux32 Problems

I experienced several problems with the Linux32 system. In this section I try to give an overview of these problems, as well as the steps taken to try to remove them.

1. **Low throughput**

   The first tests done with the Linux32 system gave surprisingly poor results. The maximum bandwidth measured was 256MB/s. After a while the reason was found to be a RAID card connected to the PCI-X bus. This card did not operate at full speed, thus forcing every other PCI-X card to go into low-speed mode. Removing the RAID card solved this problem.

2. **Unreliable network stack**

   After the removal of the RAID card, a new problem was discovered. The network stack was very unreliable, especially when sending data from the Linux32 system to the Linux64 system. From time to time the network stack would just stop functioning, making it impossible to get proper benchmarks. The solution to this problem has not yet been found. I have tried several versions of the Linux kernel, from version 2.5.70 to 2.6.0-test2. This did not help .

   After further investigations I discovered that running the Linux32 system in single user mode makes it more reliable. This could mean that some tasks are interfering with the network driver for the 10Gb card. I was not able to locate the exact reason for the problem. In this mode, it was possible to complete a set of tests without any problems.

~~6~~ 6 Conclusion

After performing these tests, I can say that performance is quite good, but not as good as we had hoped. The goal of 80% of PCI-X speed was not reached. My Linux64 to Linux64 tests maxed out at 744MB/s (70%) for traffic in one direction and 774 (73%) for bi-directional traffic. These are nevertheless very respectable numbers.

When a Linux32 system was used, the results were much lower. The maximum throughput was 444 MB/s when transferring from Linux64 to Linux32. When changing directions, the maximum throughput was only 283MB/s. The PCI-X bus in the 32-bit computer was only running at 100MHz, but this should still give a bandwidth of 800MB/s. I only got about 56% of this, so I do not expect the PCI-X bus to be the limiting factor.

The Linux32 system had some trouble sending data. I was not able to find the exact cause of this, but running the computer in single user mode worked, so there has to be something running that interferes with the network component for some reason. In any case the Linux32 system was not able to perform as well as the Linux64 systems.

The CERN openlab team will try to investigate the Linux32 problem further. These investigations might also help discover what is needed to get the performance of the Linux32 system closer to the Linux64 performance

Appendix A - Benchmark program.

This is the code for the program used to calculate the average throughput.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>

#define DEF_RUNTIME 60
#define INTERFACE "eth2"

struct timeval getBytes(unsigned long *recv, unsigned long *trans) {
  FILE *f;
  char interface[100];
  char *bytePtr;
  struct timeval tv;
  int i;
  char c;

  /* open file */
  gettimeofday(&tv, 0); /* get the current time */
  if( !(f = fopen("/proc/net/dev", "r")) ) {
    printf("An error occured while reading data...\n");
    exit(EXIT_FAILURE);
  }
  /* skip header */
  for(i=0; i<2; i++) {
    do {
      c = fgetc(f);
    }
    while(c != '\n');
  }
  /* find the correct interface */
  do {
    fscanf(f, "%s", interface);
    if(strstr(interface, INTERFACE)) {
      /* get the number of transmitted bytes */
```

```
      fscanf(f, "%*s %*s %*s %*s %*s %*s %*s %ld", trans);
      bytePtr = strstr(interface, ":")+1;
      sscanf(bytePtr, "%ld", recv);
      fclose(f);
      return tv;
    }
  }
  while(!strstr(interface, INTERFACE) && fgetc(f) != EOF);

  /* If we get here, something went wrong */
  fclose(f);
  printf("Could not find the interface %s.\n", INTERFACE);
  exit(EXIT_FAILURE);
}
int main(int argc, char **argv) {
  unsigned long recv_1, recv_2, trans_1, trans_2; /* number of bytes
received/transmitted */
  double av_recv, av_trans, av_total;
  double time_1, time_2;
  struct timeval t1, t2;
  int runtime;

  /* find the desired run time. Use default if not provided */
  if(argc == 2)runtime = atoi(argv[1]);
  else runtime = DEF_RUNTIME;

  /* get the numbers */
  t1 = getBytes(&recv_1, &trans_1);
  sleep(runtime);
  t2 = getBytes(&recv_2, &trans_2);

  time_1 = t1.tv_sec + t1.tv_usec*0.000001;
  time_2 = t2.tv_sec + t2.tv_usec*0.000001;

  av_recv = (recv_2-recv_1) / (time_2-time_1);
  av_trans = (trans_2-trans_1) / (time_2-time_1);
  av_total = av_recv + av_trans;

  printf("Bytes received   : %ld B\n", (recv_2-recv_1));
```

```
  printf("Bytes transmitted: %ld B\n", (trans_2-trans_1));
  printf("Elapsed time     : %.2lf s\n", (time_2-time_1));
  printf("Average Bandwidth:\n");
  printf("\tReceive : %.2lf KB/s\n", av_recv/1000);
  printf("\tTransmit: %.2lf KB/s\n", av_trans/1000);
  printf("\tTotal   : %.2lf KB/s\n", av_total/1000);

  return EXIT_SUCCESS;
}
```

Appendix B - Normal Settings (set_normal)

This is the script used to turn off all tunings (kernel + driver).

```
#! /bin/tcsh


sysctl -w net.ipv4.tcp_rmem="4096        87380   174760"
sysctl -w net.ipv4.tcp_wmem="4096        16384   131072"
sysctl -w net.ipv4.tcp_mem="97280        97792   98304"
sysctl -w net.core.netdev_max_backlog=300
sysctl -w net.core.rmem_default=65535
sysctl -w net.core.wmem_default=65535
sysctl -w net.core.rmem_max=65535
sysctl -w net.core.wmem_max=65535
sysctl -w net.core.optmem_max=20480
sysctl -w net.ipv4.tcp_sack=1
sysctl -w net.ipv4.tcp_timestamps=1
sysctl -w net.ipv4.tcp_tw_recycle=0
sysctl -w net.ipv4.tcp_tw_reuse=0



ifconfig eth2 down;rmmod ixgb
modprobe ixgb
ifup /etc/sysconfig/network-scripts/ifcfg-eth2 boot
ifconfig eth2 up
```

Appendix C - Kernel Tuning (set_ktuned)

This is the script used to turn on the kernel tuning. Note that if you want to go from full tuning to only kernel tuning, you have to apply the set_normal script first, and then add the kernel tuning.

```
#! /bin/tcsh


sysctl -w net.ipv4.tcp_sack=0
sysctl -w net.ipv4.tcp_timestamps=0
sysctl -w net.core.rmem_default=524287
sysctl -w net.core.wmem_default=524287
sysctl -w net.core.rmem_max=524287
sysctl -w net.core.wmem_max=524287
sysctl -w net.core.optmem_max=524287
sysctl -w net.core.netdev_max_backlog=300000
sysctl -w net.ipv4.tcp_rmem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_wmem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_mem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_tw_recycle=1
sysctl -w net.ipv4.tcp_tw_reuse=1
```

Appendix D - Full tuning (set_improved)

This script will turn on all tunings. Both kernel and driver. It also sets the MTU to 16114B which is the highest value supported by the network.

```
#! /bin/tcsh


sysctl -w net.ipv4.tcp_sack=0
sysctl -w net.ipv4.tcp_timestamps=0
sysctl -w net.core.rmem_default=524287
sysctl -w net.core.wmem_default=524287
sysctl -w net.core.rmem_max=524287
sysctl -w net.core.wmem_max=524287
sysctl -w net.core.optmem_max=524287
sysctl -w net.core.netdev_max_backlog=300000
sysctl -w net.ipv4.tcp_rmem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_wmem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_mem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_tw_recycle=1
sysctl -w net.ipv4.tcp_tw_reuse=1




ifconfig eth2 down;rmmod ixgb
modprobe ixgb RxIntDelay=0 RxDescriptors=2048 TxDescriptors=2048 XsumRX=1
XsumTX=1
ifup /etc/sysconfig/network-scripts/ifcfg-eth2 boot
ifconfig eth2 mtu 16114;ifconfig eth2 up
```

Appendix E - Full tuning, Linux32

The Linux32 system uses different values for the tuning. I show the script used to turn on all tunings. The kernel tuning is the first part of this script.

```
#! /bin/tcsh


sysctl -w net.ipv4.tcp_sack=0
sysctl -w net.ipv4.tcp_timestamps=0
sysctl -w net.core.rmem_default=524287
sysctl -w net.core.wmem_default=524287
sysctl -w net.core.rmem_max=524287
sysctl -w net.core.wmem_max=524287
sysctl -w net.core.optmem_max=81920
sysctl -w net.core.netdev_max_backlog=30000
sysctl -w net.ipv4.tcp_rmem="409600  873800   1747600"
sysctl -w net.ipv4.tcp_wmem="409600 163840   1310720"
sysctl -w net.ipv4.tcp_mem="972800  977920   983040"
sysctl -w net.ipv4.tcp_tw_recycle=1
sysctl -w net.ipv4.tcp_tw_reuse=1




ifconfig eth2 down;rmmod ixgb
modprobe  ixgb  RxIntDelay=0  RxDescriptors=2048  TxDescriptors=2048  XsumRX=1
XsumTX=1
ifup /etc/sysconfig/network-scripts/ifcfg-eth2 boot
ifconfig eth2 mtu 16114;ifconfig eth2 up
```