Mark V Memo #4

# The Sustained Disk Streaming Performance of COTS Linux PCs

Ari Mujunen
Metsähovi Radio Observatory

13-Jul-2001

**Abstract**

The ability of generic PC hardware being able to sustain a disk transfer data rate of 256 Mbit/s for a long period of time is tested and verified. The tests are being performed using a standard distribution of the Linux operating system and without using any Linux-specific features in the generic UNIX-style test software written in plain C language.

## Contents

# 1 Introduction

After the introduction of VLBI Standard Interface Hardware (VSI-H) specification,[1] the idea of using a sufficient number of commercial standard PC computers to record and playback VLBI data has been explored and investigated at Metsähovi Radio Observatory. VSI-H nicely standardizes a set of 32 data bit streams without concerning if the data originates from a Gigabit sampler, the sampler stage of multiple baseband converters, from data formatter, or maybe from another recorder. Simple converters from the varying electrical standards used in existing parallel 32-bit data paths of existing VLBI hardware can be built so that any of those can be connected to new VSI-H compliant recording / data transmission systems.

One standard VSI-H cable transfers 1 Gbit/s (32 parallel bits at 32 MHz). This is too much for generic, cheap, everyday office PCs of today. However, this may not be the case for the PCs of tomorrow. Thus an architecture must be developed where the number of PCs used to capture and playback VSI-H signals does not really matter. A natural point to split and combine signals is to split those 32 parallel streams into 2, 4, 8... groups of 16, 8, 4... bits each. This is to be done *without* a full-fledged “crosspoint switch” of any kind—instead a simple “ribbon cable split” is to be used. All the PCs are equally capable of handling any part of the data and if re-organizing (“formatting”) the data is required it should be performed by using standard networking technology (Ethernets, LANs, WANs) moving data files to the point where they are needed.

Before creating (or adapting commercial) hardware for the VSI-H data I/O part of this task it makes sense to verify that the “COTS PC” architecture indeed is capable of writing and reading data files at the VLBI data rates. The basis for this test was the probable scenario of “1 Gbit/s VLBI recorder” consisting of four ordinary PCs with four ordinary hard disks in each of them. Each PC will take care of 8 (VSI-H-style) data bit streams, resulting in a requirement of 256 Mbit/s of sustained data transfer rate for the individual PCs. This article describes the tests performed to verify the viability of long periods of 256 Mbit/s of sustained data transfer rate using commodity PCs as the hardware platform.

# 2 Environment

For these tests a “pure” “commercial off-the-shelf” (COTS) environment was selected. A very popular standard general-purpose/office PC platform was purchased in May 2001[2] from a local PC dealer/shop. For software, the freely-available Linux operating system and its standard GNU C Compiler (GCC) development tools were selected. No “real-time” nor “embedded” development software was used.

## 2.1 Hardware

A commonly-used AMD Athlon motherboard with a VIA chip set was selected (Microstar MSI K7T MS-6330) together with an AMD Athlon 1333 MHz (266 MHz memory bus) “Thunderbird” processor and 256 MB of “PC-133” SDRAM memory. (“PC-266” DDR SDRAM memory was not cost-effective in May 2001.)

[1] `http://dopey.haystack.edu/vsi/index.html`

[2] This PC was not purchased specifically for these tests. It is a replacement for the existing SCSI-based '/home' NFS server at Metsähovi Radio Observatory.

The MSI K7T motherboard supports two "Integrated Drive Electronics/AT Attachment" (IDE/ATA)[3] Ultra DMA/100 interfaces, one primary and one secondary. Both interfaces can accept up to two disk drives and can transfer a peak of 100 MB/s. IDE/ATA transfer rates greater than 33 MB/s require the use of 80-conductor ribbon cables where extra ground wires have been added in between every signal wire for better signal integrity. Two drives sharing a common interface and ribbon cable cannot transfer data simultaneously, however, due to large (typically 2 MB) buffers in disk drives and the fast bus speed, the transfers occur in bursts, allowing a single IDE bus usage to be interleaved between two drives, up to 50 MB/s each.

A large amount of standard RAM was determined to be one of the cheapest ways to buffer continuous streaming data. The typical way to do this would be to use a data I/O board which is capable of doing PCI bus master transfers (or similar mechanisms available in future PC bus systems). An example of such board would be for instance an Adlink PCI-7300A 32-bit digital I/O board[4] with a PLX bus mastering PCI chipset.

Two disk types were selected for this PC: two 40 GB "high-performance" 7200 rpm IBM drives ("IBM Deskstar 60 GXP Series")[5] and one 80 GB "high-capacity" 5400 rpm Maxtor drive ("Maxtor DiamondMax 80").[6] The purpose was to evaluate the real difference in performance levels of supposedly fast drives when compared against drives with the lowest cost per gigabyte.

In addition to the processor, memory, and disks, the configuration includes a standard floppy drive, a keyboard, a display controller (but no own monitor), and a 100 MBit/s 3Com PCI bus-mastering network board.

## 2.2 Linux System Software

The operating system used in these tests was Debian/GNU/Linux 2.2 "potato" with Linux kernel 2.2.19 (that is, the same Linux distribution which will be used in "FS Linux 4"). A standard software configuration as installed by default by the "dselect" installer was used. No attempt was made to disable background daemon processes nor "cron jobs". In particular, the Network Time Protocol (NTP) daemon was active during performance measurements.

To retain compatibility with older IDE/ATA drives, Linux kernels do not enable advanced IDE/ATA features by default. A command called "`hdparm`" is used to manage these settings on a drive-by-drive basis. In this test the following features were turned on with "`hdparm -c1 -d1 -u1 -m16`":

**-c1** 32-bit I/O (instead of 16-bit) is being used.

**-d1** PCI bus master I/O (instead of processor-initiated I/O transfers) is being used.

**-u1** In the IDE/ATA interrupt handler other interrupts are still enabled, i.e. the execution of IDE interrupt handler does not hinder other interrupts from occurring.

**-m16** The low-level IDE/ATA capability to transfer multiple sectors in one command is being used.

[3] http://www.pcguide.com/ref/hdd/if/ide/index-i.htm

[4] http://www.adlink.com.tw/products/DataAcquisition/PCI-7300A.htm

[5] http://www.storage.ibm.com/hdd/prod/deskstar.htm

[6] http://www.maxtor.com/products/diamondmax/diamondmax/default.htm

No POSIX "real-time" extensions of Linux were used, nor memory locking (disabling virtual memory and keeping memory pages in physical RAM). IDE/ATA drive-internal write caching (write commands return as soon as data is in drive memory buffer and not on disk) was not enabled.

## 2.3 System Cost Estimate

When purchased in May 2001 the "1333MHz/256MB" PC and the disks cost FIM 9780; today (11-Jul-2001) the same hardware costs FIM 8280. These prices are "one-off" retail prices and they include 22% of VAT. Thus the corresponding (no VAT) USD prices would be USD 1152 and USD 975. The system was enclosed in an existing 4U rack enclosure for testing.

If we separate the current price for one PC into the PC part and the disk part we get USD 424 for the PC and a four-disk set of 80 GB Maxtor drives goes for USD 1022.

Viable enclosure options for these "VLBI recorder PCs" include an Adlink RACK-200S which is an ATX 2U rack enclosure. It costs USD 300 in Finland but it would require mechanical modification work (openings in top plate) to allow for four swappable IDE disks to be mounted on top of the enclosure. A total of 4*(2U+2U)==16U of rack space would be sufficient for the complete recorder.

4U rack enclosures (such as the one used in this test) do not typically support four 5.25" drive bays, thus they offer no benefit over the smaller 2U model.

A standard mini/midi tower PC enclosure which accepts four 5.25" disk carriers retails typically for USD 65 but this must be compensated by four CRU DataPort Vplus cartridge frames which retail for USD 62 each, resulting in a total enclosure cost of USD 313. Only two mini/midi tower enclosures can be put side-by-side on a standard 19" shelf (their width varies between 180–200mm) and thus a four-PC configuration will probably require well over 20U of rack space. Of course it is possible to locate these four PCs somewhere without a rack.

So, a complete four-PC 1 Gbit/s VLBI configuration would cost (without disks) USD (4*(424+300))==2896 plus rack assembly work. This would accept "bare disks" at USD 4088 per a lot of 16 80 GB disks (the equivalent of 2.133 thin tapes). Alternatively, a "four-tower" configuration with CRU cartridges would be USD (4*(424+313))==USD 2948. This would need the disks in CRU cartridges at USD 68 each, making the 16-disk lot of 80 GB drives USD 1088 more expensive, USD 5176.

As an indicative price for the data I/O board we can mention that Adlink PCI-7300A costs USD 700 in Finland. This is relatively expensive since it is almost as expensive as the PC plus its enclosure itself. It will be probably worthwhile to develop a custom data I/O board using the PCI-7300A as the starting point. (The capability to quickly re-design a data I/O board for a new, upcoming PC bus architecture would be beneficial in any case, since replacements for the current PCI-32/33 bus architecture will start appearing quite soon.)

In any case, it is probably fair to say that the PC costs will divide into two equal halves, the PC and its enclosure ("USD 500+500==1000"). This price will be doubled by the data I/O board, resulting in USD 2000 per PC, or USD 8000 for the set of four. The current going price for disks would be USD 4000 per a set of 16, or USD 5000 if CRU cartridges are absolutely required. The capacity of the disk set is of course expected to increase steadily. Thus USD 12000 should get you a complete 1 Gbit/s recorder with the disks (capable of recording the same amount of data as two thin tapes—or, the later the disks are being bought, more than two thin tapes at the same price.)

## 2.4 Measurement Software

A 200-line plain C program '`wr.c`' (see Appendix A for the source code listing) was written which writes data appearing in a large memory ring buffer using plain UNIX/Linux '`open()/write()/close()`' system calls into ordinary plain UNIX/Linux disk files. (If invoked using the name 'rd', the same program can also '`read()`' the data into buffer.) For the purposes of this disk-only evaluation, the memory buffer was initialized with random values only once in the beginning of program. This is to ensure that no memory and disk block cache sharing techniques (inherently available in Linux) will distort the result.

Two major measurement styles were implemented. With the first command line argument "byte rate" being zero the program measures the total duration and total amount of data written to (or read from) disk(s) and announces this total throughput in decimal "Mbit/s". This is the unit commonly used when discussing VLBI "256 Mbit/s" or "1 Gbit/s" data rates and it refers to one million bits (not 1024*1024 bits) per second. This is quite compatible with the "disk industry standard" practice of announcing disk capacities in decimal MB, "million bytes", and GB, "billion bytes".

With the "byte rate" argument being positive the program calculates the amount of data readily available in memory buffer using system clock and the byte rate argument. This effectively simulates a sustained stream of data appearing in memory buffer. The typical value of "32000000" (32 million bytes per second) was used to simulate the arrival of 8 bit streams at the VSI standard data rate of 32 MHz, i.e. the "target" sustained bit rate of 256 Mbit/s, or, one quarter of the full standard VSI data rate of 1 Gbit/s. If a full '`write()`' block of data is not ready and available, the program "sleeps" for (nominally) 1 ms (in Linux-i386 this becomes 10–20 ms or more), waiting for more data to become available. On the other hand, if system delays between '`write()`' calls become long, the elapsed time is converted to the number of unwritten data bytes waiting in memory buffer and the maximum value of this being tracked. This value tells the maximum amount of ring buffer memory which was actually needed to overcome disk and operating system latencies during the test run.

The rest of the command line parameters allow the "size" of the test run to be modified as follows:

**Block size.** This is the number of bytes written in one '`write()`' call (or read in one '`read()`' call). All other size parameters are counted in multiples of this block size.

**Total blocks.** The total number of blocks to be written to (or read from) files on one or more disk directories.

**Memory blocks.** The number of blocks allocated for the large ring buffer in memory.

**# of directories.** How many directory/file name templates are to be kept open simultaneously and written to (or read from) in round-robin fashion. For each of these:

**Path.** The pathname template which is used in generating file names for this directory. This can be on the same or different disk than other directory/file name templates. The template must contain one '%d'-style '`printf()`' formatting directive which is replaced with an incrementing number when successive files are being created using this name template.

**Blocks.** The maximum number of blocks to be written to (or read from) a single file before incrementing to the next one.

These parameters were changed in numerous test runs to discover the dependencies between parameters and the resulting run-time performance.

# 3 Measurements

The measurement strategy was to determine a usable block size for '`write()`' calls first, then ensure that an adequate amount to memory is available for ring buffer of incoming data to ensure that no data is lost because the disks are busy writing. After these measurements (effectively fixing the block size and memory parameters for '`wr.c`'), disk drive performance variability according to disk type, interface, and data location on disk was measured. Finally, sustained write/read tests of 30 minutes and more were conducted to check that the measured performance levels can be kept for long runs.

## 3.1 Writing Block Size

It is straightforward to design the architecture of write/read software around a single block size which is used for both memory and disk data operations. A small block size would be versatile but it tends to cause extra overhead in processing. On the other hand, an arbitrarily large block size makes it more difficult to e.g. split files etc. Measurements shown in Table 1 were conducted to determine a suitable compromise.

The resulting write bit rate is largely independent of the block size used. Also the block size being a multiple of a power of two does not bring any significant performance increase nor does it noticeably reduce CPU processing overhead. Reducing the block size below a few tens of kilobytes seems to start increasing CPU overhead from the asymptotic value which remains largely unchanged when the size is increased from tens to hundreds of kilobytes. A compromise value between a small and a large block size was chosen for the remaining tests, 40960 bytes.

## 3.2 Memory Ring Buffer Size

PC RAM memory is extremely cheap and thus it is a natural choice for main ring buffer for streaming data. Basically all of "extra" memory (that is, memory not needed for operating system and the running program) could and should be allocated for this ring buffer. Experiments with different buffer sizes were performed and the results are illustrated in Figure 2.

The number of virtual memory page faults increases rapidly when nearly all of the available physical RAM is being consumed by the memory ring buffer. A reasonable amount of RAM must be left for the operating system and for disk buffer cache. The rough division of 160 MB of memory ring buffer, 70 MB of disk buffer cache,

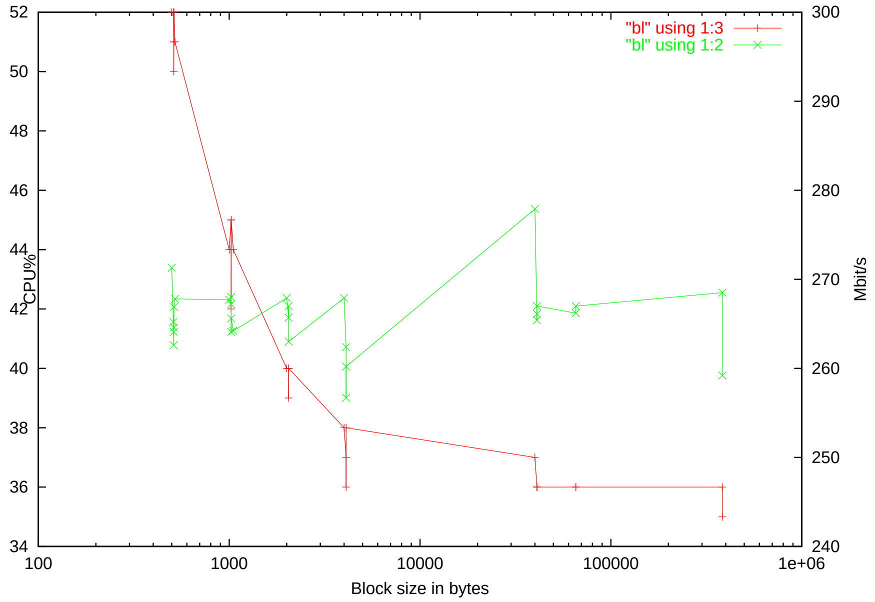

Figure 1: Block size and the resulting bit rate and CPU overhead.

and 26 MB remaining for the program and the operating system seemed to produce reasonably few page faults. With a small memory ring buffer a large amount of memory is left for disk buffer cache use and this produces unrealistically high write bit rates because the blocks written last will be flushed onto disk after test run termination.

### 3.3 Disk Type, Start/End Effects, and Interface Contention

Relatively small (when compared to total disk capacity) 2 GB partitions were allocated on each disk at the physical beginning and end of disk. The remaining space in between these two was partitioned into a single large "middle" partition. In the tables that follow, single-letter symbols are used to refer to these partitions, 's' for start of disk, 'm' for the middle large partition, and 'e' for end of disk.

Repeated measurements with 40960 byte blocks, 45000 total blocks (resulting in 1.8 GB of data in the 2 GB partition) and a 160 MB of memory ring buffer were performed and the results are tabulated in Table 1.

```
# IBM Max IBM  Mbit/s(dec)
1 s            268.25  267.53  269.11 = 268.3
1   e          142.29  143.81  141.89 = 142.6
1     s        234.74  232.25  233.90 = 233.6
1       e      131.33  130.99  130.97 = 131.0
1         s    268.30  264.79  267.34 = 266.8  -> 267.5
1           e  143.76  143.05  142.90 = 143.2  -> 142.9
# I.e. IBM start of disk 267, Maxtor 233 Mbits(dec)/s.
#      IBM end   of disk 143, Maxtor 131 Mbits(dec)/s.
```

Table 1: Write performance vs. data location on disk.

The rotational speed of the disk does not seem to be a decisive factor in sustained disk performance. The combination of bit density on disk tracks combined with the "rpm" figure and with the number of simultaneously operating read/write heads all

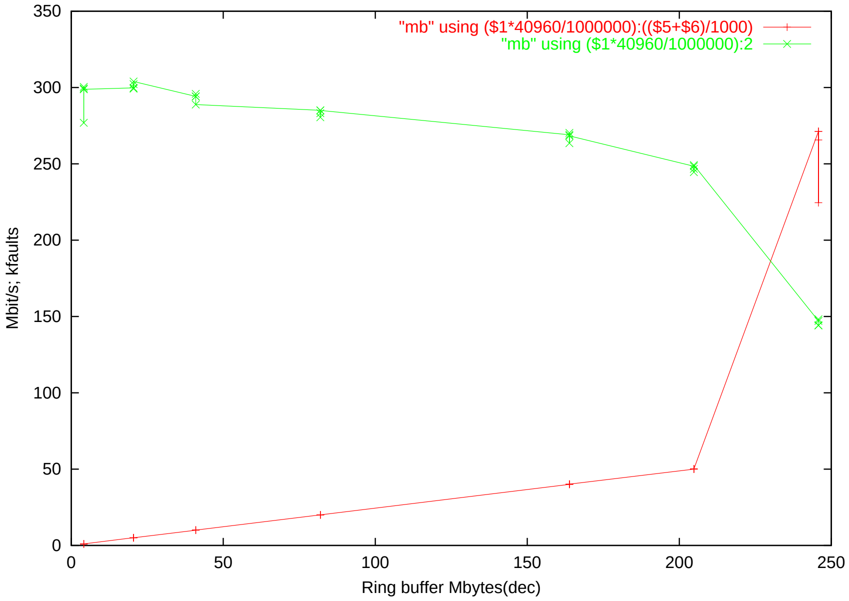

Figure 2: Memory ring buffer size and page faults.

together decide the total throughput of a given drive. This is clearly illustrated in the 7200 rpm 40 GB IBM drive being only 9–14% faster than the 5400 rpm 80 GB Maxtor drive although the rotational speed alone is 33% faster.

From the single-disk measurements it is easy to see that the sustained bit rate is very dependent on the physical data location on disk. In rotational disk drives with constant rotational velocity the bit rate is kept relatively constant by changing the number of sectors per track linearly according to track radius. Although the bit rate coming out of the disk stays in principle constant, the inner tracks are slower because track-to-track seeks must be performed more often.

Fortunately, for both drives tested the ratio between slowest and fastest bit rates does not fall below 0.5, suggesting that extremely short radii, i.e. very short inner tracks are not (yet?) being used by disk industry to achieve extra (low-performance) storage capacity.

The measured performance levels agree quite well with published specifications. For IBM drives, the published sustained transfer rate is 40.61 MB/s (324.9 Mbit/s) in the beginning of disk and 19.42 MB/s (155.4 Mbit/s) at the end of disk.[7] Maxtor does not publish sustained performance figures for the "DiamondMax 80" drive,[8] but their upcoming "Maxtor 536DX" 100 GB drive has a similar architecture (only track density has increased from 34000 tpi to 52000 tpi) and they have published sustained transfer rates of 16.4 MB/s (131.2 Mbit/s, end) to 29.0 MB/s (232.0 Mbit/s).[9]

To determine the achievable throughput when two disks are being written "semi-simultaneously", i.e. writing blocks alternately on two disks, the same amount (1.8 GB) of data was written split into two directories/files on two disks. Various combinations of same and different type disks on primary and secondary IDE interfaces and as master and slave devices were tested. The results are in Table 2.

Combining two disk areas of similar performance (for instance, two "start" or "end"

[7] http://www.storage.ibm.com/hdd/support/prodspec/d60gxp_sp.pdf

[8] http://www.maxtor.com/products/diamondmax/diamondmax/FullSpecs/dm80spec.pdf

[9] http://www.maxtor.com/products/diamondmax/diamondmax/FullSpecs/536DXspec.pdf

```
#./wr 0 40960 45000 4000 2...
# IBM Max IBM  Mbit/s(dec)
2 s       s    407.66
2 s         e  294.28
2   e     s    282.17
2   e       e  274.49
2     s   s    406.73  405.50
2     s     e  292.67
2       e s    261.80
2       e   e  264.01
2 2s           269.40  Two files; as fast as one file
2 s e          152.80  Max seeking still faster than end only.
2 s e          154.10
2  2e          142.40  Two files; as fast as one file
2 s   s        430.84  425.14  423.54
2   e s        261.06
2 s     e      259.91
2   e   e      257.87
```

Table 2: Writing to two locations at the same time.

areas of the same disk type) results in roughly 75–95% of the "theoretical" performance of 2x, i.e. the performance figures simple added together. Combining areas of different performance results in usually roughly 95% of the 2x performance of the slower of the areas. This is natural, since a program which writes an equal amount of data (interleaved) on two drives is effectively bound by the throughput of the slower drive, i.e. the time needed to write half of the data on the slower drive dominates the end result.

No noticeable performance difference was found between primary and secondary IDE interfaces nor between master and slave disks.[10] Ultra DMA/100 apparently allows actual transfers go much faster than what the drives need to actually write the data onto disk. Also the built-in Linux buffer cache helps by slightly delaying blocks to be written in main memory and then sorting and grouping writes into larger single writes. Write buffer cache makes it also possible to write in two locations on the same disk at the same time since the disk accesses are interleaved in buffer cache memory (and not on disk causing extra head seeks).

Combining three disk areas (on three separate disks) further increases performance over two-disk configurations. In the case when "slow" end areas of disks are being used, the combined performance reaches still 86% of the "theoretical" summed performance of the three disks. Please see Table **??** for measurement results.

```
#./wr 0 40960 135000 4000 3 ...
# IBM Max IBM  Mbit/s(dec)
3 s   s   s    417.68  416.12  416.47 = 416.7
3   e   e   e  367.00  364.26  351.47 = 360.9
```

Table 3: Writing to three locations at the same time.

However, when three "fast" start areas of the disks are being used, the speedup factor remains quite low, only 54% of the "maximum attainable" performance and not significantly better than two-disk performance. This is most probably because at speeds of 300–400 Mbit/s (40–50 MB/s) we are starting to see effects of motherboard, chipset, and possibly PCI bus performance limits. Also sharing a single IDE/ATA bus cable at Ultra DMA/100 leaves less than 50 MB/s per drive avail-

[10] Admittedly the IBM disks were never tested as slaves nor the Maxtor as a master.

able. Even with four disks operating concurrently it will probably be difficult to achieve 512 Mbit/s sustained performance using the current PC architecture.

However, at speeds below 300 Mbit/s the hard disk alone is clearly the dominating bottleneck and not the CPU, memory, operating system, or user software.

## 3.4 Sustained Writes and Reads

The first measurements with large 31 GB partitions ("middle partitions") on each disk confirm that using two disks simultaneously is enough to get more than 256 Mbit/s sustained write throughput. (See Table 4.) These tests wrote a total of 61 GB of data in each run, splitting it into files of 46875 blocks (40960 byte blocks). This file length represents exactly one minute of 8 bit streams sampled at 32 MHz. The attempt to write this amount on a single 62 GB "middle" partition on the Maxtor drive confirmed that alone it cannot sustain the 256 Mbit/s rate desired.

```
# ./wr 0 40960 1500000 4000 2... 46875 blocks each file
# hda hdb hdc
# IBM Max IBM  Mbit/s(dec)
2  m       m   372.34  374.29
2      m   m   386.11  386.37
2  m   m       384.40  383.00
# ./wr 0 40960 1500000 4000 1...
1      m       185.23  (1500000 blocks)
1      m       189.00  (1680000 blocks)
```

Table 4: Two- and single-disk sustained writes.

The ability of two-disk combinations to really sustain 256 Mbit/s writing for half an hour was tested by setting the incoming byte rate to '32000000' in the test program command line. The results are shown in Table 5.

```
# ./wr 32000000 40960 1500000 4000 2...
# hda hdb hdc
# IBM Max IBM  Mbit/s  Max buffer
2  m       m   255.993  6881524 (6.6MB)
2      m   m   255.992  8410129 (8.0MB)
```

Table 5: Maximum memory ring buffer need.

These tests ran for 1920 seconds (32 minutes) each and the peak amount of unwritten data in memory buffer was very small when compared to the amount allocated (160 MB) for this purpose.

Another set of tests were performed where the data written on disks was also read back into memory ring buffer (for simulated "playback" by a bus master data I/O board). The results in Table 6 show that reading is at least as fast as writing and in the case of equally performing disks, many times even 10–30% faster than writing. Each of the tests in the table ran for 20–30 minutes, the last to closer to an hour, so these together represent roughly 8 hours.

An interesting detail is the last test performed where the "twice-as-large" 62 GB Maxtor partition is used twice, i.e. two files are being written in the same file system at the same time. Even this does not destroy the sustained system performance which stays well above 256 Mbit/s.

The same sustained read/write tests as in Table 6 were also performed with the simulated data I/O rate set at 32000000 bytes/s and the results are shown in Table 7.

```
# ./wr 0 40960 1500000 4000 2...
# ./rd 0 40960 1500000 4000 2...
# hda hdb hdc  Write   Read
# IBM Max IBM  Mbit/s(dec)
2  m       m   373.62  500.61  1/2hrs each
2      m   m   386.63  409.86
2  m   m       383.50  380.24
1      m       195.58  189.12
1      m       188.29  184.42
3  m   m   m   410.69  503.09  (2250000 blocks)
4  m  2xm  m   377.98  322.33  (3000000 blocks, two files on Max, 1hr)
```

Table 6: Sustained read/write tests measuring throughput

```
# ./wr 32000000 40960 1500000 4000 2...
# ./rd 32000000 40960 1500000 4000 2...
# hda hdb hdc  Ring buffer requirement
# IBM Max IBM  Write               Read
2  m       m     6892558 ( 6.6MB)  22855103 (21.8MB)
2      m   m     8276389 ( 8.0MB)  24313886 (23.2MB)
2  m   m        34840486 (33.2MB)  51981966 (49.6MB)
1      m      2147483042 (2GB)   2147479527 (2GB) overflow
1      m      2147430094 (2GB)   2147482470 (2GB) overflow
3  m   m   m    10895783 (10.4MB)  22032936 (21.0MB)
4  m  2xm  m    28257460 (26.9MB)  80679196 (76.9MB)
```

Table 7: Sustained read/write tests measuring memory buffer need.

Even in the worst case where the (inherently slower) Maxtor disk is abused by forcing it to write double the amount of data in the same time as faster IBM drives, even then only less than half of the available 160 MB of ring buffer is actually needed.

The sustained data rate tests ran for about 28 hours in total without ever “dropping” any data if two or more disks were used concurrently.

### 3.5 CPU Consumption

The user and system CPU percentages reported by ’time’ command indicate that even in the case of attempting to write to the disk at its maximum allowed rate, no more than 40–50% of CPU is required. Thus it is highly likely that some data processing tasks (such as decimation of sampled data to support lower sampling rates than 32 MHz) could be well done in software.

When doing ’usleep()’ calls due to rate limitation the reported CPU usage percentage dropped quite significantly for all the sustained tests, below 15%. This may be because some of physical disk writing activity is now counted as “general system CPU time” and not anymore charged to a sleeping process.

However, there was no test which could achieve a larger CPU load than 57%. It is quite apparent this application is not bound by current GHz-class CPUs.

## 4 Lessons Learned

The price drop rate of generic PC hardware is astonishing: the price of the test system fell more than 15% in less than three months. However, the drop occurs only in the class of most generic "office PCs". Any "CompactPCI", "ISP Rack Server" and similar specialty variants of the PC architecture will remain expensive and are thus bad candidates for a highly "clusterized" system with many PCs.

We learned that with two disks writing concurrently we will probably sustain more than 256 Mbit/s and with 3 or 4 disks we will attain more than 300 Mbit/s—but that it will be difficult to go over 500 Mbit/s with the existing hardware. However, it should always be kept in mind that by sticking to the most standard hardware architecture this problem will cure itself automatically. Next-generation "office PCs" only 1–2–3 years away will easily surpass the 512 Mbit/s barrier. Please note that specialty variants like "CompactPCI" will always lag behind in performance and remain more expensive.

Using only "standard Linux" without any "real-time" or "embedded" extensions helps enormously with software when the hardware below has to be changed to get more performance. The UNIX mode 'wr.c' program shown in Appendix A could have run 25 years ago in original UNIX systems or equally well in Linux 0.99.5 in 1994.[11] With 1998 vintage 233 MHz Pentiums and SCSI disk arrays it is problematic to get anything more than 40 Mbit/s using 'wr.c'—and the RAM needed for ring buffer memory would have been quite expensive. This shows the absolutely dramatic increases of performance in everyday PC hardware.

# 5 Conclusions and Further Work

The concept of four standard PCs each recording one-fourth of the full 1 Gbit/s VSI-H data seems quite feasible in the light of the current disk write/read tests. The results make it worthwhile to pursue the data I/O interface options (VSI-H converters, PCI bus mastering input/output boards, 1-to-4 "splitters").

Future work and additional tests could include the following:

1. Motherboard/IDE Interface type dependencies: a test with a recent Intel motherboard and/or a test with the "slower" (Maxtor) drive as a primary master would be applicable.
2. It might be useful to check with a long sustained run if a large Linux buffer cache would actually be more useful than an unnecessarily large memory ring buffer.
3. Testing with real PCI bus master data input board which would really fill in data in the memory ring buffer. Will PCI bus be so occupied that disk writes (or data input) will slow down?
4. Will disk PCI bus traffic appear at all on the "real" physical PCI bus? Several motherboard chipset documents seem to suggest that PCI IDE/ATA disk traffic will occur inside the chipset and does not cause any traffic at all on the "real" PCI bus, leaving it completely for the data I/O board.
5. Experiments with the Linux file system (shorter files? numerous files? longer file names?) would be in order to verify that data files can really be organized quite freely.

[11] Of course we would have discovered performance less than 1 Mbit/s or similar, something definitely not enough for VLBI.

6. Testing using the CPU for reducing sample rate (decimating; probably easy) and/or rearranging input bit streams into separate files. It would probably be possible to add conventional “Mark IV” formatter frames in playback phase.
7. Writing and reading multiple (say 8) files per disk concurrently? Quick tests indicate that writing will probably be easy but that reading can easily cause lots of seek activity and thus slow down the sustained data rate. However, four files per file seems to be generally okay.
8. The optimum (cheapest) physical layout: PC enclosure, disk arrangement, carriers?

## Appendix A. 'wr.c' source code

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include <sys/types.h>  /* open() */
#include <sys/stat.h>  /* open() */
#include <fcntl.h>  /* open() */

#include <sys/time.h>  /* gettimeofday() */
#include <unistd.h>  /* gettimeofday(), usleep() */

#include <string.h>  /* strstr() */


typedef struct sDir {
  char *nameTemplate;
  int nameNumber;
  int blocksPerFile;
  int fid;
  int nowBlocks;
  double prevOperTime;
  double maxOperTime;
} tDir, *pDir;


/* Buffer occupancy counters. */
int byteRate;  /* bytes per second */
int bytesInBuffer;  /* how much unread data in buffer */
int maxBytesInBuffer;  /* the max of previous */

double
tim(void) {
  struct timeval tv;
  double t;
  static double prevT = 0.0;

  assert( gettimeofday(&tv, NULL) == 0 );
  t = (double)tv.tv_sec + (double)tv.tv_usec/1000000.0;
  if ((prevT > 0.0) && (byteRate > 0)) {  /* has been called once; call #2.. */
```

```
    /* Update the amount of data "flown" in & its max. */
    bytesInBuffer += (int)(byteRate*(t - prevT));
    if (bytesInBuffer > maxBytesInBuffer) {
      maxBytesInBuffer = bytesInBuffer;
    }
  }
  prevT = t;
  return t;
}  /* tim */


void
randbuf(char *buf, ssize_t nbytes)
{
  int i;

  for (i=0; i<nbytes; i++) {
    buf[i] = (char)rand();
  }  /* for */
}  /* randbuf */


int
main(int argc, char *argv[]) {
  int readMode;
  int blksize;
  int totalblks;
  int memblks;
  char *memblk;
  char *p;
  int ndirs;
  pDir dirs;
  int i;
  double recordStart;
  double totalDur;
  int usleeps;
  int blk;
  double mbits;

  if (argc < 6) {
    fprintf(stderr, "%s: needs at least byterate blocksize totalblocks memblocks ndirs { path
    exit(EXIT_FAILURE);
  }

#define TIMESTART(i) \
      { dirs[i].prevOperTime = tim(); }
#define TIMESTOP(i) \
      { double dur = tim() - dirs[i].prevOperTime; \
        if (dur > dirs[i].maxOperTime) dirs[i].maxOperTime = dur; }

  /* Init variables according to command line, allocate buffers. */
  readMode = (strstr(argv[0], "rd") != NULL);
  /* later, after rand() time measurement: byteRate = atoi(argv[1]); */
  blksize = atoi(argv[2]);
```

```
totalblks = atoi(argv[3]);
memblks = atoi(argv[4]);
assert( (memblk = malloc(memblks*blksize)) != NULL );
/* Init memblk with random values. */
byteRate = 0;
{
  double st = tim();

  printf("Initializing memory buffer with rand()...");
  fflush(stdout);
  randbuf(memblk, (memblks*blksize));
  printf("took %f seconds.\n", tim()-st);
}
byteRate = atoi(argv[1]);
ndirs = atoi(argv[5]);
assert( (argc >= (6 + ndirs*2)) );
/* Allocate one extra struct for call-to-call time statistics. */
assert( (dirs = malloc((ndirs+1)*sizeof(tDir))) != NULL );
for (i=0; i<ndirs; i++) {
  dirs[i].nameTemplate = argv[6+i*2];
  dirs[i].nameNumber = 0;
  dirs[i].blocksPerFile = atoi(argv[7+i*2]);
  dirs[i].nowBlocks = dirs[i].blocksPerFile;  /* init to end */
  dirs[i].maxOperTime = 0.0;  /* init to smallest value */
}  /* for each directory */
dirs[ndirs].nameTemplate =   "call-to-call  ";
dirs[ndirs].maxOperTime = 0.0;

/* Start looping for total number of blocks. */
bytesInBuffer = maxBytesInBuffer = usleeps = 0;
recordStart = tim();
TIMESTART(ndirs);  /* call-to-call (not per file) */
for (blk=0; blk<totalblks; blk++) {
  i = blk % ndirs;
  if (dirs[i].nowBlocks >= dirs[i].blocksPerFile) {
    /* Open a new file in this directory / template. */
    char pn[255];
    int n = blk / dirs[i].blocksPerFile;

    snprintf(pn, sizeof(pn), dirs[i].nameTemplate, n);
    if (n==0) TIMESTART(i);  /* only for first file */
    if (readMode) {
      assert( (dirs[i].fid = open(pn, O_RDONLY)) != -1 );
    } else {
      assert( (dirs[i].fid = creat(pn, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) != -1 );
    }
    dirs[i].nowBlocks = 0;
  }

  /* If we are doing rate-limited testing, check for available new data. */
  if (byteRate > 0) {
    while (bytesInBuffer < blksize) {
      /* Not one full block in buffer, wait. */
      usleep(1000);  /* a small amount, probably ends up to be 10--20msec */
      usleeps++;
      (void)tim();  /* update 'bytesInBuffer' */
    }  /* while not at least one full block in buffer */
  }

  /* Write (or read) one block. */
  p = &(memblk[(blk % memblks)*blksize]);
  if (readMode) {
    assert( (read(dirs[i].fid,
                   p,
                   blksize)) == blksize );
  } else {
    assert( (write(dirs[i].fid,
                   p,
                   blksize)) == blksize );
  }
  TIMESTOP(i);  /* accumulates close--open-write in worst case */
  TIMESTOP(ndirs);  /* call-to-call */
  /* Once the previous have updated 'bytesInBuffer' increase, */
  /* only then decrement for the amount written. */
  bytesInBuffer -= blksize;
  TIMESTART(i);
  TIMESTART(ndirs);
  dirs[i].nowBlocks++;

  /* Time to switch to another file? */
  if (dirs[i].nowBlocks >= dirs[i].blocksPerFile) {
    /* Close the ready file before opening a new. */
    assert( (close(dirs[i].fid) != -1) );
  }
}  /* for each block */
TIMESTOP(ndirs);
totalDur = tim() - recordStart;

/* Final report. */
mbits = 8.0*totalblks*blksize/totalDur/1000.0/1000.0;
printf("Took %f seconds, %f Mbits(dec)/s (%.1f%% of PCI33).\n",
       totalDur,
       mbits,
       mbits/(33.0*4.0*8.0)*100.0
       );
if (byteRate > 0) {
  printf("Max. unwritten bytes in buffer %d  (%d usleeps()).\n",
         maxBytesInBuffer,
         usleeps
         );
}
for (i=0; i<=ndirs; i++) {
  printf("%s  max. %f seconds.\n",
         dirs[i].nameTemplate,
         dirs[i].maxOperTime
         );
}  /* for each directory */
```

```
  return(EXIT_SUCCESS);
}  /* main */
```