Experiences: Overcoming Data Transfer Bottlenecks across SUN-Transputer Interfaces

M. Stella Atkins, Yan Chen and Florina Olariu
Simon Fraser University and TRIUMF

ABSTRACT: In many computationally-intensive tasks such as medical image reconstruction for which transputers can be used, the data communication rate between a display host such as a SUN-workstation and the transputer network becomes a constraint on system performance. Our goal is to maximise the SUN-transputer data exchange rate. We use several kinds of transputers connected to the VME-bus of the host SUN and this paper explores alternatives for designing their interfaces. We also use a VME-bus memory module memory-mapped to both the SUN and the transputers.

Device drivers for the transputers connected by serial links must implement the required blocking semantics of the transputer links. There are two major designs for programmed i/o serial link interfaces—"polled" interfaces, and "interrupting" interfaces which work with a process scheduler to block the invoking process until the desired condition is true. We measured both kinds, expecting that the transputer hard link throughput would limit the performance. However, we found that the SUN's cpu cycles limit the performance of an

Research funded by the Natural Science and Engineering Research Council of Canada, under grants #8020 and #45395, and by the Simon Fraser University Center for Systems Science.

unbuffered byte interface; a SUN4/110 is only capable of transferring 457,000 bytes/second. By reducing the software overhead in the SUN using a buffered FIFO block transfer we can increase throughput to 888,000 bytes/second. Now the bottleneck is the byte-wide VME-bus access mode used for the serial links.

Throughput between the SUN and the transputers, using shared 32-bit wide VME-memory, is four times the throughput of the serial links. Hence, for optimum throughput, we use the transputer links for synchronisation of data which is exchanged in bulk using the shared memory. Further performance increases are only possible using different hardware interfaces which support the VME-bus block transfer mode.

1. Introduction

Our application is the compute and data-intensive task of real-time data acquisition and processing of medical tomography data from next generation 3D Positron Emission Tomography (PET) scanners currently being developed at TRIUMF [Rogers et al. 1989, 1990]. In PET the patient is injected with a substance such as water or glucose, in which the molecules have been altered to contain a positron emitting tracer. This tracer travels in the body as part of the injected substance and accumulates in the organs where the substance is used. As each atom of tracer decays, a positron is emitted which travels a short distance and then collides with an electron. Both the electron and the positron are annihilated. The energy from this annihilation is given off in the form of two gamma rays, which travel in directions 180 degrees from one another. The annihilation producing the gamma rays is called an event. A detected event is one for which both gamma rays from an annihilation are detected on the ring of Tomograph detectors. The detected event is encoded by the detector hardware providing the information to the data acquisition system which is necessary for image reconstruction and analysis. These raw event records are used to reconstruct a 3-dimensional image of the tracer activity within the object being imaged. The manner in which these raw event records are manipulated depends on the specific reconstruction algorithm which is being used. Since the data acquisition is a necessary first part of image reconstruction, it is desirable to process the events in real-time, as it is desired to display the image soon after the data is collected. This is useful for checking that the object being imaged has been placed in the correct position. Meeting this real-time processing goal requires around 20 MFlops performance from the data acquisition system [Murray 1990], and an estimated 400 MFlops for real-time 3D image reconstruction.[2]

One approach is to have a single processor with the desired performance, but it does not scale and is expensive. Our approach is to use many smaller processors working together in parallel [Wilkinson et al. 1988; Murray 1990; Atkins et al. 1991]. This maintains the flexibility of software control and scalability while at the same time being reasonably priced. We decided to use a network of transputers as a parallel engine.

As our application is both compute and communication intensive our goal is to maximise the SUN-transputer data exchange rate. For portability we prefer to use off-the-shelf hardware. Problems arise because of the complex interrelationships between hardware and software, and the difficulty in writing efficient code for hardware/software interfaces. This paper addresses some of the issues and outlines our solutions, extending work reported in [Atkins & Chen 1991] with results for data transfers using VME-memory.

1.1 Transputer Overview

The transputer T800 is a RISC processor which was introduced by INMOS with 4 kilobytes of on-chip memory, 4 high speed data links, and a memory interface [Stein 1988], thus allowing parallel processing networks to be built easily and economically. The transputer allows communication and computation to occur simultaneously and

[1] Both the event size and processing algorithm will vary as the gamma ray detection hardware continues to improve, so the exact rates quoted are understood to be adopted only for the purpose of making this study concrete.

autonomously. There is no memory shared between any two nodes in a transputer network, thus the autonomous communication improves the efficiency of the message passing model used by transputer networks. As each transputer has four data links with which to pass data to other nodes a large number of topologies can be easily built. This connection flexibility gives the system designer many topology choices for exploiting the parallelism of a problem.

1.2 Architecture Environments

1.2.1 Introduction

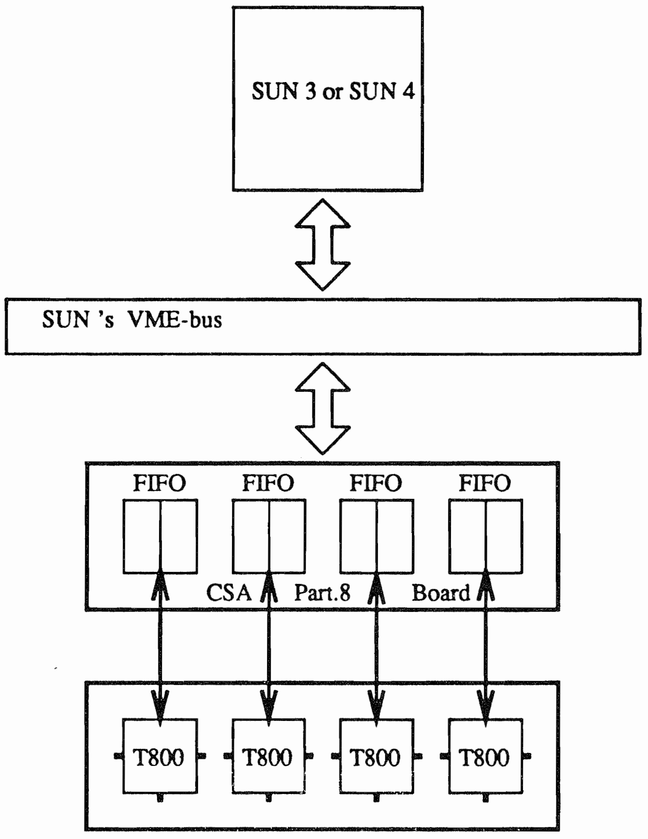
The medical images are displayed on a SUN workstation running the UNIX operating system, which acts as a host to a network of transputers which supplement the SUN's computing horsepower. The SUN's VME-bus is used to connect the SUN to the transputer network.

The software used for the investigations described here is developed using Logical Systems C compiler [Logical 1988]. This C system possesses a library which allows the concurrency features of the transputer to be easily exploited. The features of this library are similar to those provided by the OCCAM language originally designed for the transputers [INMOS 1987].

One hardware configuration used for the tests is based on CSA boards [CSA 1989] connected to the VME-bus of either a SUN3/110, a SUN4/110 or a SUN4/260 host through serial links, shown in Figure 1 and detailed in Section 1.2.2. The other uses Parsytech boards [Parsytech 1988] and some VME-shared memory connected to a SUN4/370, shown in Figure 2 and detailed in Section 1.2.3.

1.2.2 Use of Serial Link Interfaces

Figure 1 shows the use of serial link interfaces, where CSA transputers are connected to a SUN3 or SUN4's VME-bus. The hardware consists of a CSA Part.8 board with 4 32-bit link adapters connected to the SUN's VME bus, and also connected via transputer hard links (which are bi-directional RS422 interfaced links where every byte of data sent on a link is acknowledged on the input of the same link) to a CSA Part.6 board with 4 T800 transputers [CSA 1988]. Many more transputers are connected in a network to the transputers on the Part.6 boards (not shown).



CSA Part.6 Board

Figure 1: Architecture of the CSA System.

The SUN must treat each link as a synchronous serial link which can be accessed through blocking i/o invocations, similar to the transputer's blocking *ChanIn/ChanOut* invocations. These synchronous i/o invocations have all the advantages of OCCAM's synchronisation methods [Barrett & Suffrin 1991] which are almost indispensable for programming parallel MIMD architectures.

One disadvantage of this architecture is that the transputer link interfaces may be slower than the host-VME interfaces, and at best, on any single link, data can only be transferred in one direction at the transputer hard link speed of 20 Mbps, or 1.7 MBytes/second of actual data. We therefore expect that the transputer hard link bandwidth will limit the Sun-transputer performance to 1.7 MBytes/second. Another disadvantage relates to the implementation of the synchronous i/o invocations on the SUN/UNIX host; the invoking process on the SUN

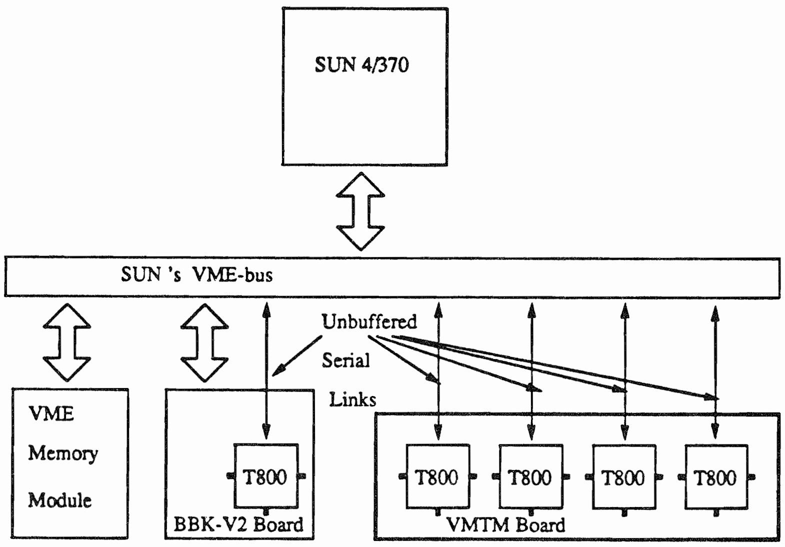


Figure 2: Architecture of the Parsytech System.

must wait if data is not available on a *ReadLink* invocation, or if the data has not been absorbed by the transputer on a *WriteLink* invocation.

Serial link interfaces may either be "dumb" (so the interface must be polled by the host for the "transfer possible" condition which is waiting for data on a read or waiting for the data to be removed on a write), or "interrupting" (so require a process scheduler to block the invoking process until the desired condition is true). Some VME/transputer interfaces can only be polled, while others have interrupt capabilities. Polling in a busy-wait loop has the advantage that the invoking process on the UNIX host will be immediately able to continue, giving optimal performance for the transfer rate across the device interface. But the host's cpu will be busy-waiting on the device leading to intolerable performance degradation for other work on a time-shared system. The solution is to increase the polling interval, but this reduces the performance of the interface, because of the delay in recognizing that the interface "transfer possible" condition is true and the overhead in process context switching (detailed in [Ousterhout 1990]).

Interrupting interfaces solve the busy-wait problem, but they are more complex, and the host kernel must be reconfigured to provide access to the interrupt handlers. Furthermore, as for periodic polling, process context switching will degrade the performance.

1.2.3 *Use of Shared VME-Memory*

Our other system, shown in Figure 2, transfers data between a SUN4/370 and the transputers through two kinds of shared VME-memory: dual-ported memory on a transputer board, and a VME memory module. The Parsytech BBK-V2 board contains a 20MHz T800 transputer and 2MBytes of dual-ported memory with memory cycle time of 300 nsecs. One of the T800's unbuffered serial links is connected (through a C012 link adaptor) to the VME-bus. This is a "dumb" interface which must be polled by the SUN host, and is used for synchronisation purposes only, as described in Section 4.2. The T800's 3 other links are serially buffered according to the RS422 protocol and may be connected to a transputer network on VMTM boards. A separate 16MByte VME-memory board with memory cycle time of 250 nsecs can be accessed by both the SUN and the transputer on the BBK-V2. The performance of such data transfers is not limited by the transputer link transfer rates (to 1.7 MByte/sec).

To access the VME-memory, the user-level process on the SUN must map the VME-memory (either the dual-ported memory on the transputer board or the separate VME-memory board) into its address space. The SUN then accesses the VME-memory with simple assignment statements such as:

**dst = *src*

where *dst* points to the VME memory, and *src* points to the SUN's local memory.

The T800 transputer on the BBK board accesses the separate VME-memory through an absolute address located in a register, details of which are given in Section 4.3. However, using these methods there is no memory protection so programming the data transfers is very hazardous; any VME-memory can be corrupted.

In Section 2 we describe and measure the performance of unbuffered and buffered serial link interfaces which must be polled from the SUN host. Section 3 describes and gives the performance of a buffered interrupting serial interface. We discuss the performance of the shared VME-memory for data transfers and our solutions to the synchronisation problem in Section 4. Section 5 presents our conclusions.

2. *Use of Polled Transputer Links for Host/Transputer Data Transfer*

2.1 *Introduction*

Polling interfaces are divided into two kinds: unbuffered and buffered. An unbuffered interface provides only one byte of data at a time over the link, and a check for "transfer possible" must be made between every byte. A buffered interface (such as is used for discs) allows a whole buffer of data to be transferred at a time, so the overhead of checking for the "transfer possible" condition may be amortized over a larger transfer, thus reducing the overhead. Both types of interface are available for transputers, and we examined the performance of them both. The host user-level code and the transputer code are the same for both kinds of interfaces, and are described below.

2.2 *Host User-level Code*

The user-level interface for a UNIX process to a transputer link for polling is similar to the standard UNIX convention for referencing a FIFO byte stream. The user accesses the link on the host through invocations to special C-language i/o library routines: *OpenLink, WriteLink, ReadLink* as shown in Figure 3. These routines must be linked with the user's program to build a binary image, and they are described in detail in the sections following.

This code is used in a test program to measure the transmission rates between the SUN and the transputers. The test program consists of a UNIX process on the SUN3 or SUN4 which reads/writes large amounts of data to a transputer link in various size packets corresponding to the variable *cnt* in Figure 3. The packet sizes increase by powers of two, from 4 bytes/packet to 1 MByte/packet, where the packet size is encoded in the first 4 bytes of the data. The process then waits for an equal packet of data to be returned from the transputer. Note that no disc transfers are involved, only memory-memory transfers are made as the SUN memory is 4 or 8 MBytes, and the transputer memory is 2 MBytes. The complete code for the test program on the SUN is in Appendix A.

```
/* user-level C-code on SUN host */
int   chan,cnt,timeout,chars_sent,chars_rcvd;
char *buf;
chan = OpenLink(0);       /* opens first available channel */
  ...
chars_sent = WriteLink(chan, buf, cnt, timeout);
chars_rcvd = ReadLink (chan, buf, cnt, timeout);
  ...
```

Figure 3: User-level access on a SUN to a transputer link.

The implementation of the user-level C-language i/o library routines differs for buffered and unbuffered interfaces, as shown in Sections 2.4 and 2.5.

2.3 Transputer Code

The corresponding transputer code to transfer data between the host and itself is shown in Figure 4. The transputer repeatedly reads packets by decoding the first four bytes and then reading the remainder of the packet in a single gulp. The transputer then reflects the packet back to the SUN host.

The transputer i/o library routines *ChanIn* and *ChanOut* are linked with this code to form a binary image which is downloaded then run on the transputer.

2.4 Host's Unbuffered Polling Device Driver

2.4.1 Software Details

The INMOS B014 and the Parsytech VMTM transputers shown in Figure 2 use an unbuffered byte-serial link between the VME host (i.e. the SUN running UNIX) and the T800 nodes. The CSA Part.8 FIFO link shown in Figure 1 can also be treated as an unbuffered link by treating the FIFO as if it had a capacity of just one byte. The original software for the device interface of CSA's Part.8 board treats the transputer links in this way, i.e. as unbuffered serial links.

```
/* nrate.c (for transputers)
 * The transputer receives a message on LINK 0 then sends it back.
 * The length of the message is encoded in the first 4 bytes.
 */
unsigned char buf[1024*1024];   /* need transputers with more than
                                   1 MByte mem. */
main()
{
        int i,length;
        for (;;) {
                ChanIn(LINK0IN,buf,4);
                length = (int)buf[0];
                length = (length<<8) + (int)buf[1];
                length = (length<<8) + (int)buf[2];
                length = (length<<8) + (int)buf[3];
                if(length<4)      break;
                if(length>4)      ChanIn(LINK0IN,buf+4,length-4);
                ChanOut(LINK0OUT,buf,length);
        }
}
```

Figure 4: Transputer code for Link Transfers.

A polling driver can be invoked by the user through calls to i/o library routines such as *ReadLink* and *WriteLink*. The code for the i/o library routine, *WriteLink,* provided as the original software for polling the device interface of CSA's Part.8 board is shown in Figure 5. The data is transferred through a special memory location associated with the link—i.e. the structure pointed at by *lkb*. This style of busy-wait loop has minimum overhead, but the host cpu is 100% occupied. The code for *ReadLink* is similar.

2.4.2 *Performance of Unbuffered Device Driver*

The test program described in Section 2.1 and detailed in Appendix A was used to measure the performance of the link. All tests were run while the machine was quiescent.

The throughput rates for the original unbuffered polling device driver executing on a SUN3/110 host for the various packet sizes are given in columns 1 and 2 of Table 1, and illustrated in Figure 6. Column 1 is for the SUN code compiled without any optimizing

```
/* C i/o library code linked with the user-level C-code on the SUN */

# include "link.h"          /* decs. for status registers for each link */
                            /* Used by OpenLink, ReadLink etc. */
struct link_st *lkb;        /* link status and data register structure  */
                            /* initialised by a call of chan = OpenLink()*/
/*******************************************************
 * Procedure:  WriteLink
 * Description:  Writes data to output FIFO
 * Parameters:  chan, buf, cnt, timeout
 * Return Value:  Number of characters sent
 *******************************************************/

int WriteLink( chan, buf, cnt, timeout)
int chan;
char *buf;
unsigned int cnt, timeout;
{
        int chars_sent = 0;
        while (cnt-- ) {
                /* spin till transfer possible */
                /* implemented as: %while (LINK.OFull & 1) ; */
                while (transfer-not-possible) ;

                /* now able to transfer data */
                /* write a byte to the register */
                lkb->data_s[chan].bt[1] = *buf++;
                chars_sent++;
        } /* while */
        return(chars_sent);
}
```

Figure 5: Original Unbuffered Polling Driver.

options, and column 2 is for the same code compiled with the C-compiler's Level 2 optimizing option.

Although the optimizing compiler has improved the transfer rates from 100,000 bytes/second to 163,000 bytes/second, it appears that the SUN cpu-cycles are limiting the system throughput rather than the

Packet Size (bytes)	SUN3/110 (2 MIPS) original (UNOPT) (bytes/sec)	SUN3/110 (2 MIPS) original (OPT) (bytes/sec)	SUN3/110 (2 MIPS) best (OPT) (bytes/sec)	SUN4/110 (7.5 MIPS) best (OPT) (bytes/sec)
4	7272	6779	7142	25804
16	24241	24612	27112	88393
64	55652	68084	94116	147460
256	83116	121319	226550	356948
1024	94902	150137	356771	424504
4096	98319	160078	414423	450500
16384	99203	162560	431484	455887
65536	99438	162812	435070	457872
262144	98596	162560	433273	439179
1048576	99015	145222	388342	439179

Table 1: SUN-transputer throughput for Unbuffered Polling Device Drivers

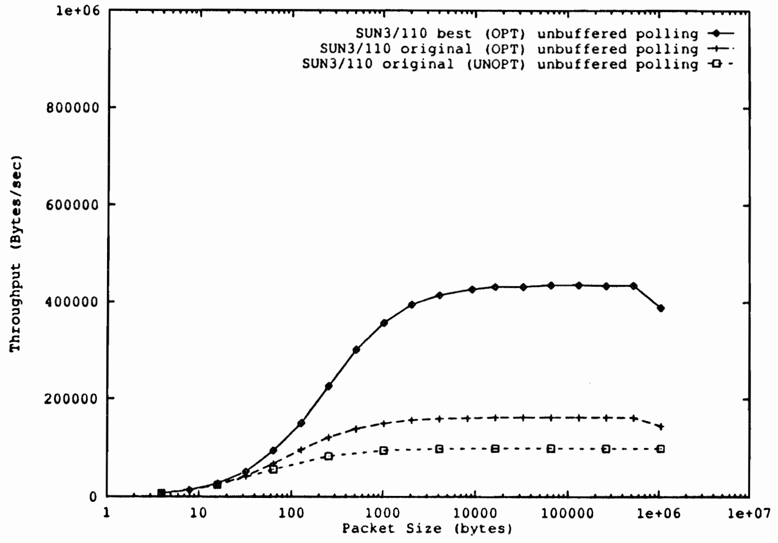


Figure 6: Unbuffered Polling Driver on SUN3/110.

interface or link hardware, which we expected to perform near the theoretical maximum of 1.8 MBytes/sec. We therefore experimented with the device driver code on the SUN3, and made a few obvious improvements to both the *ReadLink* and *WriteLink* routines. The first improvement adds a *register* variable *data* which is assigned to point to the special memory location associated with the link structure. Thus we changed the code of Figure 5 with the following:

```
register u_char *data;
   ...
data = (u_char*) &(lkb->data_s[chan].bt[1]);
   ...
*data = *buf++;
```

The new code improved the SUN-3 performance by 40%, from 163,000 bytes/sec to 229,000 bytes/sec.

A second change added a register variable to check for the "transfer possible" condition thus in *WriteLink:*

```
register u_char *empty;
   ...
empty = LINK.OEmpty;
   ...
while (*empty) ;              /* spin till transfer possible */
```

This code change increased throughput to a maximum of 405,000 bytes/second. We were surprised that the optimizing compiler had not detected this optimization, so we continued to "hand-tune" the code with a further small change, to alter the *int chars_sent* to *register int chars_sent*. This change increased the performance to a maximum of 435,000 bytes/second, which appears to be the limit for this byte-by-byte processing on the SUN3. This last result (best (OPT) unbuffered) is given in column 3 of Table 1 and plotted in Figure 6. As a check on the optimizing compiler we re-compiled this "best" code without the optimize option and were only able to achieve a throughput of 181,500 bytes/second. Hence all subsequent tests were performed only on the optimized code (OPT).

There is always a slight dip in performance on the SUN3 when the packet size reaches 1 Megabyte. The SUN3/110 does not have an internal cache so this dip cannot be attributed to the cache. To determine why the dip occurs we executed the test program with a substitution of code in the *ReadLink* and *WriteLink* routines so that access to the link interface was replaced by access to a temporary register variable. This test program also showed a similar dip in performance, indicating that the architecture of the SUN3/110 causes the performance dip. Our SUN3/110 has only 4 Megabytes of memory, and some paging may occur when running the test programs on SUN OS 4.1.

Next we executed the test program on a SUN4/110 with 8MBytes of memory attached to the same transputer (a CSA Part.6 and Part.8). Data is given in column 4 of Table 1, and is plotted together with the data from column 3 of Table 1 for the SUN3 as the lowest two lines in Figure 7. Although the faster SUN obtains slightly higher transfer rates with the same hardware, the increase is not linear with the MIPs ratings possibly because RISC architectures have allowed cpu speed to scale much faster than memory bandwidth, with the result that memory-intensive benchmarks do not receive the full benefit of faster cpu's [Ousterhout 1990].

So the combination of these three changes more than doubles the transfer rate between a SUN3/110 and a single transputer: from 163,000 to 435,000 bytes/sec. This is still very much less than is possible over a transputer-transputer link (1.7 MBytes/sec). CSA designed a buffered FIFO VME-interface (the Part.8) to improve the transfer rates.

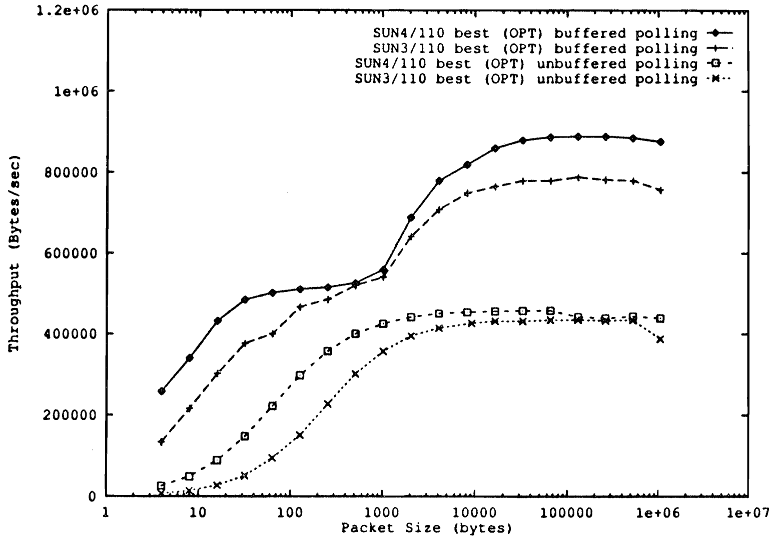


Figure 7: Unbuffered and Buffered Polling Drivers.

2.5 *Host's Buffered Polling Device Driver*

2.5.1 *Software Details*

The SUN/transputer buffered link hardware is shown in Figure 1. The size of the packets transferred can be increased up to the maximum buffer size (FIFO_SIZE). The buffered device has extra status registers detailed in Appendix A to allow block transfers. The *WriteLink* routine in the driver can test whether the FIFO is empty (LINK.OEmpty is TRUE), and if so, it can write the full FIFO_SIZE of data to the data register, without extra checking. Similarly, the *ReadLink* routine can check if the FIFO is full (LINK.IFull is TRUE) and if so, it can read the whole FIFO_SIZE of characters from the data register without further checking. We wrote new library routines *ReadLink* and *WriteLink* which divided large packets of data into complete FIFO_SIZE batches and transferred a complete FIFO_SIZE of data at a time, except for the last packet. Code implementing these semantics for *WriteLink* is shown in Figure 8.

2.5.2 Performance of Buffered Polling Device Driver

The same test program described in Section 2.1 was run, using the new library to transmit a FIFO-full of data at a time whenever possible. On the SUN3 the data is transferred much faster this way, peaking at 773,800 bytes/second as shown in column 1 of Table 2. The data show that performance levels off for packets approaching the FIFO size of 1024, so we experimented to see if the FIFO size of 1024 was a limitation to throughput, by adding checks to see if the buffer was half full, and transferring a half-buffer at a time instead of a full buffer. The results are shown in column 2 of Table 2, and are plotted in Figure 7 as best (OPT) buffered polling. Both of these show the benefits of testing the FIFO once for several FIFO accesses. The reasons are explained below.

Suppose FIFO accessing time, *access*, and FIFO testing time, *test*, are the same, (say, writing one byte to the FIFO takes one microsec, and testing the FIFO full or not takes one microsec). Then the throughput usage for "1 test for every access" (unbuffered mode) is

$$(access) / (access + test) = 50\%$$

The throughput usage for "1 test every 512 accesses" mode is

$$(512{*}access) / (512{*}access + test) = 99.8\%$$

The throughput usage for "1 test for every 1024 accesses" mode is

$$(1024{*}access) / (1024{*}access + test) = 99.9\%.$$

So for these relative access and test times, a FIFO size of even 64 bytes would provide a good throughput (99.5%) for packets larger than the FIFO size. For any transfer less than the FIFO size (i.e. the

```
/**************************************************
 * Procedure:  WriteLink
 * Description:  Writes data to output FIFO
 * Parameters:  chan, buf, cnt, Timeout
 * Return Value:  Number of characters sent
 **************************************************/
int WriteLink(chan, buf, cnt, Timeout)
int chan;
register u_char *buf;
register int cnt;
unsigned int Timeout;
{
        register int ncnt;
        register int chars_sent = 0, tmp=0;
        register u_char *data,*empty;

        ncnt = cnt;
        data = &(lkb->data_s[chan].bt[1]);
        empty = &(LINK.OEmpty);
        while (ncnt >= FIFO_SIZE )        /* send a FIFO-full if poss. */
          {
                 tmp += FIFO_SIZE;
                /* spin till LINK is empty, then fill it in one gulp */
                while ((*empty & 1) == 0) ;
                for ( ; chars_sent < tmp; chars_sent++)
                                *data = *buf++;
                   ncnt -= FIFO_SIZE;
          }
        chars_sent += ncnt;
        /* send the remaining bytes (< FIFO_SIZE of them) */
        /* when the FIFO is empty */
        while (ncnt>0 && (*empty & 1) == 0);
        while (ncnt-- > 0)
               *data = *buf++;
        return(chars_sent);
}
```

Figure 8: Buffered Polling Driver.

Packet Size (bytes)	SUN3/110 (FIFO-full) (bytes/sec)	SUN3/110 (half FIFO-full) (bytes/sec)	SUN4/110 best (OPT) (bytes/sec)
4	133335	133333	258065
16	266671	301850	432433
64	400002	399966	501956
256	474066	485421	515250
1024	519760	540465	559216
4096	633527	708465	779587
16384	733198	765347	859435
65536	768107	779536	886715
262144	773817	782475	888593
1048576	703709	757057	875608

Table 2: SUN-transputer throughput for Buffered Polling Device Driver

last transfer of a large packet), the test-every-access mode is required. Transferring a half-buffer at a time speeds up the transfer rates for large packet sizes, because the waiting time for the opposite end is reduced—the transfer can start after only half the FIFO is filled.

We still do not know where the throughput bottleneck is: are the SUN3 cpu cycles still limiting the throughput (to 782,500 bytes/sec), or is the link hardware saturated? We therefore ran the same test programs with the buffered driver on a SUN4/110. The results are in column 3 of Table 2 and are plotted in Figure 7. The SUN4/110 transfer rates peaked at 888,600 bytes/sec, a little faster than the SUN3. These data indicate that the SUN's cpu cycles are not limiting the transfer rate; other factors must now be the bottleneck in achieving still higher throughput. It would be helpful to use block transfers over the VME-bus, but the hardware interfaces do not permit this.

3. Use of Interrupt-Driven Links for Host/Transputer Data Transfer

3.1 Host User-level Coae

The CSA Part.8 is capable of interrupting the SUN host, and to overcome the high cpu usage associated with busy-waiting in the polling driver on the SUN, we wrote an interrupt device driver for the Part.8.

For SUN/UNIX interrupt-driven i/o devices, the user-level interface on the host computer uses the standard UNIX i/o routines *read/write*. The device is an interrupt-driven buffered serial link (a channel) configured with the special name "/dev/tlr0" for the read end and "/dev/tlw0" for the write end. Note that links beyond the first are called "/dev/tlr1..n", "/dev/tlw1..n". The user first opens the device for reading and writing, then accesses it using C-code as in Figure 9. Only the standard C i/o library needs to be linked with the user's code, as the device interface driver is configured with the UNIX kernel at boot time.

```
char chan_r[20], chan_w[20];
int chars_sent, chars_rcvd, length, tr,tw;
char *buf;
...
sprintf(chan_r,"/dev/tlr0");
sprintf(chan_w,"/dev/tlw0");
...
tr = open(chan_r,O_RDONLY);
tw = open(chan_w,O_WRONLY);
...
chars_sent = write(tw,buf,length);
chars_rcvd = read (tr,buf,length);
```

Figure 9: User-level code with an interrupt handler.

3.2 Transputer Code

The transputer code is the same as that for the polling driver, given in Figure 4.

3.3 Host's Interrupt-Driven Device Driver

3.3.1 Software Details

The interrupt hardware provides several interrupt conditions such as "interrupt on empty" and "interrupt on half full." The device driver, an interrupt handler, is complex and contains low-level device-dependent code, written according to the specifications in the SUN documentation [SUN 1990]. As we designed the interrupt handler after our experiments with the polling driver, we were able to implement the most efficient version of the polling code for the device driver, based on code which transmits up to half a FIFO-full of data at a time.

3.4 Performance of the Interrupt Driver

Performance of the interrupt driver for the SUN3 and the SUN4 is given in columns 1 and 2 of Table 3. Column 1 is for a SUN3/110 host, and column 2 is for a SUN4/110 host to the same transputers configured as shown in Figure 1. Figure 10 shows a comparison of the polling and interrupt performance on the SUN3/110.

Packet Size (bytes)	SUN3/110 (2 MIPS) (bytes/sec)	SUN4/110 (7.5 MIPS) (bytes/sec)
4	2806	6055
16	10973	23356
64	40274	81214
256	121550	213411
1024	357548	547888
4096	609601	775244
16384	704877	806550
65536	747593	829684
262144	751612	844723
1048576	729392	843024

Table 3: SUN-transputer throughput for Interrupt Device Driver

As expected, the polling driver performs better than the interrupt driver over all packet sizes, although for packet sizes of 4096 or larger, the interrupt-driven performance approaches within 97% of the polling driver. This result shows the effects of interrupt latency and the high context switching overhead. For small packet sizes, the interrupt driver's performance is poor because of frequent context switching which has a cost of 2.4 ms on a SUN3 and 1 ms for a SUN4 [Ousterhout 1990]. For larger packets this overhead is spread over more data, leading to the increased data rates observed. As expected, the

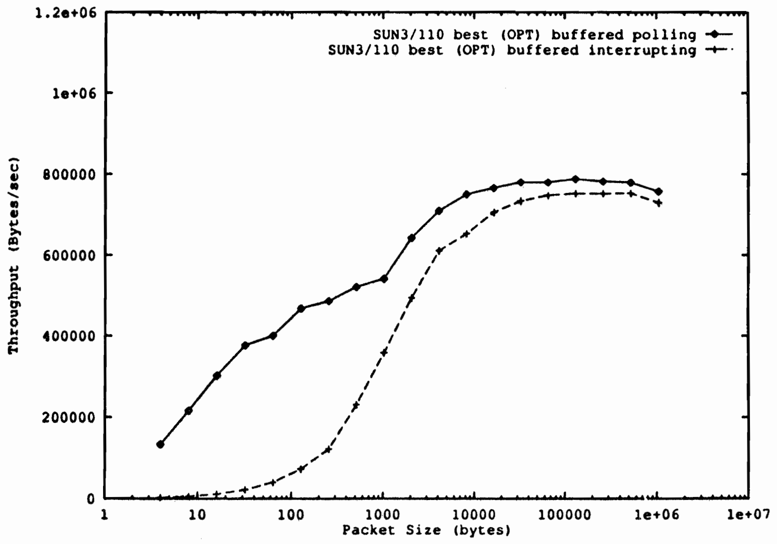


Figure 10: Sun-transputer throughput for Interrupt and Buffered Polling Drivers.

SUN4 performs a little better than the SUN3, as shown in Figure 11 where the data in Table 3 is plotted. These results all show that the serial link transfer rates have reached their peak and no further gains can be expected without a DMA interface.

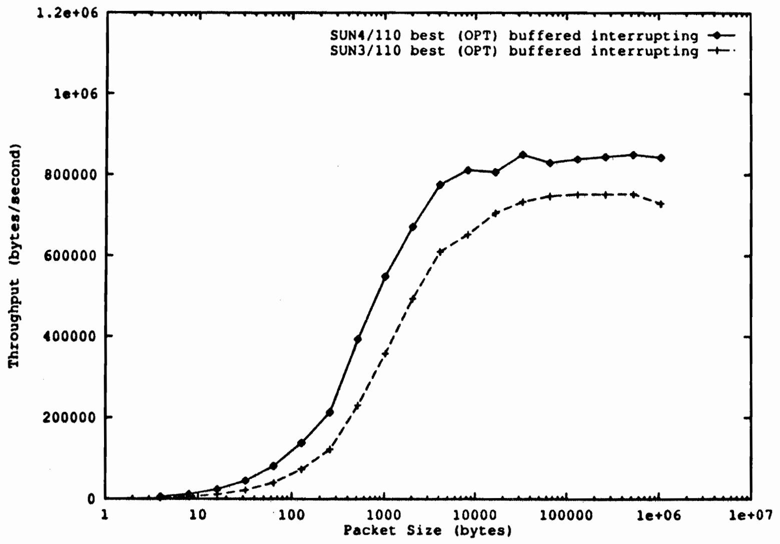


Figure 11: Comparison of SUN3 and SUN4 Interrupt Handlers.

4. *Use of Shared Memory for Host/Transputer Data Transfer*

We explored different ways of using a shared memory interface (configured as in Figure 2) for data exchange between the host and the transputer network, in order to maximize the SUN/transputer data transfer rates.

There are two distinct types of memory that both the Sun and the transputer T800 located on the BBK-V2 can access: the 2 MByte dual-ported RAM situated on the BBK-V2 and any other memory extension board installed in the free VME address space of the system (in this particular case a 16MByte module).

As stated in Section 1.2.3, using shared memory to transfer data between the SUN and the transputers suffers from the disadvantage that there is no built-in security—users can easily corrupt the memory in the VME-address space. Also the synchronisation has to be programmed explicitly. However, we felt that this was still the best way to increase the SUN/transputer transfer rate.

Several C-language constructs were used in the SUN and in the transputer to move single words and blocks of data from one memory to another and the transfer rates were measured.

4.1 Host User-level Code

The host can perform only 4-byte accesses to the VME memory. We timed the READ and WRITE access time and the transfer time between the SUN and the two types of memory used (dual-ported RAM on the BBK-V2 and a 16MByte shared memory module).

The memory zones are allocated from the SUN using standard C-language storage allocation functions. The user-level process on the SUN uses memory mapping to access the shared VME-memory, which is split into two contiguous zones described by the *ram* structure in Figure 12. The user first opens the device for reading and writing, then accesses it using C-code as in Figure 12.

Synchronization of the transfer is achieved by using mailbox locations situated in the dual-ported memory of the BBK-V2 so the transputer doesn't poll over the VME bus. The host however has to poll across VME both on a shared memory location and on the link-

```
/* user-level C-code on SUN host */
/* shows copying a single integer from the SUN */
/* to a random VME memory location */
#include <sys/mman.h>
typedef struct {
        unsigned char dram[0x200000];   /* 2 MByte dual-ported mem. */
                                        /* on transputer board */
        unsigned char vme[0x1000000];   /*  16 MByte VME-memory */
} RAM;
RAM *ram;
...
register int local=54321;           /* an int. in SUN's local memory */
register int *bufptr_dram,*bufptr_vme;
...
/* Ugly code to init. the memory map */
/* using mmap args specified in SUN manual */
fd = open ("/dev/vme32d32",O_RDWR)
sadr = (caddr_t) 0;
len = 0x1200000;                        /* total memory space */
/* physical address of the BBK-V2 board */
offset = (off_t)0x80e00000;
ram=(RAM *) mmap(sadr, len, PROT_READ| PROT_WRITE,
                    MAP_SHARED, fd, offset);
...
bufptr_dram = (register int)&(ram->dram[0]);        /* dual-ported memory */
bufptr_vme = (register int)&(ram->vme[0]);          /* VME-memory */
...
/* data is moved from local SUN memory to VME-memory */
*bufptr_vme = local;
```
Figure 12: User-level code for transfers to VME-memory.

adaptor dedicated to the CIO server (CIO), which is a UNIX process always present on the host to receive I/O requests (status,results, errors) from the transputer on a link-adaptor shown in Figure 2.

There are two different situations: when the transfer is executed by the SUN's cpu and when the transfer is executed by the transputer.

1. The activity on the VME bus and on the transputer link is shown in Figure 13, for the case in which the SUN executes the transfer and the transputer measures the time. In the host the

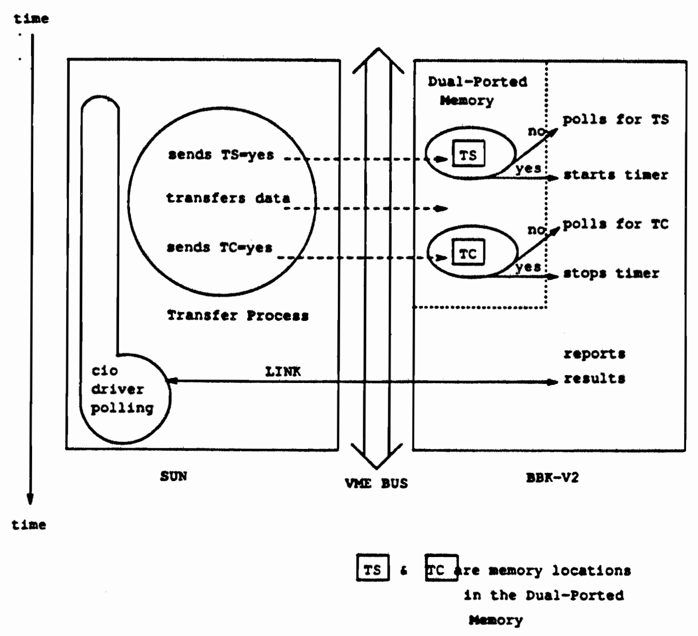


Figure 13: SUN to Dual-ported Memory Transfer Synchronisation.

link-server(CIO) polls in the background. The Transfer Process, started in the SUN after the transputer receives its bootstrap code, sets a memory TS="Transfer Started" location which signals the beginning of the transfer to the transputer. When this message is received the transputer starts to time the transfer and also polls the TC="Transfer Completed" on-board memory location to see if the SUN has finished transferring the data. When the "Transfer Completed" message is received, the timer is stopped and the results are sent on the link to the SUN via the link-server(CIO).

2. Figure 14 reflects the case of the transfer being executed by the transputer. After downloading the code, the SUN waits for the transfer to start by polling across the VME bus the TS="Transfer Started" condition. Once the location "Transfer Started" is set by the transputer, the SUN starts timing and polling over the VME bus into the dual-ported memory for the TC="Transfer Completed" condition. Although polling on the

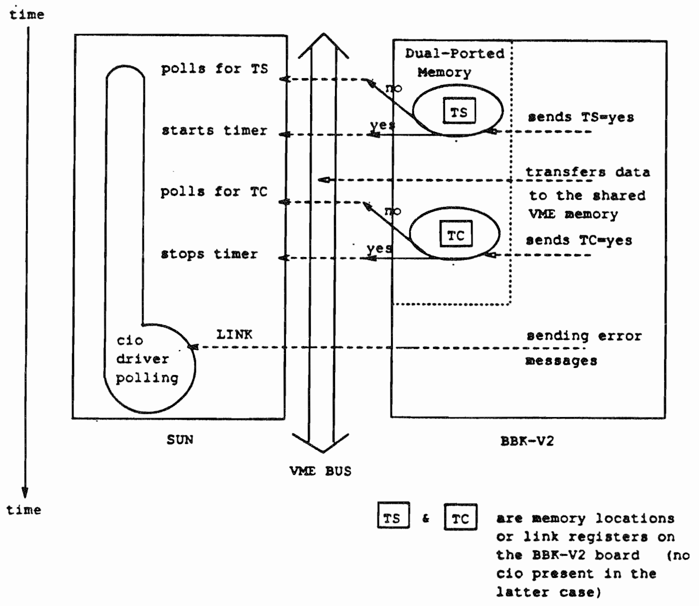


Figure 14: Dual-ported Memory to SUN Transfer Synchronisation.

VME bus is undesirable, it is the only way to achieve synchronization in this case. The CIO is available for debugging purposes.

4.2 Transputer Code

The same tests (i.e. reading and writing to the dual-ported memory and to the shared VME memory) were performed by the BBK-V2 module. The random 4-byte access rates (presented in Section 4.4) are very small compared to the SUN's access rates. Therefore we tried to find a way to improve them, by performing single instruction block transfers between the transputer memory and the VME shared memory, as shown in Figure 15. We used the transputer C-library *bcopy* for copying a memory region from one base location to another.

The assembler code for *bcopy* is presented in Figure 16. The transputer assembler instruction *move* is used by *bcopy*. The first three *ldl* (load local) instructions load the operands for the *move* instruction into registers from the storage area for local variables of which the first sixteen locations can be accessed using a single byte instruction. The

```
/* Transputer C-code to copy a block using bcopy */
#include<string.h>
/* hardware address for Address Register R3 */
#define R3_addr 0x80060000
/* physical address of shared VME RAM */
#define vme_ram_start    0x81000000
...
register int *src_dram,*dst_dram; /* source and dest. pointers in DRAM*/
register int *src_vme,*dst_vme;   /* pointers in shared VME RAM*/
int       size;                   /* size of the arrays in words */
...
R3_ptr = (int *)R3_addr;          /* R3 is initialized */
*R3_ptr = vme_ram_start;
...
src_dram = (register int *)calloc(size,sizeof(int));   /* arrays in DRAM */
dst_dram = (register int *)calloc(size,sizeof(int));
src_vme = (register int *)(vme_ram_start);             /* shared VME RAM */
dst_vme = (register int *)(vme_ram_start + size*sizeof(int));
...
bcopy(src_dram, dst_vme, size*sizeof(int));
...
```

Figure 15: Transputer code for block transfers to VME memory.

```
#include<string.h>
void    *bcopy(src, dst, size)
        const   void    *src;
        void    *dst;
        size_t  size;
{
        ldl     1                   ;src
        ldl     2                   ;dst
        ldl     3                   ;size
        move
        ldl     2                   ;dst
}
```

Figure 16: Source code for bcopy.

transfer is performed by *move* in 2*w+8 processor cycles (T800), where w is the size of the block to be copied, in bytes. Each *load_local* instruction takes 2 processor cycles. The last *load_local* returns the *bcopy* function value, a pointer equal to the start address of the destination zone.

With the start address of the BBK-V2 board selected to be in the VME A32/D32 range, and the access mode set to 32BA (32 Bit Access), the on-board RAM is followed by an extension memory zone up to 512 MBytes. VME memory extension modules can have a variable base address anywhere in this space. The T800 transputer on the BBK-V2 can dynamically select different modules by setting the address register (R3).

The transputer code is shown in Figure 15. The arrays used for transfer situated in the transputer DRAM have been allocated from the heap segment using standard C storage allocation functions. Arrays in the shared VME RAM are accessed by an appropriate value in address register R3.

4.3 Performance of transfers using VME-memory

We refer to the separate memory extension board as shared VME RAM and to the BBK-V2 on-board memory as dual-ported RAM. Table 4 shows measured 4-byte transfer rates between the SUN4, the BBK-V2 transputer and the VME-memory.

Code Style	SUN-4 to/from shared VME RAM (bytes/sec)	SUN-4 to/from dual-ported RAM (bytes/sec)	BBK-V2 trans. to/from shared VME RAM (bytes/sec)
*dst_vme++=local_const	4,474,000	3,735,000	702,200
*dst_local=*src_vme++	3,907,000	3,730,000	606,700
*dst_vme++=*src_local++	3,231,000	2,813,000	520,800
*dst_local++=*src_vme++	2,853,000	2,795,000	532,600

Table 4: SUN-transputer throughput for 4-byte Transfers to the VME-memory

The pointers indexed "_vme" refer to the memory space involved. Thus, the code styles correspond to those presented in Figure 17.

```
*dst_vme++ = local_const;        /* WRITE access to VME memory */
*dst_local = *src_vme++;         /* READ access to VME memory  */
*dst_vme++ = *src_local++;      /* (SUN-4,BBK-V2) to VME memory */
*dst_local++ = *src_vme++;      /* VME memory to (SUN-4,BBK-V2) */
```

Figure 17: Source code for word by word transfers.

Rows 1 and 2 of Table 4 show that the SUN and the transputer can write data into the VME-memory board faster than they can read the data back out, in part because of the difference between the READ and the WRITE access time of the memory circuits (for example, in the case of the shared VME memory, the manufacturer's specified READ access time is 190 ns and the WRITE access time is 50 ns).

Columns 1 and 2 of the first row contain transfer data between the SUN4 and VME-memory using a constant value to be written into the VME-memory, coded as shown in Figure 17. Comparing with row 3 shows that the software overhead in the SUN4 to update the destination address penalises the performance by 25%, as throughputs of up to 4.5 MBytes/sec can be achieved if the SUN merely writes constants into the VME-memory. However, the more realistic situation demands some kind of loop control overhead whereby both source and destination addresses are incremented for each 4-byte transfer.

The 32-bit wide data rates between the SUN and the VME-memory are about 4 times the rates observed over the byte-wide serial link described in Section 3, showing that in the SUN the VME-bus access mode is the bottleneck.

The performance guaranteed by the manufacturers for the two memory modules can account for the results in column 1 and 2 of Table 4; as the access rates to the dual-ported RAM (300 ns cycle time) are lower than those to the shared VME RAM (250 ns cycle time).

The random 4-byte access rates from the transputer to the shared VME RAM (column 3) are very slow compared to the other transfer rates. The relatively low data rates for the BBK transputer transfers also show that pointer manipulation in the transputer has a relatively high overhead. Buffering the data in the dual-ported RAM and using block transfers to the shared RAM proved to be more efficient, as shown in Table 5.

Table 5 shows the measured rates for the BBK-V2 to/from the shared VME-memory module transfers using the *bcopy* C-library function for different size blocks of data.

The transfer rate of 446,000 bytes/sec for a package size of 4 bytes is 14.3% slower than the rate of 520,080 bytes/sec in the case of using the simple addressing mode (presented in Table 4, column 3, row 3). These results discourage the use of the *bcopy* function for small block sizes (<16 bytes). Above packet sizes of 256 KBytes, the performance exceeds that for the SUN, because of the loop overhead in the SUN's word by word transfers.

Packet Size (bytes)	Dual-ported RAM to shared VME RAM (bytes/sec)	Shared VME RAM to dual-ported RAM (bytes/sec)
4	446,000	446,000
16	1,250,000	1,315,000
64	2,500,000	2,325,000
256	3,225,000	2,877,00
1024	3,455,000	3,053,000
4096	3,522,000	3,106,000
16384	3,542,000	3,121,000
65536	3,548,000	3,129,000
262144	3,548,000	3,130,000

Table 5: Transputer-Memory Throughput for Variable Size Transfers using bcopy

One disturbing feature is that if there is any other traffic on the VME-bus, all these rates drop off considerably so the data given are for a quiescent system with only one user. For the data acquired in a system with other users active, relative to the single-user measurements, the variance is 50%, too high an error for a correct interpretation. For different trials on the single-user standalone system the variance was 2%.

5. Summary

By carefully timing the data transfer rates between various SUN hosts and a single transputer, we have shown that several factors affect the performance.

1. The code must be well-written so as to make best use of pointers in registers. Poorly written code can degrade performance by almost 500%.
2. The best possible performance from a single byte (unbuffered) user-level interface is limited by the SUN cpu speed, as there is a high operating system overhead, to around 435,000 bytes/second on a SUN3/110 and 456,000 bytes/second on a SUN4/110.
3. Block transfers through a FIFO hardware buffer can almost double the performance—to 783,000 bytes/second on a SUN3/110 and to 888,500 bytes/second on a SUN4/110.
4. Interrupt-driven code can approach within 97% of the polling driven interface, with the advantage of reduced cpu usage.
5. 32-bit wide data transfers using shared memory on the VME-bus improves throughput by four times over the byte-wide serial links, showing that the VME-bus access mode is a performance bottleneck.
6. Further increases in performance are only possible if the SUN/VME interface allows VME "block transfer" mode; i.e. more powerful SUNs will not increase performance further.

Acknowledgements

Thanks are due to Joel Rogers at TRIUMF for patiently reading earlier drafts of this paper, and to the staff at Simon Fraser University, especially John Ens, for helping to reconfigure the hardware for the timing tests.

References

M. S. Atkins and Y. Chen, Performance of Sun-Transputer Interfaces: some surprises, *Proceedings of Transputing '91 Conference, Sunnyvale, CA*. IOS Press, pp 124–138 (April 1991).

M. S. Atkins, D. Murray and R. Harrop, Use of Transputers in a 3-D Positron Emission Tomograph, *IEEE Transactions on Medical Imaging* (Sept. 1991).

A. G. Barrett and B. Sufrin, Formal Support for Distributed Systems: occam and the Transputer, *Proceedings of Transputing '91 Conference, Sunnyvale, CA*. IOS Press, pp 388–405 (April 1991).

CSA, Part.8 and Part.6 Users' Manual, Computer System Architects, Provo, Utah USA 94604 (June 89).

INMOS, The Transputer Data Book (1987).

Logical Systems, Transputer Toolset Version 88.3 (June 1988).

D. Murray, A Real-Time Transputer-based Data Acquisition System, *MSc Thesis, School of Computing Science, Simon Fraser University, Canada* (April 1990).

J. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast As Hardware? *USENIX Summer Conference, Anaheim, CA*. pp 247–256 (June 1990).

Parsytech, BBK and VMTM transputer boards Users' Manual, Parsytech, W. Germany (1988).

J. G. Rogers, R. Harrop, G. H. Coombes, N. A. Wilkinson, M. S. Atkins, B. D. Pate, K. S. Morrison, M. Stazyk, C. J. Dykstra, J. S. Barney, P. W. Doherty, and D. P. Saylor, "Design of a Volume-Imaging Positron Emission Tomograph," *IEEE Transactions of Nuclear Science*, 36(1), pp. 993–997, Feb. 1989.

J. G. Rogers, M. Stazyk, R. Harrop, C. J. Dykstra, J. S. Barney, M. S. Atkins, and P. E. Kinahan, "Towards the Design of a Positron Volume Imaging (PVI) Camera," *IEEE Transactions on Nuclear Science*, 37(2), pp. 789–794, April 1990.

SUN microsystems, Writing Device Drivers, *Revision A of SUN microsystems Reference Manual* (March 1990).

N. A. Wilkinson, M. S. Atkins, and J. G. Rogers, A Real Time Parallel Processing Data Acquisition System, *Proceedings IEEE 9th Real-Time Systems Symposium,* pp. 54–59 (Dec. 1988).

[submitted May 17, 1991; revised Oct. 28, 1991; accepted Jan. 8, 1992]

Appendix A: Host Code for the Test Program

```
/*
 * read_write tester for the SUN/transputer interface
 * To be used with nrate on the transputers
*/
#include<stdio.h>                 /* Standard include
                                     file */
#include<ctype.h>                 /* Character
                                classification stuff */
#include<string.h>                /* String functions */
#include<sys/types.h>
#include<sys/timeb.h>
#include<sys/time.h>
#include        "link.h"

#ifdef sun
#include        <fcntl.h>
#include<sys/file.h>
#include<sys/signal.h>
#endif

#define DEF_DEC_TOUT  1000L   /* 1 sec load decode
                                 timeout * default */
#define MAX_DEC_TOUT  20000L  /* 20 secs maximum decode
                                 timeout */
#define MIN_DEC_TOUT  500L    /* .5 secs minimum decode
                                 timeout */
#define DEF_LVL_TOUT  500L   /* .5 sec/level timeout
                                default */
#define MAX_LVL_TOUT  1000L  /* 1 secs/level maximum
                                timeout */
#define MIN_LVL_TOUT  25L    /* .025 secs/level minimum
                                timeout */

#define MAX_LINKS  4               /* Max # of
                                      links/processor */
#define MAX_NODES  1000            /* Maximum user node # */

#define MAX_PACKET 255             /* Longest packet
                                      with this driver */
```

```
#define FROM_SYSTEM_RESET  0x01   /* Boot is from
                                     system chain */
#define FROM_SUBSYS_RESET  0x02   /* Boot is from
                                     sub-system chain */
#define MAX_LENGTH  1024*1024
extern int chan;
int    LinkId = 0;  /* LinkId from "OpenLink" */
/*
 * read_write.c
 * read and write transputer channels
 * Usage:
 * read_write [0/1/2/3] [msg_length] [trials]
 */

u_char buf[MAX_LENGTH];

main(argc,argv)
int argc;
char *argv[];
{
register int i;
int length,trials,j,num_chan,child;
char chan_r[20];
char chan_w[20];
struct timeval tp1,tp2;
struct timezone tz1,tz2;
long s_start, us_start, s_stop, us_stop,
     s_time, us_time;
float t;
unsigned int cnt, ncnt, Timeout;

if(argc>1){
         num_chan= atoi(argv[1]);
}
else{
         num_chan= 1;
}
if(argc>2){
        length = atoi(argv[2]);
        if(length > MAX_LENGTH){
                  fprintf(stderr,
                         "msg too long, max=%d0,
                          MAX_LENGTH);
                  exit(0);
        }
}
else
        length = 1024;
        if(argc>3){
                trials = atoi(argv[3]);
                if(trials<1){
                       fprintf(stderr,"error:
                       trials < 00); exit(0);
                }
        }
        else
                trials = 1000;
        LinkId = OpenLink(0);   /* Open specified link
                                   channel */
        if (LinkId<1) {
                 fprintf(stderr, "Unable to open link0);
                 exit(0);
        }
for(length = 4; length< 1050000; length = length*2)
  {
        trials = 2*1048576/length;
        if(trials > 1000)
                  trials = 1000;
        buf[3] = (u_char)(length & 0xff);
        buf[2] = (u_char)((length>>8) & 0xff);
        buf[1] = (u_char)((length>>16) & 0xff);
        buf[0] = (u_char)((length>>24) & 0xff);
        gettimeofday(&tp1, &tz1);
        s_start = tp1.tv_sec;
        us_start = tp1.tv_usec;
        for (i=0; i<trials; i++) {
                  WriteLink(LinkId, buf, length,
                  MAX_DEC_TOUT);
                  ReadLink(LinkId, buf, length,
                  MAX_DEC_TOUT);
                  }
        gettimeofday(&tp2, &tz2);
        s_stop = tp2.tv_sec;
        us_stop= tp2.tv_usec;
        s_time = (us_stop<us_start) ? (s_stop - s_start-1)
                                    : (s_stop-s_start);
        us_time= (us_stop<us_start)
                       ? (1000000 + us_stop - us_start)
                         : (us_stop - us_start);
        t = (float) (s_time) + (10e-7)*(float) (us_time);
        printf("%%d0,(int)((float)(length*trials*2)/t));
  }
}
```