Architectures, Languages and Patterns
P.H. Welch and A.W.P. Bakkers (Ed.)
IOS Press, 1998

A PCI based Network Interface Controller for IEEE 1355 DS Links

M. Boosten[1,2,3] R.W. Dobinson[1] B. Martin[1] P.D.V. van der Stok[2]
[1] *CERN, The European Laboratory for Particle Physics, Geneva, Switzerland*
[2] *Eindhoven University of Technology, Eindhoven, The Netherlands*
[3] *CERN, 513-R060, CH-1211 Geneva 23, Switzerland, +41-22-767-8705, Marcel.Boosten@cern.ch*

Abstract. We have investigated the construction of a parallel computer using IEEE 1355 high-throughput low-latency DS link networks and high-performance commodity processors running a standard operating system. In this context a DS Network Interface Controller (DSNIC) has been developed. The board's hardware, controlled by FPGA firmware, together with host software, provides a CSP based message passing interface between standard OS processes. This paper describes how the design and realisation of the DSNIC reflect our aim: low-latency high-throughput inter-process communication. We show the benchmark results, their analysis, and suggest further performance gains that might be possible.

1 Introduction

CERN is currently building a new accelerator to allow the study of collisions between elementary particles at very high energies. A major problem will be the huge amounts of data produced during a collision combined, with the frequency at which collisions occur. For example, the ATLAS[2] experiment requires a computer system to perform real-time analysis at the rate of 10 Gbytes/s on this data.

The use of high-performance commodity processors, like the DEC Alpha, Power PC, or Pentium, to perform the real-time analysis is interesting because of their very good price/performance ratio. Apart from computing power, high-performance communication between the computing nodes is also essential. In this context, a DS Network Interface Controller (DSNIC) has been developed. The DSNIC has allowed us to investigate building a parallel computer using high-throughput low-latency IEEE 1355 DS links[10] and 200 MHz Pentium Pro PCs running the Linux 2.0.27 Operating System (OS). Building a high-performance parallel computer means aiming at reliable low-latency high-throughput communication that generates little CPU load. In this paper we describe how the design and realisation of the DSNIC reflect the performance aim. This work has been carried out as part of the EU funded ARCHES Esprit project[1].

2 Switched IEEE 1355 DS link networks

IEEE 1355 DS links are bidirectional flow-controlled point-to-point serial links running at 100 Mbaud. DS link networks can be built using the 32 port SGS Thompson C104 packet switch[17]. This chip uses wormhole routing to route DS link packets; preloaded routing tables determine the packet's destination port on the basis of the packet header, as soon as this header has been received by the source port. The C104 does *not* use a bus based architecture, instead a non-blocking crossbar is implemented to route packets. It has a switching latency of 1 μs.

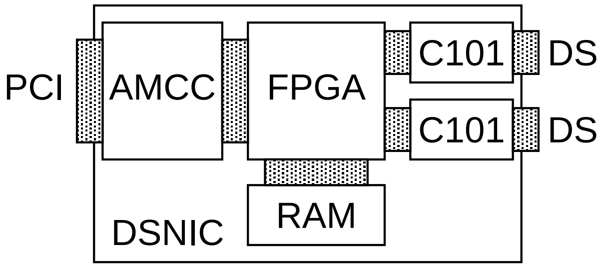

Figure 1: The DSNIC board.

As part of the Macramé Esprit project, CERN has constructed a very large network testbed[21] based on DS links and C104 switches. This has allowed the investigation of latency and throughput for many different traffic patterns and network topologies up to 1024 terminal-nodes. The Macramé network has demonstrated that large DS link networks can be built, that they scale very well, and that these networks can cope with the high data rates required for the ATLAS experiment[5]. Furthermore, it has proven that the per-link flow control, together with well designed hardware[13], can result in very reliable systems: a maximum Bit Error Rate (BER) of 9.6×10^{-18}[21] has been reported.

3 DSNIC board

A PCI based DSNIC board[3] has been developed jointly between CERN and INFN[11]. The wide acceptance of the PCI bus standard allows the board to support many present and future processors.

The board, see Figure 1, contains the AMCC 5933 PCI interface chip, two SGS Thompson C101 parallel to serial DS link interfaces[16], 256 Kbytes RAM, and the Altera FLEX 10K50 Field-Programmable Gate Array (FPGA). The spare pins of the FPGA are brought out to headers to allow hardware extensions, e.g., via a daughter board. The flexibility of the FPGA allows part of the communication functionality to be off-loaded to the board; this functionality is thus spread over the board's firmware and the controlling software, i.e., the driver, executed by the host CPU.

4 Communication API

We base our parallel computer on Communicating Sequential Processes [8] (CSP). We use OS processes, i.e., Linux processes or threads, for the CSP processes. The CSP channels are virtual links between these processes. The Application Programming Interface (API), see Table 1, provides full CSP communication functionality, including facilities for active channel selection.

We add non-blocking communication to the blocking CSP interface, since non-blocking communication allows concurrent computation and communication without task switching. This improves the performance of the DSNIC, since task switching is a time consuming operation in general purpose OSs. The definition of blocking and non-blocking corresponds to the definition in the Message Passing Interface[12] (MPI) standard. In non-blocking communication, the send or receive call may return before the operation is completed, and before the user is allowed to re-use resources, such as buffers, specified in the call. In blocking communication, returning from a send or receive call indicates that the user is allowed to re-use resources specified in the call.

Table 1: The communication API.

Basic CSP constructs
`int Send(NI* ni,int vl,char* address,int length)`
`int Receive(NI* ni,int vl,char* address)`
`int WaitAndSelectFirst(NI* ni,int* vl_nrs,int size)`
`int WaitAndSelectRandom(NI* ni,int* vl_nrs,int size)`
`int SelectFirst(NI* ni,int* vl_nrs,int size)`
`int SelectRandom(NI* ni,int* vl_nrs,int size)`
Non blocking communication
`int StartSend(NI* ni,int vl,char* address,int length,Request* r)`
`int StartReceive(NI* ni,int vl,char* address,Request* r)`
`int Completed(NI* ni,Request* r)`
`int Complete(NI* ni,Request* r)`
Asynchronous communication and its non blocking variant
`int ASend(NI* ni,int vl,char* address,int length)`
`int StartASend(NI* ni,int vl,char* address,int length,Request* r)`
Message memory handling
`void* Allocate(NI* ni,int size)`
`void Free(NI* ni,void* memory)`
Initialisation and termination
`int Open(NI* ni,char* resource)`
`void Close(NI* ni)`
`void SetUpVirtualLink(NI* ni,int nr,int buffer_size, dword remote_header,int remote_header_size, int remote_nr,int remote_buffer_size)`

Furthermore, we add asynchronous communication to the synchronous CSP interface. In synchronous communication, the sender and receiver processes not only exchange a message, but they also synchronise during this communication: the send and receive operations only terminate after they both have been executed. In asynchronous communication, the send and receive operations do not necessarily synchronise: the send operation may terminate even before the receive operation has been executed, by exploiting the use of buffers. By using buffers on the receiving host, asynchronous communication may result in latency improvements, since the message data may already have arrived at the destination host even before it is requested by the receiving process. The API allows the user to specify the amount of buffer space on the receiving host for each virtual link.

To avoid memory-to-memory copying on the host CPU, we make the DSNIC collect and deposit the message data directly in user space. Direct user space access puts constraints on the kind of memory that can be used: the memory must be locked, word aligned, and accessible via Direct Memory Access (DMA). Therefore, the API is extended with special allocation and release functions for message memory.

5 Protocol design

The IEEE 1355 standard describes the communication protocol up to the packet level: a DS link packet consists of a destination address, payload, and a packet delimiter. At the application level, processes need to be able to exchange messages. The gap between these two interfaces must be covered by a communication protocol. We try to optimise the performance,

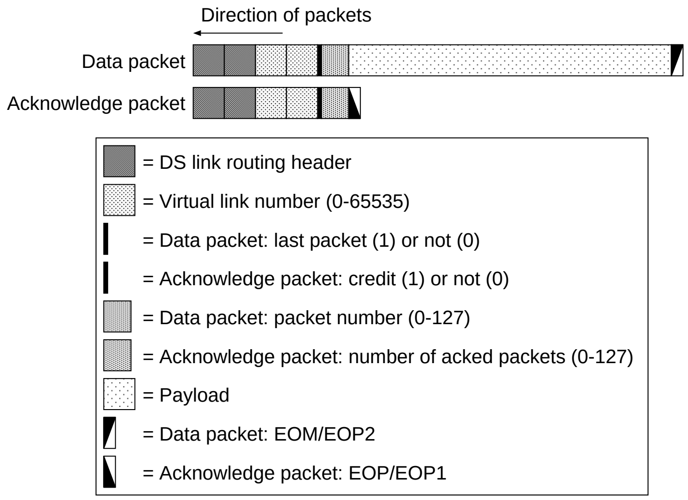

Figure 2: The format of data and acknowledge packets.

i.e., network latency and throughput, induced by the protocol.

5.1 Splitting messages into packets

The length of a message is not limited; it is determined by the application process. Messages must be communicated by exchanging packets. Using large packets in wormhole routed networks results in a performance penalty in both network latency and available network throughput due to network congestion. The impact of the packet size on the available network throughput has been shown in [6]: the maximum network throughput in a 512 end-node Clos network drops by 22 % if packet size 1024 is used instead of 16. Section 5.4 presents more detailed network performance measurements that are related to the characteristics of the DSNIC protocol. To avoid the performance penalty which occurs if each message is transmitted as a single packet, we have decided to split up each message into a number of limited-size packets. Each message is sent as a number of maximum size packets, and one last packet, which is smaller than or as large as the maximum size.

5.2 Optimising throughput

The protocol supports the adaptive routing capabilities of the C104 switch to improve the network throughput performance[6]. However, adaptive routing can cause packets of the same message to arrive out-of-order at their destination.

The communication protocol of the T9000 transputer[18] uses a strict acknowledgement scheme to avoid out-of-order packet arrival. This scheme limits the maximum message throughput[7] due to the latency of the acknowledge. Measurements have shown that two T9000 transputers, interconnected via one C104, can achieve a throughput of 4.7 Mbytes/s over a single virtual link. Interconnecting them via five C104s, which increases the latency of the acknowledge by 4 μs, reduces the maximum throughput to 2.3 Mbytes/s.

We use a sliding window protocol[19], together with an acknowledgement scheme without the limit in maximum throughput: each acknowledgement packet can acknowledge a sequence of packets. The protocol does not use piggybacking because we do not expect a performance gain from it, since DS link networks perform well for small packets, and the protocol requires only a few acknowledgements per message.

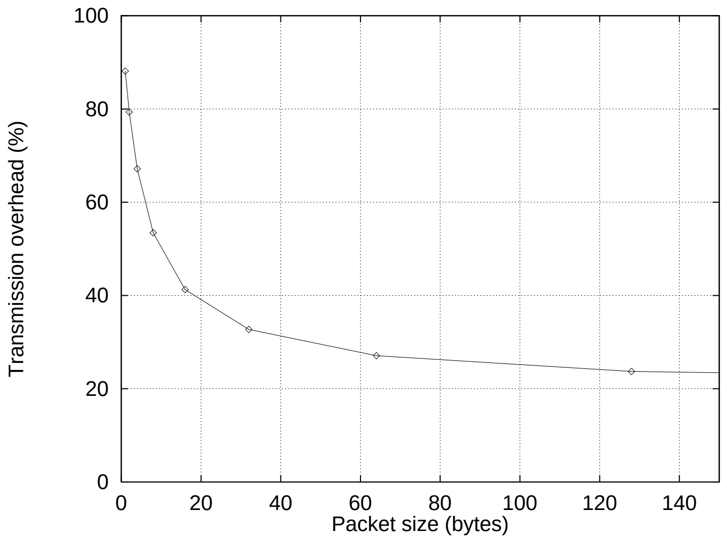

Figure 3: The transmission overhead against packet size for message length 1000.

5.3 Reliability and end-to-end flow control

The protocol should provide reliable communication. Since DS link networks are very reliable, we decide to accept the maximum BER of 9.6×10^{-18} for our purposes. The consequence is that the occurrence of an error in the network will result in unspecified system behaviour. We assume that the DS network does not to lose or corrupt any data, however, we should also avoid data being lost by the message receiver due to overflowing buffers. Therefore, a means of end-to-end flow control is required.

We choose *not* to implement end-to-end flow control via the DS per-link flow control, holding data reception as soon as the buffers are full, since this blocks the network until the receiver executes a receive operation, and therefore leads to poor network performance and possibly deadlock. We use credit based end-to-end flow control: the receiver allows a sender to start message transmission by sending it a credit.

5.4 Transmission overhead and optimal packet size

We define the transmission overhead to be the number of bits, communicated for a message, that do not represent the data bits of the message. In particular, packet headers and acknowledge packets determine the transmission overhead. Most forms of serial communication use encoding, e.g., adding stop bits, to ensure that the receiving side correctly interprets the serial bit stream. DS links use an encoding scheme that extends each byte, 8 bits, to a 10 bit token. The transmission overhead is therefore at least 20 %.

Figure 2 shows the format of data and acknowledgement packets that we use for the DSNIC protocol. It shows that, apart from the DS link routing header, three characters are used for protocol specific information, independent of the payload size. Together with the acknowledgement scheme, this allows us to calculate the transmission overhead for different packet sizes. Figure 3 shows the transmission overhead against packet size, or, more precisely, against payload size, for sending a 1000 byte message. The packet size strongly influences the required number of packets and packet headers, and thereby the transmission overhead.

Figure 4 shows the influence of the packet size on the maximum network throughput for a 512 end-node Clos network under random traffic. This graph shows an optimal network

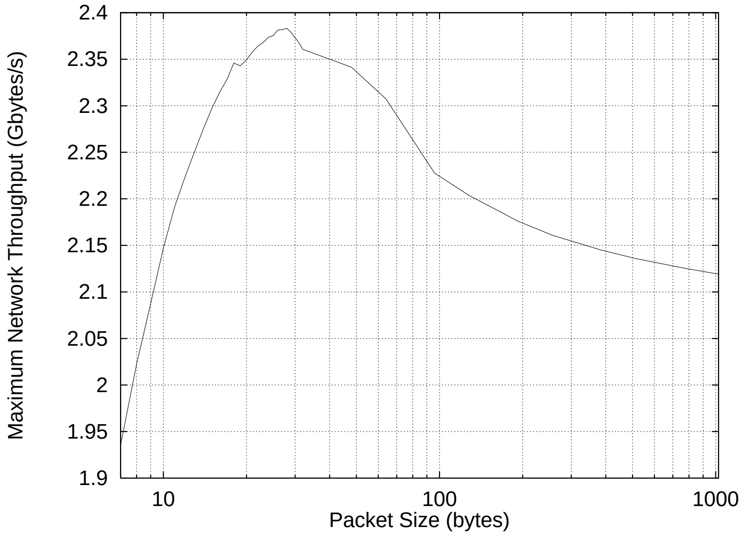

Figure 4: The maximum network throughput versus packet size.

throughput for packet size 28. For packets smaller than 28, the network throughput drops due to the domination of the transmission overhead. For packets larger than the optimum, the network throughput becomes worse due to network congestion.

The header of each packet needs to be processed by the DSNIC. This processing requires time. Using a small packet size, such as the optimal 28, requires a lot of processing to achieve full bandwidth communication. To keep this processing from becoming the system's bottleneck, we choose not to fix the packet size, but to make it adaptable so that its influence on the performance of the DSNIC can be investigated. We only support powers of two for the packet size to accommodate the implementation.

6 System design

The DSNIC should offer the described API to processes, and implement communication via the described protocol. This functionality must be distributed over the FPGA firmware and the host software.

Measurements have proven that the maximum throughput cannot be reached if the host CPU is used to transfer every single byte of a message via register I/O. The firmware therefore minimally needs to offer a packet transfer interface. Being unable to estimate the development time and resource utilisation, i.e., the required number of Logic Cells (LCs), for complex firmware functionality, made us decided to initially only implement the packet transfer functionality in firmware.

A well-known problem in interfacing is memory-to-memory copying. Memory-to-memory copying is an expensive and often nonessential operation which causes CPU loading and latency. Consequently, we choose to avoid it: message data must be DMA-transfered directly to the right address in a process' memory.

6.1 Firmware

A block diagram of the firmware components is shown in Figure 5. Two main data flows can be recognised: data reception and data transmission.

Comin takes care of packet reception, it splits packet headers from packet data. The headers are sent to the *headlenfifo*, and the data is sent to the *infifo*. Apart from splitting,

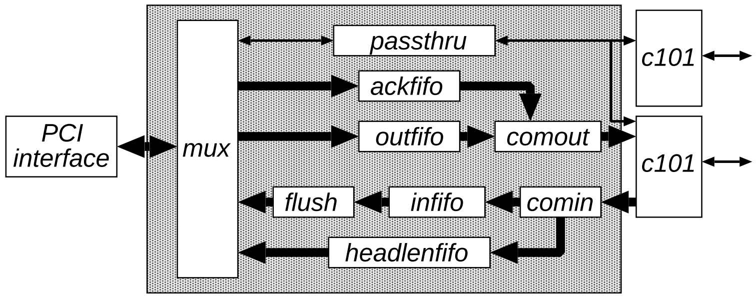

Figure 5: Block diagram of the firmware.

comin also counts the length of the packet, which is known as soon as the end-of-packet character has been received. This length is also sent to the *headlenfifo*. Data in the *headlenfifo* is delivered to an AMCC mailbox, which can generate an interrupt. On a header reception interrupt, the host CPU can determine destination address of the packet data and establish the receiving DMA, thereby avoiding memory-to-memory copying.

To avoid a store-and-forward mechanism, the receiving DMA must be established as soon as the header of the packet has been received. To set up a DMA, the length of the DMA transfer must be known beforehand. Therefore *flush* pads out each packet to the full packet size, even though less data is communicated over the DS link. DMAs can now be set up immediately on header reception, using the full packet size.

Comout takes care of sending data and acknowledge packets. If data is available in both the *ackfifo* and the *outfifo*, data in the *ackfifo* will be selected. This way, acknowledgements can bypass enqueued data packets.

In order to hide the reaction latency of the software driver, there are FIFOs on both the receiving and the transmitting side. The buffer size of 1 Kbyte was chosen because it fits well into the internal resources of the Altera FPGA. A 1 Kbyte FIFO can hide a reaction latency up to 100 μs, which should be sufficient.

Mux multiplexes the AMCC to all the FIFOs so that interleaved transmission and reception is possible. *Passthru* provides access to the registers of the C101s. The registers of both C101s are accessible via register I/O. The firmware only supports one C101 efficiently, i.e., with a DMA driven interface.

6.2 Host software

The host software consists of an interface library that provides the API, as shown in Table 1, to processes. This library communicates via system calls to a driver, resident in the kernel, that implements all the message communication functionality, using the packet transfer interface provided by the DSNIC board.

We choose to make the software interrupt driven and to deschedule processes performing blocking communication, in order to allow concurrent computation and communication.

The host software should take care of the communication of every packet, of every message, for any process, over any virtual link. These aspects make the host software complex. Using an Object-Oriented[15] approach helped in the development of this software.

By only using facilities that are common in every operating system that offers real-time facilities, such as process scheduling, memory locking, and interrupt handling, we facilitate portability of the DSNIC software to other operating systems with real-time features, assuming they allow access to a PCI bus. Furthermore, using only standard kernel facilities makes the DSNIC an extension of the OS: processes are *not* restricted in the use of other OS facilities, e.g., storage facilities or other communication facilities.

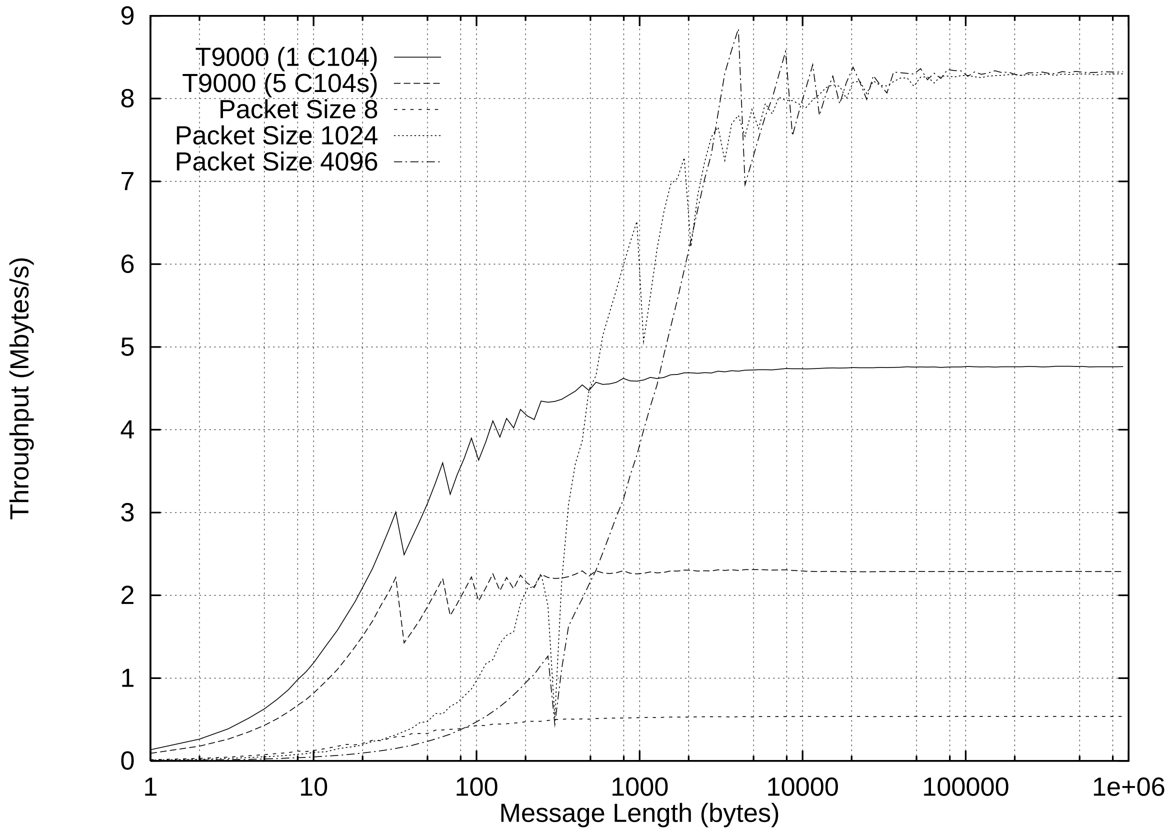

Figure 6: Throughput versus message length for Comms1 for various maximum size packets.

7 Measurements

We have used two benchmarks: Comms1 and Comms2. These benchmarks are based on Parkbench[9]. For both benchmarks, two Pentium Pro 200 MHz PCs, each running a communication process, are interconnected via the DSNIC with a single DS link. Not using a large DS network ensures that we are able to measure the process-to-process communication performance without network influence.

In both Comms1 and Comms2, the processes bounce messages of a certain length. For each message length, we obtain the process-to-process message latency by measuring the message bounce time and halving it. The throughput is obtained by dividing the message length by the message latency. In Comms1, the two processes, using blocking communication, bounce a single message. The link is therefore mainly used unidirectionally. In Comms2, the two processes exchange messages simultaneously, using both blocking and non-blocking communication. The link usage is therefore bidirectional. For an ideal bidirectional link, the maximum throughput in Comms2 is twice the maximum throughput in Comms1.

We have performed the Comms1 and Comms2 benchmarks for the following packet sizes: 8, 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096. To ensure readability, we only show the results for packet sizes 8, 1024, and 4096 in the graphs. As a reference, we also show the benchmark results of the T9000 transputer. These results have been obtained on two 20 MHz T9000 transputers, that are interconnected via a C104 switch network. The communication processes on the two T9000 transputers communicate via a single virtual link. The T9000 transputer uses a packet size of 32 bytes.

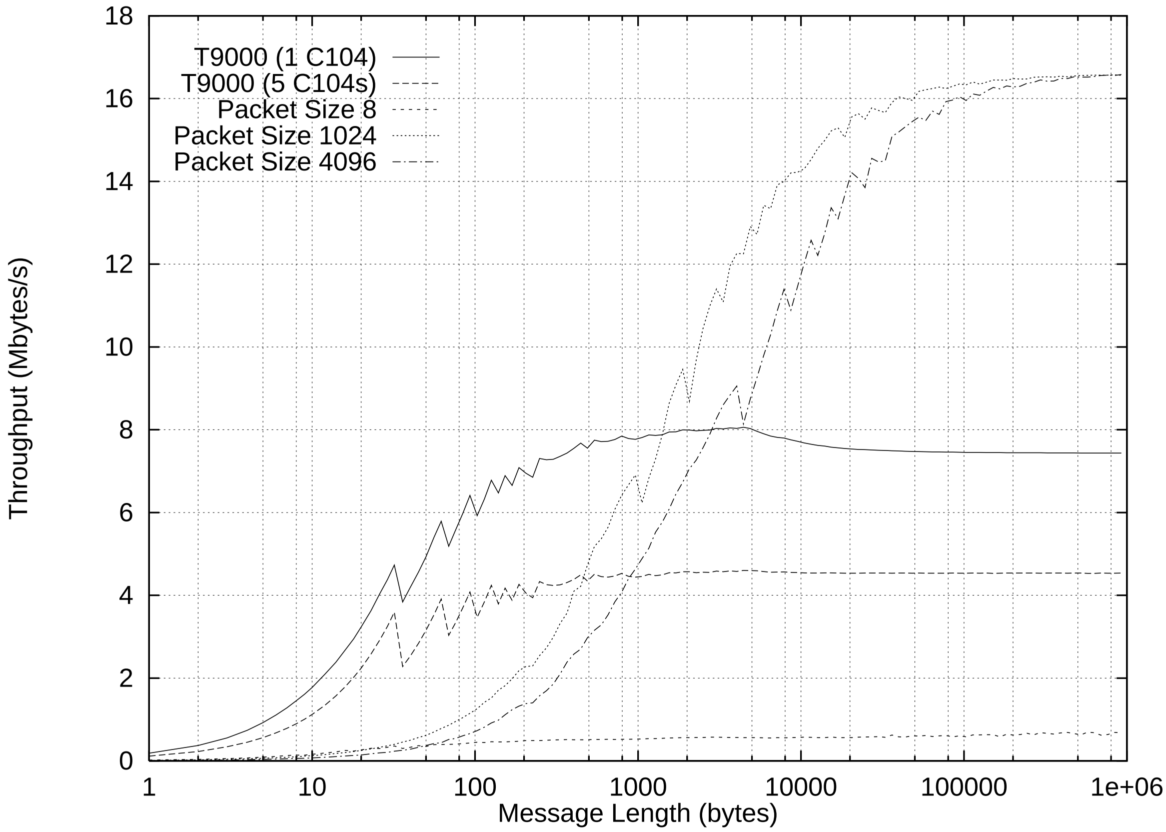

Figure 7: Throughput versus message length for Comms2 for various maximum size packets.

7.1 Throughput

Figure 6 shows the throughput versus message length for Comms1 for the three packet sizes and the T9000 transputer. The maximum throughput reached for long message lengths is 8.3 Mbytes/s. This is less than the maximum C101 throughput of 9.2 Mbytes/s. The reason for this is a limitation in the C101 receive FIFO. Only in some very specific cases, the limitation allows achieving a throughput higher than 8.3 Mbytes/s. The peaks in the 4096 packet size graph show such a situation.

The minimal packet size required to achieve the maximum throughput in Comms1 is 512. Using packet size 256 the 8.3 Mbytes/s is nearly reached. The other packet sizes are too small to be handled efficiently enough to reach the maximum throughput.

The plots in the graphs show a sawtooth pattern. This pattern is related to the packet size: if the message length exactly fits within a number of data packets, the packet handling overhead is relatively low, and therefore the throughput will show an optimum. If, however, the message length is one byte longer, an extra data packet is needed, so the packet handling overhead will be relatively high, which causes a throughput dip.

The maximum Comms1 throughput achieved on two T9000 transputer depends on the network latency, caused by the C104 switches that interconnect the two transputers. Directly interconnecting the T9000 transputers will result in an increase of maximum throughput of about 25 %. Due to the use of a different acknowledgement scheme, interconnecting two DSNICs via a number of C104s does not affect the maximum throughput, it only affects the latency.

Figure 7 shows the throughput versus message length for Comms2. As expected, the

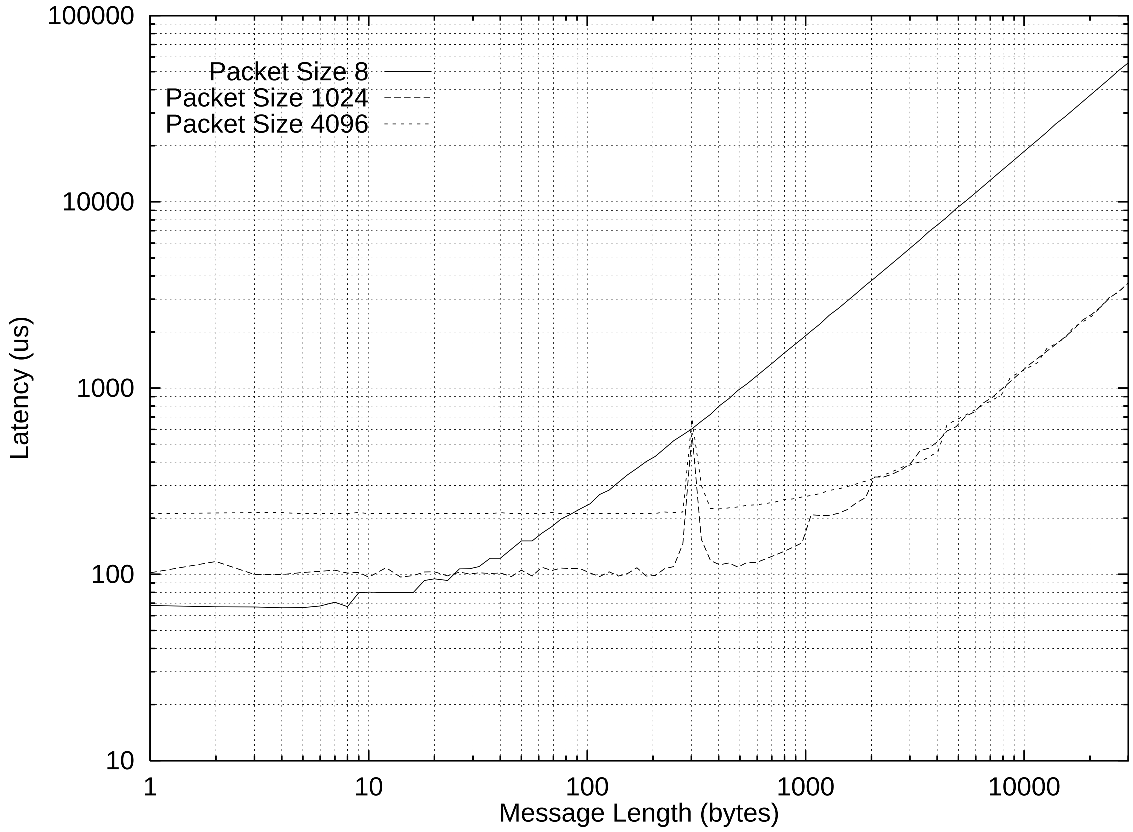

Figure 8: Latency versus message length for Comms1 for various maximum size packets.

maximum throughput reached for long messages is 16.6 Mbytes/s: twice the 8.3 Mbytes/s Comms1 throughput. This proves that the DSNIC can fully exploit the bidirectional bandwidth of DS links.

The minimal packet size required to achieve the maximum throughput in Comms2 is 512. Packet size 256 definitely does not achieve the maximum throughput. Considering the Comms1 situation, this is to be expected: since the throughput is twice as high, the packet size must roughly be doubled in order to keep the CPU load the same.

7.2 Latency

Figure 8 shows the Comms1 latency versus message length. For message length 1, the graph shows the minimal latency. The minimal latency for all packet sizes up to 512 bytes is 67 μs. The minimal latency for 1024, 2048 and 4096 sized packets is higher. Similar results have been obtained for Comms2.

The higher latency for large packets, see Figure 5, is caused by the padding of *flush*. This overhead can be removed by modifying the protocol, so that the packet's header contains the packet's length. This change will make the minimal latency 67 μs, independent of the packet size.

The T9000 transputer has a minimal Comms1 latency of 7.4 μs; two directly connected T9000 transputers will even improve on this. The factor of 10 difference shows the performance lost due to the use of standard OS facilities, implemented in software, instead of dedicated hardware, i.e., the hardware communication system and the hardware scheduler.

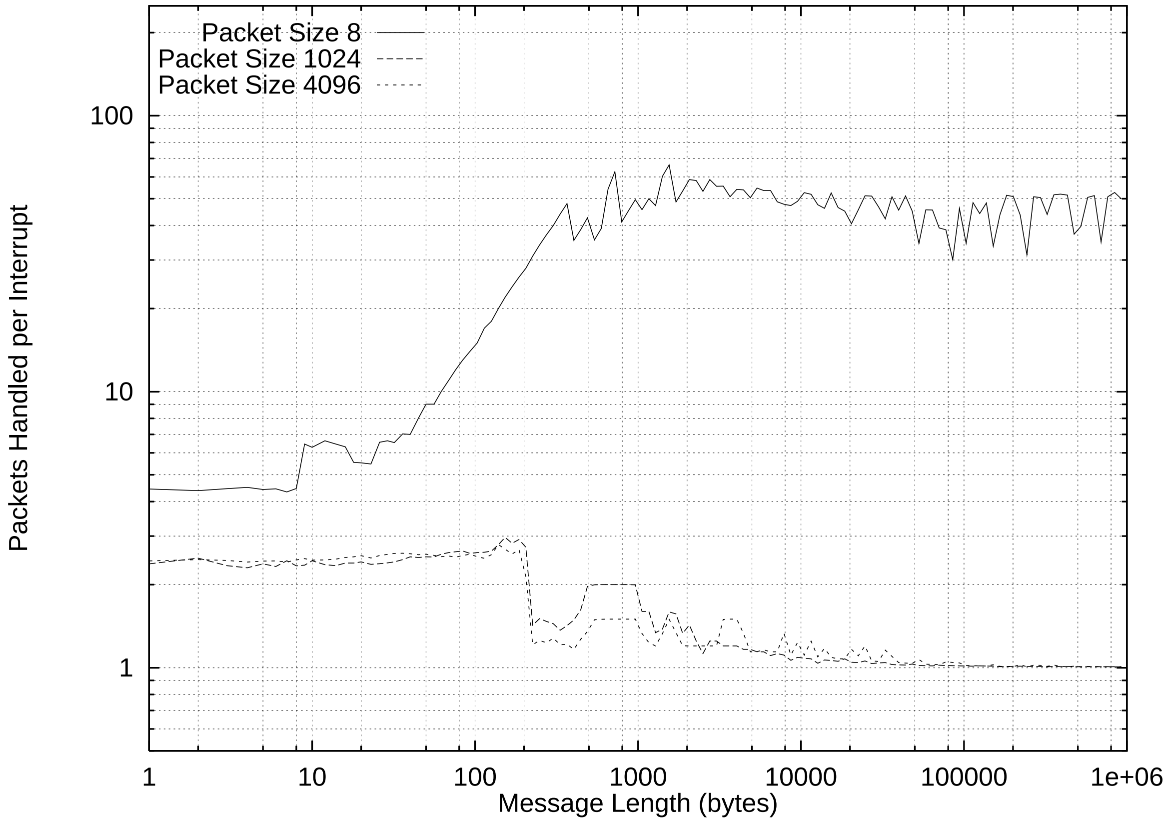

Figure 9: Number of packets handled per interrupt versus message length for Comms1 for various maximum size packets.

7.3 *Interrupt packet handling*

In the DSNIC, an interrupt handler takes care of all the packet handling to communicate messages. Figure 9 shows for Comms1 the number of packets handled per interrupt versus message length. The results in this graph are strongly related to the way in which the interrupt handler operates. The interrupt handler is activated by a number of hardware events, e.g., FIFOs, or mailboxes, becoming full or empty. The interrupt handler could terminate as soon as the events that interrupted the processor are handled. However, in order to improve performance, the interrupt handler, after having handled the events, checks whether any other hardware events need handling. If so, it handles them, and checks again for new events. Therefore, the interrupt handler can handle many packets each time it is called.

Handling many events per interrupt improves performance, since it avoids unneccesary processor interrupts. However, if many events are available, the interrupt handler is active for a long time. During this time no other interrupts can be handled, which has proven to cause problems to the timer interrupt. To avoid interrupt handling that takes too long, we limit the maximum number of events handled per interrupt. The effect of this is shown by the limit of about 70 handled packets per interrupt for the smaller packet sizes.

In Figure 9 two stable states can be identified, *many* and *one*: the state *many*, handling the maximum number of packets per interrupt, and the state *one*, handling one packet per interrupt. The small packet sizes result in such a large event handling frequency compared to the event handling time, that state *many* is reached. The large packet sizes result in such a low event handling frequency that state *one* is reached.

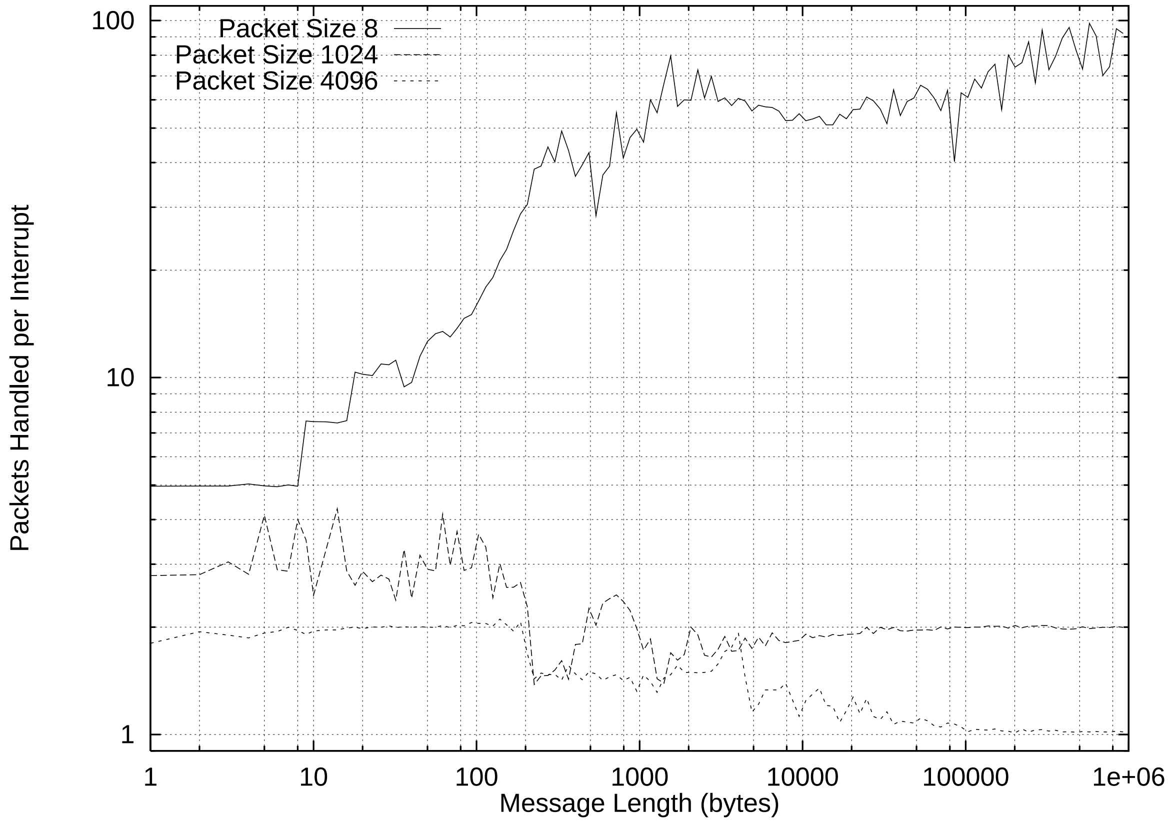

Figure 10: Number of packets handled per interrupt versus message length for Comms2.

The larger packet sizes show a drop off at message length 270 to about 1 packet handled per interrupt. So, if a message of more than 270 bytes is sent, the interrupt handler only handles one packet, because at the end of the handler no new events are available. Handling one packet on interrupt therefore takes as much time as it takes to communicate a 270 byte packet. This drop in efficiency of the interrupt handler also causes the latency peak and throughput dip at message length 270, see Figure 8 and Figure 6.

Figure 10 shows the number of packets handled per interrupt versus message length for Comms2. In Comms2, we are dealing with bidirectional communication. Therefore, three stable states can be identified, the two states from Comms1, i.e., states *many* and *one*, and one extra state, state *two*. In state *two*, two packets are handled per interrupt, an outgoing and an incoming packet. Medium sized packets, such as packet size 1024, end up in state *two*.

7.4 Processor load

The communication results can only be put into perspective if the communication load to the processor is known. Consequently, we have also performed load measurements. The load is measured by concurrent execution of a communication task and a computation task on each of the two processors. The computation task measures the remaining processor power. It does this by increasing a counter in memory. By measuring the counter increase per second on an unloaded system beforehand, we can determine the processor load in a system with communication.

In Linux, there are two facilities to execute a task: processes and threads. Each process has its own memory space, which keeps all other processes from accessing this memory.

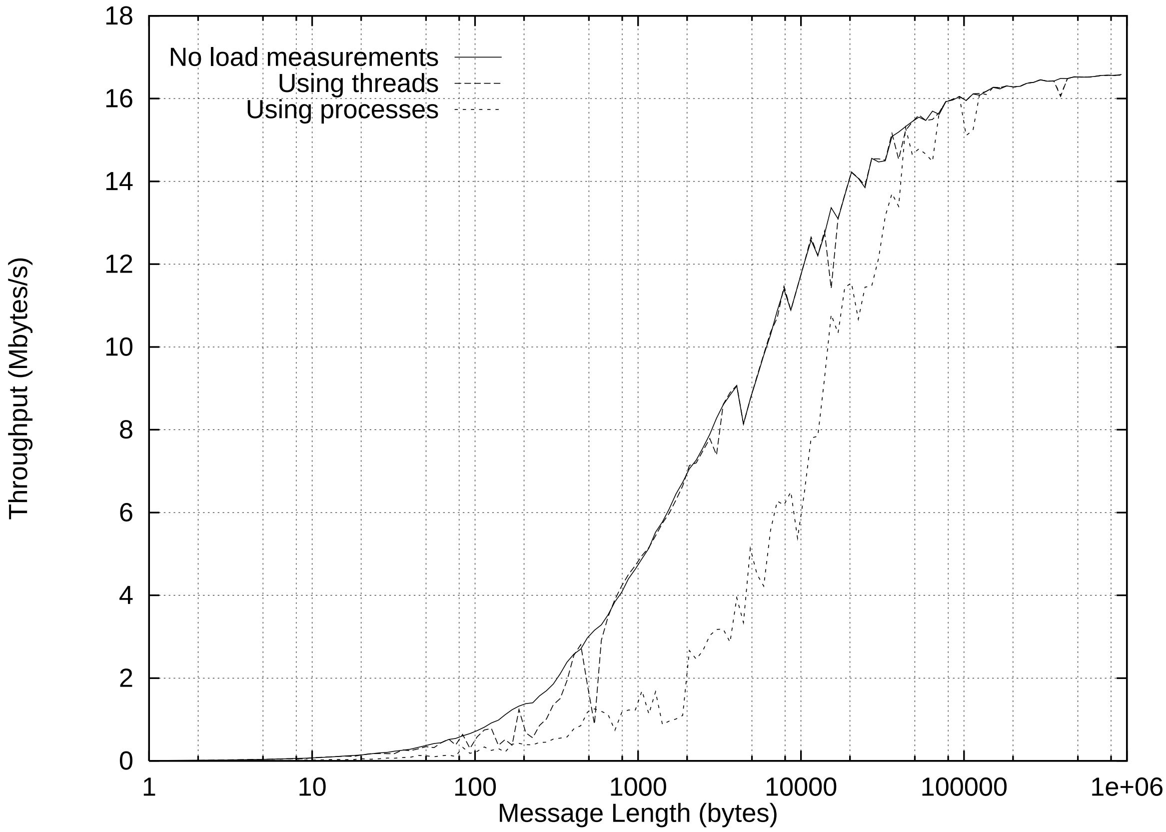

Figure 11: The effect of computation on the throughput for different message lengths, Comms2 and maximum packet size 4096.

Threads exist within processes, sharing the memory space of the process. Memory spaces are mapped onto physical memory by the Memory Management Unit (MMU) of the processor. Context switching requires changing the mapping of the MMU. Since threads of the same process use the same mapping, context switching between these threads takes less CPU time than context switching between processes.

The amount of time available for a process depends on the amount of time assigned to it by the scheduler. To avoid suffering of the communication process from the calculation process, a high priority real-time scheduling strategy is used for the communication process. This strategy should make sure that the communication process gets scheduled-in as soon as work for it is available, i.e., at the moment the blocking communication has finished.

Figure 11 shows three Comms2 throughput graphs: without computation, computation by a thread, and computation by a process. The graphs show that using concurrent threads has little impact on the communication, and that using processes has a significant impact. We can estimate this impact by using a quantification based on the Comms1 and 2 linear timing model[9]:

$$t_n = t_0 + \frac{1}{r_\infty} \times n$$

$$r_n = \frac{n}{t_n}$$

In this model, t_n is the latency for a message of size n; t_0 is the latency to send a zero byte message; r_n is the throughput; and r_∞ is the asymptotic bandwidth, i.e., $\lim_{n\to\infty} r_n = r_\infty$,

Table 2: Estimations of the 0-byte message latency t_0, for Comms2 and packet size 4096.

Situation	$n_{1/2}$**(bytes)**	$t_0(\mu$**s)**
No load	3000	187
Thread load	3000	187
Process load	13000	783
T9000 (1 C104)	26	3.25

Table 3: Maximum CPU load for long messages.

Packet size (bytes)	**Comms1 CPU load (%)**	**Comms2 CPU load (%)**
8	100.0	92.8
16	100.0	87.4
32	100.0	82.8
64	99.9	79.5
128	99.9	89.8
256	93.7	93.0
512	55.3	97.2
1024	28.9	53.4
2048	14.1	36.1
4096	7.1	19.0

the limit on the maximum throughput. From this model one can derive that:

$$t_0 = \frac{n_{1/2}}{r_\infty}$$

The value $n_{1/2}$ is the half-performance message length, i.e., the message length required to achieve half the asymptotic bandwidth. Knowing $r_\infty = 16.6$ Mbytes/s, we can estimate t_0, by obtaining $n_{1/2}$ from the throughput graphs in Figure 11. These estimations, see Table 2, show that the context switching overhead due to the use of processes instead of threads is significant: a difference of about 600 μs in t_0. The T9000 results, which are also included in this table, show how the T9000 outperforms the DSNIC with respect to 0-byte message latency.

The CPU load of both the process and the thread version are similar for long messages. This is to be expected: for large messages few process switches need to be performed because the communication task is nearly continuously sleeping, so the load is caused by interrupt handling which is identical in both situations. In that case, the interrupt frequency, which is directly related to the packet size, has a major effect on the load.

Table 3 shows the maximum CPU load measured during Comms1 and Comms2 for 0.5 Mb up to 1 Mb messages. For Comms1, we see that the CPU load roughly doubles if the packet size halves, so *packet size* $\times$ *load* $=$ *constant*. According to this formula, the *constant* is the packet size at which the CPU load would be 100 % to achieve full bandwidth utilisation. Using packet sizes smaller than the *constant* will not allow full bandwidth utilisation. For Comms1, the *constant* is approximately 290 for packet sizes of 512 and more. The 290 indicates that the maximum throughput cannot be reached with packet size 256, as we have seen before. Furthermore, the 290 corresponds to the end of the throughput dip in Comms1, where the expected throughput is reached again, see Figure 6.

8 Analysis

8.1 CPU time for handling a packet

During the communication of small messages, the CPU is almost completely dedicated to packet handling: it is nearly continuously executing the protocol for every packet that is communicated. The only interference is caused by exiting the interrupt handler for a short while when the maximum number of events that can be handled in a single interrupt has been reached. This allows us to calculate a maximum of CPU time required to handle a packet if it would be dedicated to this task. For Comms2 and a packet size of 8 bytes, the maximum throughput reached is 0.696 Mbytes/s. This means that the CPU, if dedicated to packet handling, can handle a packet within 11.5 μs. Since the same protocol implementation is used for any packet size, the required CPU time for handling a packet does not depend on the packet size.

8.2 Off-loading to a dedicated processor

A dedicated processor might be an interesting solution for off-loading more DSNIC functionality to the board. The CPU time for handling a packet allows us to relate the CPU power of such a dedicated processor to the packet size that can be handled by this processor, assuming the same communication protocol is used. This would result in a system in which the board takes care of all the packet handling, and the interface between the CPU and the board allows message communication.

Suppose the dedicated on-board processor is n times slower than the 200 MHz Pentium Pro. This means that it requires $n \times 11.5$ μs to handle a packet. To be able to keep up with the full bidirectional DS link bandwidth, 16.6 Mb/s in our case, we can calculate the required minimal packet size: 16.6 Mbytes/s $\times$ 11.5 μs $\times$ n.

The typical message length used for real-time analysis in ATLAS is 1 kbyte. For ATLAS, the dedicated processor is therefore required to be capable of handling packets of this size adequately. A communication processor that is 5.4 times slower than the Pentium should just be capable of link saturation. It is expensive to use a dedicated processor to achieve the optimal packet size of 28 bytes, since it requires a processor with about 7 times the power of the Pentium. If such a small packet size is required, one has to use dedicated hardware.

8.3 Interrupt service time

In Section 7.3, we derived that an interrupt takes as much time as it takes to send a 270 byte packet. Knowing that data is sent at 8.3 Mbytes/s, we can derive that it takes 32.5 μs to service an interrupt.

This interrupt service time can be checked against the CPU load for long messages. For long messages, the processor load depends only minimally on the task switching time. In this case, the CPU load is caused by interrupts and memory bandwidth usage.

Consider the Comms1 communication of long messages in 4096 byte packets. Knowing that an interrupt takes 32.5 μs to handle, we can conclude that on an average, communicating a single byte requires 32.5 / 4096 = 7.93 ns CPU time. It requires 65800 μs CPU time every second to sustain the full data rate of 8.3 Mbytes/s. The CPU load due to interrupts for long messages and packet size 4096 is therefore 6.6 %. This 6.6 % is close to the 7.1 % observed. The difference can be explained by the memory bandwidth usage, and cache disturbance due to the interrupts.

8.4 Interrupt overhead

We have derived that a packet can be handled in 11.5 μs, and that an interrupt is serviced in 32.5 μs. Therefore, we can calculate the interrupt overhead for an interrupt that services one packet: 32.5 - 11.5 = 21 μs. The interrupt overhead is caused by context switching. On interrupt handler entry, the context of the current process and its thread must be saved. On exit, it must be restored. Another negative effect of context switching is cache usage. On handler entry, memory locations different from the memory locations in the cache are used, and therefore cache updates are required. In a similar way, the interrupted process suffers from the interrupt: the cache it was using is disturbed, and cache updates are required.

9 Performance improvements

Improving performance means obtaining higher throughput, lower latency, and lower CPU load during communication. The communication performance depends on both network and interface performance. Network performance is influenced by the communication protocol, as discussed in Section 5. The interface performance is determined by the efficiency of the implementation of the required operations.

We can distinguish three classes of operations: (1) *O*(*message length*) operations, such as copying of message data and CRC checking, (2) *O*(*nr of packets*) operations, the handling of packets, and (3) *O*(*nr of messages*), communication initialisation and task switching.

By using DMA directly to process space and a reliable network, we have avoided implementing any *O*(*message length*) operations on the CPU. The maximum throughput is therefore limited by the C101 hardware and not by the CPU power. We have chosen to implement the *O*(*nr of packets*) operations, i.e., the packet handling, on an interrupt basis. The *O*(*nr of messages*) operations are implemented as kernel calls that execute reschedule operations.

We focus on two methods to improve performance: using the DSNIC board for packet handling, and using alternatives for OS functionality. By extending the Comms1 and 2 communication model so it takes the CPU load caused by communication into account, we can estimate the effect of the two performance improvements on the throughput and the CPU load graph, see Figure 12.

9.1 Modelling the CPU load

We model the CPU time to communicate a message of length n by the sum of a fixed time c_0 and a time dependent on the message length n:

$$c_n = c_0 + c_{\mathit{byte}} \times n$$

The factor c_{byte} is the per-byte CPU time. Notice that $0 \leq c_0 \leq t_0$. Knowing the CPU time to communicate a message, c_n, and the communication latency, t_n, the following equation expresses the CPU load l_n:

$$l_n = \frac{c_n}{t_n} \times 100\ \%$$

The value l_0, which equals $(c_0/t_0) \times 100\ \%$, expresses the CPU load for sending a 0-byte message. Furthermore, we can derive the following for the asymptotic CPU load l_∞:

$$l_\infty = \lim_{n\to\infty} l_n = c_{\mathit{byte}} \times r_\infty \times 100\ \%$$

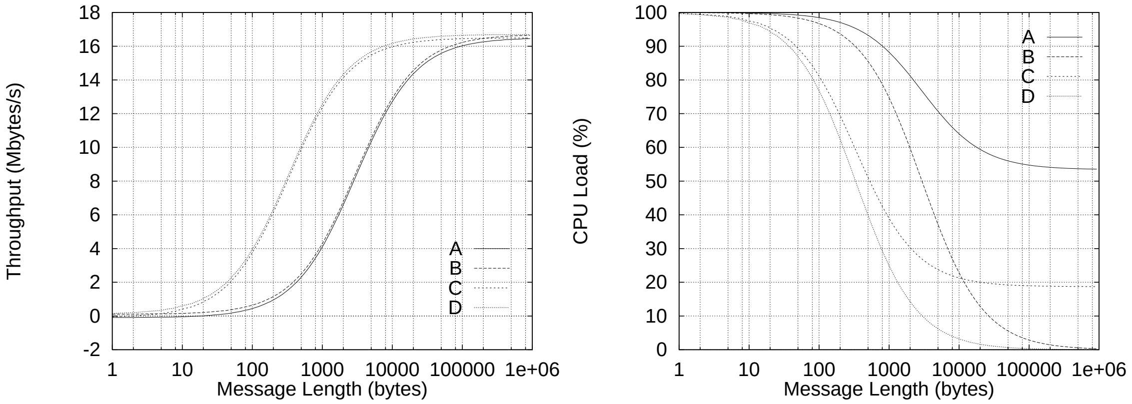

Figure 12: The effects of two performance improvement methods on the throughput and CPU load graph for Comms2 and packet size 1024. Graph A models the performance of the current system when using threads to perform the computation. Graph B models the performance of a system in which all packet handling is off-loaded to the DSNIC board. Graph C models the performance of a system with low-latency context switches. Graph D models a system that takes both the advantages of B and C. Compared to A and B, graphs C and D show an improvement in throughput.

In Section 8.3, we have already used this equation. The asymptotic CPU load, l_∞, calculated there, using $c_{\,byte} = 32.5\ \mu\mathrm{s}/4096$ bytes $= 7.93$ ns/byte for the per-byte CPU time for 4096 byte packets, matches the measured CPU load for long messages.

The results presented in this paper allow us to derive the parameters of the model that match the Comms2 communication using a packet size of 1024, i.e., the typical packet size required for ATLAS, and threads to perform the computation results. From Figure 7, we can determine that r_∞, the asymptotic bandwidth, equals 16.6 Mbytes/s. Furthermore, this figure allows us to determine that $n_{1/2} = 1480$. Since $t_0 = n_{1/2}/r_\infty$, we know that $t_0 = 178\ \mu$s. Knowing from Table 3 that $l_\infty = 53.4$ %, allows us to determine that $c_{\,byte} = 32.17$ ns/byte, using $c_{\,byte} = l_\infty/(100\ \% \times r_\infty)$. We cannot derive c_0 from the measurements, therefore we assume the CPU load is 100 % during 0-byte message communication, so $c_0 = t_0$. These parameters have been used to reconstruct the performance graphs of the current system, see graph A in Figure 12.

9.2 Using the DSNIC board for packet handling

By off-loading all the packet handling to the DSNIC board, as discussed in Section 8.2, all the packet handling interrupts, which are *O*(*nr of packets*), can be avoided. Avoiding these interrupts has no effect on the communication latency, since this latency is caused by the *O*(*nr of messages*) operations. Since latency and throughput are directly coupled in Comms1, there is no effect on the throughput graph. The gain is to be expected in CPU load. For ATLAS, which requires packet size 1 Kbyte, this can result in a performance gain of up to 53 %, see Table 3. This potential gain is modelled using $c_{\,byte} = 0$, see graph B in Figure 12.

9.3 Low-latency context switches

Context switches are the main cause of both latency and CPU load. We can distinguish three types of context switches: (1) rescheduling, requiring context switches between processes or between threads, (2) kernel calls, requiring context switching between process and kernel space, and (3) interrupts requiring context switching between process and kernel space.

In a general purpose OS, these context switches require a lot of CPU time, and thereby cause both latency and load. Using light weight alternatives for the OS context switches can therefore result in a significant performance improvement. However, a light weight alternative usually implies a restriction in some other field, e.g., no protection between processes, a limitation in the use of OS functionality, or support for only a single process. If the restriction is acceptable, this method offers an interesting solution. Solutions that apply this method can be found in [4], [14], and [20]. Graph C in Figure 12 shows the potential of such an improvement, assuming the 0-byte latency t_0 is 20 μs, instead of 187 μs, and assuming the interrupt handling overhead can be completely removed, which reduces the interrupt handling time from 32.5 μs to 11.5 μs. Furthermore, graph D in Figure 12 shows the effect of applying both the improvement methods: off-loading and low-latency context switches.

10 Conclusion

We have developed the DSNIC by continuously keeping the design aim in mind: optimise for low-latency high-throughput communication, requiring little CPU load. To accomplish this, the CSP based API is extended with facilities for asynchronous, non-blocking, and zero-copy communication. Furthermore, the communication protocol splits messages into limited size packets. This avoids continuous wormhole blocking and reduces the network latency. The protocol supports adaptive routing, and allows full bandwidth utilisation by a sliding window protocol and a non-restricting acknowledgement scheme. Reliability is ensured by the very low BER of DS link networks and the use of end-to-end flow control.

This way, we have managed to obtain a system that reaches 8.3 Mbytes/s unidirectional process-to-process throughput over a single virtual link. A maximum throughput of 16.6 Mbytes/s is reached for bidirectional communication. In both unidirectional and bidirectional communication, up to 90 % of the maximum theoretical bandwidth can be exploited. The DSNIC offers a 1-byte message latency of 67 μs. This is fast, considering that this is the process-to-process latency. During 8.3 Mbytes/s unidirectional communication, the DSNIC requires 7 % CPU load for a packet size of 4096 bytes. For that same packet size and 16.6 Mbytes/s bidirectional communication the CPU is loaded for 19 %.

We have observed that high throughput and low CPU load can only be obtained by using large packets, e.g., 4096 bytes long. On the other hand, we have shown that the ideal packet size for the 512 Clos DS link network is small: 28 bytes. Filling this gap completely would require dedicated hardware.

We have shown that protocol off-loading has the potential to result in a significant gain of available CPU power, e.g., up to 53 % for packet size 1024. Furthermore, we have shown that this method requires a substantial amount of on-board computing power.

In general purpose operating systems, context switches are a source of overhead. We have shown the overhead of one form of context switch: the interrupt. An interrupt that handles the communication of a single packet takes 32.5 μs, whereas the same operation on the same CPU can also be performed in 11.5 μs, an overhead of 65 %. Furthermore, we estimated the performance loss caused by context switches when using processes instead of threads to implement concurrent communication and computation.

Major performance improvement can be obtained by avoiding the use of OS facilities that require context switches, i.e., kernel calls, task switches, and interrupts; and obtaining adequate alternatives for them.

Acknowledgements

We are very grateful for the support of the European Union through the Esprit ARCHES project.

References

[1] *The ESPRIT ARCHES Project, The Application, Refinement and Consolidation of HIC Exploiting Standards*. ESPRIT P 20693

[2] *The ATLAS Technical Proposal*. CERN/LHCC/94-43, LHCC/P2, December 1994. ISBN 92-9083-067-0

[3] *DSNIC : The DS Network Interface Card*. `http://www.cern.ch/HSI/dshs/dsnic/dsnic.html`.

[4] Eicken, T. von. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. University of California at Berkeley, PhD, 1993.

[5] Haas, S., D.A. Thornley, M. Zhu, R.W. Dobinson, R. Heeley, N.A.H. Madsen, B. Martin. Results from the Macramé 1024 Node Switching Network. *Computer Physics Communications*, April 1997. CHEP '97, Berlin, Germany.

[6] Haas, S., D.A. Thornley, M. Zhu, R.W. Dobinson, R. Heeley, B. Martin. *Results from the Macramé 1024 Node IEEE 1355 Switching Network*. European Multimedia, Microprocessor and Electronics Conference, November 1997. EMMSEC97, Florence, Italy.

[7] Heeley, R. *Real Time HEP Applications using T9000 Transputers, Links and Switches*. University of Liverpool, PhD, October 1996.

[8] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 (0-13-153289-8 PBK).

[9] Hockney, R., M. Berry. *Public International Benchmarks for Parallel Computers*. PARKBENCH Committee: Report-1, February 1994. `http://www.netlib.org/parkbench/`.

[10] *IEEE Std 1355-1995, IEEE Standard for Heterogeneous InterConnect (HIC) (Low-Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction)*. IEEE Computer Society, 1995.

[11] *INFN, Instituto Nazionale Fisica Nucleare*. Sezione di Roma and University of Roma, La Sapienza, Rome, Italy. `http://www.roma1.infn.it/`.

[12] *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, Version 1.1, June 1995.

[13] *The Study of Noise on DS links*. OMI/Macramé, Esprit project 8603, Working paper 43, October 1996.

[14] Pakin, S., V. Karamcheti, A.A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MMPs. *IEEE Concurrency 1063-6552/97*, 5(2):60–73, 1997.

[15] Rumbaugh, J., et. al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[16] SGS THOMSON Microelectronics. *STC101 Parallel DS-Link Adaptor*. Engineering Data, August 1995.

[17] SGS THOMSON Microelectronics. *STC104 Asynchronous Packet Switch*. Engineering Data, April 1995.

[18] *The Transputer Data Book*, 2nd edition edition, 1989.

[19] Tanenbaum, A.S. *Computer Networks*. Prentice-Hall, Inc., third and international edition edition, 1996. ISBN 0-13-394248-1.

[20] Welch, P.H., M.D. Poole. *High bandwidth occam Demonstrator for Multiprocessor Alpha/DS-link systems*. Computing Laboratory, University of Kent at Canterbury, CT2 7NF, January 1998.

[21] Zhu, M., D.A. Thornley, J. Pech, B. Martin, N.H. Madsen, R. Heeley, S. Haas, R.W. Dobinson, C.R. Anderson. *Realisation and Performance of IEEE 1355 DS and HS Link based, High Speed, Low Latency Packet Switching Networks*. RT97, September 1997. Beaune, France.