Protocol Implementation Using Integrated Layer Processing

Torsten Braun[1] and Christophe Diot
2004 rte des Lucioles, F-06902 Sophia-Antipolis Cedex, France
e-mail: [Torsten.Braun|Christophe.Diot]@sophia.inria.fr

Abstract

Integrated Layer Processing (ILP) is an implementation concept which "permit[s] the implementor the option of performing all the [data] manipulation steps in one or two integrated processing loops" [1]. To estimate the achievable benefits of ILP, a file transfer application with an encryption function on top of a user-level TCP has been implemented and the performance of the application in terms of throughput and packet processing times has been measured. The results show that it is possible to obtain performance benefits by integrating marshalling, encryption and TCP checksum calculation. They also show that the benefits are smaller than in simple experiments, where ILP effects have not been evaluated in a complete protocol environment. Simulations of memory access and cache hit rate show that the main benefit of ILP is reduced memory accesses rather than an improved cache hit rate. The results further show that data manipulation characteristics may significantly influence the cache behavior and the achievable performance gain of ILP.

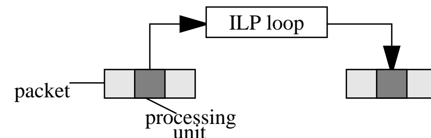
1 Introduction

During the last few years the performance of processors has been increasing faster than the performance of memory. The memory bottleneck can be reduced by avoiding access to memory as much as possible, e.g. by eliminating copy operations from protocol implementations. If avoiding memory access is not further possible another solution is to use caches, which are extremely fast but expensive. Usually, fast caches on the processor chips have limited sizes in the range of a few kbytes (e.g., 16 kbyte data cache and 20 kbyte instruction cache on a SUN SuperSPARC processor, 8 kbyte data and instruction cache on a DEC Alpha 21064). To get benefits from a cache memory it is necessary to keep as much useful data as possible within the cache memory.

The concept of Integrated Layer Processing (ILP) tries to gain from both sides, namely avoiding memory access and using cache memories. ILP allows the implementor to perform all manipulation steps in one or two integrated processing loops [1]. This integrated processing of data is commonly called "ILP loop". Theoretically, an ILP protocol stack implementation reads once from main memory, keeps the read data within registers or cache memory, and performs all the data manipulations for several protocol layers within the ILP loop (Figure 1). Processed data are finally written to the destination memory. In this ideal case, ILP requires only one read and one write access to the main memory for each processing unit, which is usually a 32-bit or a 64-bit word. All the other operations should work on registers, and possibly on the cache memory.

The non-ILP implementation reads a complete packet or data unit and performs one function after the other (Figure 1). The intermediate packets are always written to the main or to the cache memory. The number of memory accesses are significantly higher than in the ILP case. However, it is shown in the paper that also without ILP a good cache behaviour can be achieved and the cache hit rate of ILP may be even lower than in the non-ILP case.

ILP



non-ILP

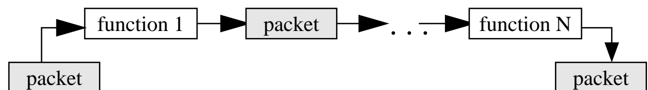


Figure 1. Concepts of ILP and non-ILP

By applying ILP to simple data manipulations like data copying and TCP checksum calculations, Clark and Tennenhouse achieved performance gains of up to 50% [1]. We carried out a similar experiment which yields nearly the same results. The XDR marshalling routine obtained from a stub compiler for an array of 20 integer values has been combined with the TCP checksum routine. The throughput is 70 Mbps for executing the two routines sequentially in contrast to 100 Mbps for integrating both functions into a single loop. The performance comparison shows over 40% gain in favor of the ILP implementation.

The achievable performance improvements gained by ILP depend on various parameters. The ILP benefits may increase with the number of integrated data manipulations as shown by Abbott and Peterson [2]. Partridge and Pink showed that performance gains by combining checksum calculation and copy operations may differ on different machines [3]. Experiments by Gunningberg, Partridge, and others proved that ILP is also sensitive to the complexity of data manipulation functions. They show that the integration of the

[1] on leave from the University of Karlsruhe, sponsored as a research fellow by the Commission of the European Communities under the Human Capital and Mobility Programme HCM (no. ERBCHBGCT930254)

DES (data encryption standard) algorithm into the ILP loop can reduce the performance gains significantly [4].

In this paper we demonstrate that benefits obtained from ILP not only depend on the data manipulation function complexity, but also on other characteristics of these functions such as the number and the size of required memory tables, and the necessary interaction between data manipulations and control functions. Therefore, experiments involving only data manipulations cannot show the real advantages of an ILP implementation. To get realistic performance results visible to applications, it is essential to perform ILP experiments with a real application on top of a protocol suite embedded into a complete system environment. The paper also describes ILP implementation concepts and experiments performed using a bulk data transfer application over a layered protocol suite.

Section 2 analyzes the general difficulties in integrating several data manipulation functions of different protocol layers. The solutions proposed in this section are applied to the implementation of a file transfer application with (un)marshalling and de/encryption on top of a user-level TCP implementation. Section 3 presents the architecture for non-ILP an for ILP implementations. Section 4 shows performance results and comparisons with conventional implementations.

2 Limitations of ILP and Possible Solutions

2.1 Related work

Operations such as multiplexing, concatenation, and segmentation that should be avoided in general according to [5] and [6] have also to be avoided within the ILP loop to ensure that single data units do not interfere with other data units and to preserve their frame boundaries according to the ALF (Application Level Framing) concept [1]. Further problems with ILP are identified by Clark and Tennenhouse [1] as well as by Abbott and Peterson [2]:

- Ordering constraints between different protocol functions occur because protocol processing consists of interactions between control functions (such as header and connection processing) and data manipulation functions [1]. Control functions often depend on the result of data manipulations. For example, a TCP header can only be completed after calculating the checksum over the TCP data. On the receiving side, data can be passed to higher functions only after a successful checksum evaluation. Furthermore, the results of the data manipulations, which will be handed out to control functions may even differ. For example, if the data passes function N without error, but a failure is detected in function N+1, the failure result has to be indicated to functions N+1 and above, while function N must assume that the data passed their tests correctly.
  A so-called three stage approach is proposed in [2] to manage ordering constraints dividing protocol processing into initial control operations, integrated data manipulations (or ILP loop), and a final protocol stage. The initial operations can usually be very small, and possibly reduced to demultiplexing and packet parsing operations. Messages are accepted or rejected in the final stage.

- Integrating code from different protocol layers may violate the modularity of the implementation. Preserving modularity in ILP implementations may be achieved by an automatic synthesis tool, e.g., based on a macro pre-processor [2], to generate the ILP protocol code. Another approach studied by us is the ILP extension of a stub compiler.

- The size of the processing unit to be manipulated may differ in various layers; e.g. an XDR marshalling procedure usually operates in 4-byte units, while encryption functions often manipulate the data in 8-byte units. Word filters are used in [2] to solve the mismatch which occurs when data passes from one data manipulation function to another one. A word filter operates on words (commonly 4 bytes). It outputs a word each time a word is input and indicates, in case of larger data units, the position of the output word in this data unit using a flag.

- In a layered architecture, the content and the structure of messages differ in each layer. The header of layer N becomes the data of layer N-1. Moreover, header content may depend on the data part (e.g., TCP checksum calculation or size changing data manipulations with length indicators in the header). The dependencies of data and headers make it impossible to process first the header part of layer N by a layer N-1 data manipulation function, and then the data part of layer N by the N data manipulation function. The solution proposed in [2] to avoid this problem are so-called segregated messages, which create a clear separation between protocol headers and application data. ILP is only applied to user data. Headers added at various layers in the protocol stack, which themselves become data, are processed separately, in a non ILP way.

2.2 Remaining Problems and Solutions

The solutions proposed in [2] to overcome ordering constraints and to preserve modularity are very general solutions, and we are using similar approaches in our ILP implementation. This paper will mainly focus on the two last problems mentioned in Section 2.1, namely the strategy to pass data across functions with different data unit sizes and dependencies between header and data.

Passing data across functions

Word filters output data as soon as it is ready, but they do not take into account whether it is more efficient for the next function to process the data in larger size units, e.g. if an intermediate result has to be written into the memory for each processing unit. Consider a protocol stack, in which encryption is applied to 8-byte units, and checksum calculation to 2-byte units, but where the word filter mechanism always hands out data in 4-byte units from encryption to checksum. This handout requires 2 write operations to calculate the checksum of a 8-byte word. A more efficient solution is to pass data in 8 byte processing units from encryption to checksum to reduce the number of write operations for checksum calculation to 1.

We propose to adjust the length of the exchanged processing units (L_e) so that

$$L_e = LCM(L_x, L_y)$$

where LCM(a,b) is the lowest common multiple of a and b, L_x is the processing unit length of function f_x, and L_y is the processing unit length of function f_y.

L_e should also be chosen large enough to utilize the hardware architecture efficiently. For example, in the case of a L_s byte wide memory bus, it could be more efficient to set

$$L_e = LCM(L_x, L_y, L_s.)$$

where L_s is the length dependent on system parameters.

Moreover, writing a packet of n bytes 1-byte-wise into a memory area which is not cached before each write operation could result in n cache misses, while writing it m-byte-wise could only cause n/m cache misses if the caching blocks are m bytes or larger.

Header and data dependencies

The solution proposed in [1] for the problem of header and data dependencies is to use segregated messages. This concept cannot be used in many situations, e.g. for encryption if a higher layer protocol header to encrypt is not aligned to the processing unit size of the encryption function. For marshalling, the complete message format including header and data is usually defined and marshalled by a given procedure. It is then necessary to process the complete message including data and higher layer headers such as RPC headers. We solve this problem by ***dividing a packet into several parts and by processing the part containing header fields dependent on data after processing the data part.***

This concept, which is a generalization of the approach of segregated messages, is explained in more detail in Section 3.2.2 and works only if the data manipulation functions are not ordering constrained. An ordering constrained function requires that data are processed in a serial order to ensure a correct result [7]. Examples of ordering constrained functions are the cyclic redundancy code (CRC) calculation for error detection and stream cipher encryption algorithms. The TCP checksum and block cipher encryption algorithms are examples of non-ordering constrained functions.

Even with non-ordering constrained functions, ILP requires that the number of bytes which have to be processed before a given block is known, because both operations need to be aligned on certain boundaries (e.g., 2 bytes for TCP checksum and 8 bytes for block cipher encryption). A major consequence is that ***ILP can be applied only if the header size is known before entering the ILP loop*** (i.e., the header size must be either constant or calculable from available protocol state information).

3 User-Level ILP Implementation

3.1 Implemented Communication System

The communication system used for the ILP experiments consists of two data manipulation functions, namely (un)marshalling and en/decryption functions, on top of TCP (including checksum) running in user space. A file transfer application is located on top of the protocol suite. The application has been developed in an UNIX environment (SUN SPARCstation with SunOS 4.1.3) using the C language. For additional experiments the program has been transferred to a DEC AXP workstation with OSF/1.

The file transfer application is based on the RPC model. A client sends a request describing the file to receive, the number of copies of this file to be received, and the maximum length of bytes to receive within a single reply message. After receiving a file transmission request, the server segments the file into smaller units and sends these units as a set of reply messages back to the client. The request and reply message formats have been described using ASN.1. For sending a request or a reply message, the application has to invoke the appropriate marshalling routine, which has been generated using the MAVROS ASN.1 stub compiler [8]. The marshalling routine generates the RPC header and the XDR format of the message.

The encryption function encrypts the output of the marshalling routine. Because most common encryption functions work on 8 byte boundaries, the complete message must be extended to a multiple of 8 bytes by adding alignment bytes at the end. The length of the message before encryption is written into a length field, which builds the encryption header. A fast encryption algorithm based on the SAFERK64 algorithm [9] has been chosen because the processing time spent in the more complex DES encryption algorithm can hide totally the ILP performance gain [4]. Even a high-speed implementation achieves only a 1 Mbps throughput on a SPARCstation [10]. SAFERK64 that is extremely fast compared to other standard algorithms such as DES (25 Mbps for SAFERK64 with 1 round compared to 0.5 Mbps for the system implementation of DES on a SPARCstation 10 with a 30 MHz clock) is still too time consuming for the ILP experiment. The encryption algorithm has been simplified to achieve a throughput of 50 Mbps on a SPARCstation 10 (100 times faster than DES). Of course, these simplifications lead to a less secure encryption. To keep the characteristics of the algorithm unchanged, at least one operation of each type occurring in the original algorithm is present in the simplified version. The simplified algorithm consists of a set of add/xor operations on each byte, followed by mixed logarithm/exponential operations on each byte, and final operations called `2-PHT(`a_1`,`a_2`)` `=(2*`a_1`+`a_2`,`a_1`+`a_2`)` on each pair of bytes (PHT: Pseudo Hadamard Transform). The add/xor operations require reading the key, while the logarithm/exponential operations access tables of precalculated values to avoid too costly run time calculations.

The encrypted message is delivered to TCP, which calculates the checksum over the pseudo header and the data. The user-level TCP implementation [11] is divided into two parts: a kernel part and a library to be linked to an application running in user space. The kernel part provides a datagram oriented socket interface and is activated for all connections, while there exists a separate user-level part for each application. The kernel part has similar functionality as UDP without checksum. For sending data, the main task of the kernel part is to pass the messages received from the user-level TCP to IP. On the receiving side, the kernel part demultiplexes IP packets to the corresponding user-level TCP connection, i.e. to the corresponding application. The request and reply messages received from TCP are decrypted and unmarshalled before the data is delivered to the application. Each TCP user-level connection receives only the packets of its associated application. TCP header options are avoided to ensure fixed-size headers. Furthermore, a single connection only supports uni-directional data transfer. According to the ALF concept, TSDUs are mapped to exactly one TPDU.

Figure 2 shows the packet format used by the three data manipulation functions. Dependencies between the data part and protocol headers are shown using dashed arrows. At the TCP level the

checksum field in the header depends on the contents of the data. The encryption fields depend on the length of data delivered by the marshalling function. Alignment bytes are added at the end, while for reasons explained in Section 3.2.2, the length field is in the header, although a length field at the end of the encrypted message as done in other security protocols would simplify an ILP implementation. Generally, trailer fields for protocol information dependent on user data could simplify ILP processing, although trailers make parsing of protocol information more complex.

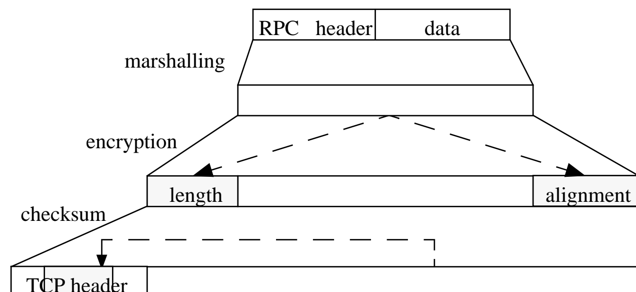


Figure 2. Data formats of the implemented protocol suite

The protocol suite has been tailored to specifically address the problem which has been identified in Section 2.1, namely that data has significant impact on the headers of all protocol functions. Some data manipulations modify the data and may even change the size of data (marshalling and encryption). The size of a minimum processing unit for data manipulations differs at each level.

3.2 ILP Implementation

3.2.1 Macros versus function calls

An important issue is how to implement the different data manipulation functions. To get efficient implementations, function calls should be avoided. A much more efficient solution is macro inlining. However, macros do not allow the dynamic adaptation of data manipulation functions depending on varying application needs or on quickly changing network characteristics such as congestion situations. Using function calls and function pointers instead supports a dynamically adaptable implementation, but experiments have shown that substituting macros by function calls results in the loss of all performance benefits gained by ILP in the first place. The experiments also show that inling of data manipulation functions increases the code size only by a few percent (approximately 3% in the example application).

3.2.2 Sending path

The data flow of the sending path is shown in Figure 3. In the non-ILP implementation the data is marshalled, encrypted, and copied from the application buffer into the TCP buffer. These operations require at least three write and three read accesses to the memory. The checksum is calculated by another read operation before the data is finally copied into a kernel buffer. Steps 1 (marshalling) and 2 (encryption) are called directly by the application program, while step 3 (copying) is performed by the TCP interface procedure `tcp_send`. The main TCP procedure for sending data (`tcp_output`) performs step 4 (checksum calculation) and invokes step 5 (system copy). To increase the probability of cache hits, the marshalling and encryption loops of the non-ILP implementation follow each other. Avoiding the system copy in step 5 would require a modification of the operating system and the network adapter architecture according to the proposals in [12][13][14][15].

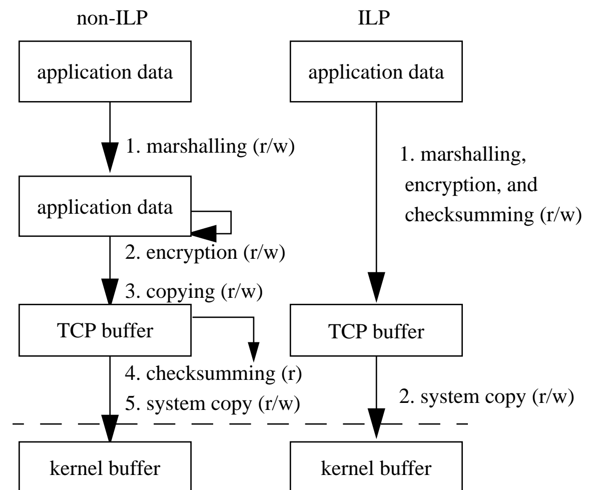


Figure 3. Data movements for sending data

Placement of the ILP loop

The first issue in designing the ILP implementation is to decide where to perform the data manipulation routines. In traditional layered implementations, higher level protocols are implemented within the application space, while lower level protocols (transport protocols and below) reside in the kernel space or even on a network adapter. In such an architecture, data has to cross the application/system boundary. If there are data manipulations, which both read *and* write data, the elimination of all copy procedures would not result in large benefits; e.g., when calling an encryption function, data to be encrypted must first be copied into another buffer to avoid overwriting the original application data for their re-use. Moreover, another data copy is required for possible retransmission at the transport level. Consequently, not all copy procedures can be avoided. In the implementation of the protocol suite, at least one copy procedure is necessary to copy data from application memory to the TCP buffers due to the simultaneous existence of encryption and retransmission.

The data manipulations have been integrated into this copy procedure without additional costs. The ILP implementation marshals and encrypts the data while being copied from the application to the TCP buffer. Two copy operations and one read access have been saved. Because TCP uses a ring buffer, to which the data is transferred during the ILP loop, the structure of the TCP buffer (i.e. the length and some buffer state information such as the actual write pointer) must be known during the ILP loop. The ILP loop also has the task to align the data to the ring buffer structure.

A problem might occur if there is not enough space for a message in the TCP buffer at the time the ILP loop is invoked. For example, the retransmission buffer may contain many unacknowledged data such that a packet of maximum size (= MTU size) cannot be stored. Data manipulations can be performed as early as possible to minimize delays [17]. Data above the TCP level is manipulated in advance; the checksum calculation and the copy to the TCP buffer are done when there is enough buffer space available again. The delay saved by this approach is not significant (approximately

100 µs on a SUN SPARCstation 10-30) compared to the total delay including driver processing and latency of the network, which is usually in the millisecond range. Since the implementation itself would become more complex, we decided to perform all data manipulations within a single loop and to delay all manipulations until they are all possible. If there is not enough TCP buffer, then all data manipulations are delayed until there is enough buffer space available again.

ILP Processing Steps

As mentioned in Section 2, it is not usually possible to process user data independently from protocol headers. It is also not possible to start the ILP loop at any byte in the data field, and then process the following data in the original order. These problems are solved by dividing a message into several parts as illustrated in Figure 4. The resulting parts are processed consecutively, starting with part B and finishing with part A.

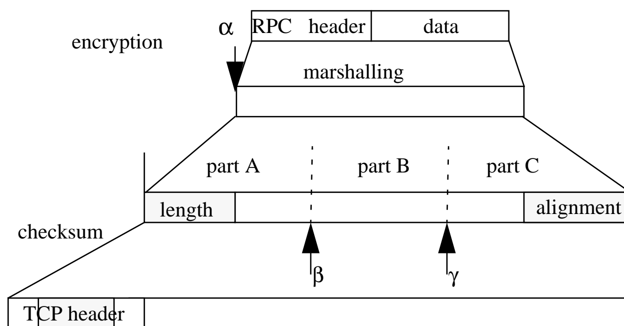


Figure 4. ILP processing steps

1. The marshalling routine is invoked by the application for a request or a response message. The implementor must know the size of the encryption header, because encryption is aligned to 8 bytes and the encryption header, which depends on the length of the data, is also encrypted. The first byte of the data part, which can be encrypted in the first step, is 8 bytes after the start of the encryption header (position β), while marshalling begins directly after the encryption header, that means 4 bytes after the start of the encryption header (position α). The problem that marshalling and encryption begin on different positions could be avoided by moving the encryption header field to the end of the message. The same problem would occur for an intermediate layer between marshalling and encryption with a header not aligned to the processing unit size of encryption (8 byte). Message part B beginning at position β is the first one processed by the data manipulation functions.

2. Because the length of the message obtained by marshalling (including the encryption header) is usually not aligned to 8 bytes, it may also happen that the last bytes of the marshalled message cannot be passed from the marshalling procedure to the data manipulation functions without adding the necessary alignment bytes. The last 8 bytes (beginning at position γ) including the alignment bytes build part C of the complete message.

3. After passing part C to the data manipulations, the length of the marshalled message is known and the length field (encryption header) is completed. The first 8 bytes (part A: encryption header and the first 4 bytes of the marshalled message) are passed to the data manipulations.

3.2.3 Receiving path

The non-ILP implementation evaluates the checksum after the system copy (`tcp_input`), decrypts the data, and unmarshalls the decrypted data (Figure 5). The last two steps are invoked by the application program. Compared to the sending side, one read and one write access can be saved.

The fact that the size and the beginning of the application data is usually unknown before unmarshalling requires an additional copy besides the system copy procedure for receiving. The copy operation is performed together with unmarshalling in the non-ILP case, while it is integrated with all the other data manipulations in the ILP case.

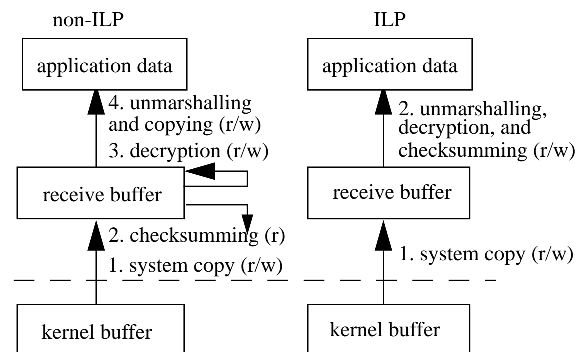


Figure 5. Data movements for receiving data

There are two reasonable locations for manipulating the received data in user space. The data can be manipulated very close to the read system call, i.e. directly after the system copy, or it can be manipulated very close to the application operations. Manipulating the data very close to the read system call brings the additional benefit that the data is probably still in the cache before the data manipulations. On the other hand, if the data is manipulated very close to the application operations, the application can benefit from the higher probability that data is within the cache before application processing. Experiments show that both approaches yield nearly identical performance. Moving the data manipulations closer to the application resulted in a very small performance benefit (5 µs on a SUN SPARCstation 10-30) for packet processing during receiving and delivering a message. Manipulating data very early during processing the receive functions has the advantage that possible errors such as checksum errors or errors because of an illegal data format for marshalling are known very early. Possible errors are known before TCP control processing so that the TCP processing can proceed without a possible roll back later on. If the data is manipulated close to the application, several TCP control actions such as acknowledgment generation must be delayed and other operations such as updating the connection state must be taken back. Therefore, the data manipulation functions have been implemented directly after the system copy operation.

ILP processing for receiving a message is easier than for sending. After receiving a TCP message, the first 8 bytes containing the encryption header are decrypted. Since this operation also decrypts the first word of the marshalled message, an offset parameter must indicate to the unmarshalling function that the next decryption operation shall start one word after the first marshalled word. The unmarshalling function is invoked by `tcp_input` and is enhanced by the other data manipulations. As soon as enough data

is decrypted for unmarshalling, it performs the appropriate unmarshalling operations. At the end of the unmarshalling routine, the application message is reconstructed. Before invoking the unmarshalling procedure the application has to decrypt the encryption header.

4 Performance Results

4.1 Comparison of the ILP and the non-ILP implementations

Figure 6 - Figure 10 compare the performance of the ILP and the non-ILP implementations for different packet (TPDU) sizes on different workstations. To get these performance numbers, a 15 kbyte file with varying message sizes has been transmitted several times from a server (sender) to a client (receiver) on the same machine using UDP in loop back mode. UDP without checksum is nearly as fast as our TCP kernel part. Packet processing times include all data manipulations within the application space, which are necessary to send or to receive a message.

The measured results show that throughput increases with the message size because of the lower number of messages required to transmit a file. They also show that the performance gaps between the ILP and the non-ILP implementations increase nearly proportionally to the packet size. Clearly, the benefits gained by doing an ILP implementation of the data manipulations are higher when data manipulations dominate the total processing cost. We also observe the impact of a second-level cache. The SS10-30 workstation does not include such a cache, and the throughput for 1280-byte packets is lower than for 1024-byte packets. The performance figures for other workstations (SS10-41, SS20-60, AXP3000/800) with second-level cache show a throughput increase for 1280-byte packets.

The integration of encryption and checksum calculation into marshalling yields a decrease of 58 µs (16%) for sending and of 56 µs (16%) for receiving a 1 kbyte packet on a SUN SPARCstation 10-30 (Figure 6 and Figure 7). On a SUN SPARCstation 20-60, packet processing for sending could be decreased by 50 µs (24%) for sending and by 41 µs (20%) for receiving. The total difference of packet processing (in terms of µs) between ILP and non-ILP decreases for the faster machine, but the relative benefits (in percentages) increase. See Section 4.2 for further explanation and the Annex for the complete results.

The benefits of ILP on DEC AXP3000 workstations are smaller than on the SUN SPARCstations. This is also explained in Section 4.2. For receiving 1 kbyte packets on an AXP3000/800 (200 MHz) the difference of 12 µs (8%) for packet processing is relatively small. Sending is 25 µs (13%) faster.

Relative performance increase brought by an ILP implementation in terms of throughput is always lower than the performance benefit considering only the packet processing time (Figure 8 and Figure 9). There are also other operations besides data manipulations which have significant impact on the total throughput, e.g., IP and network driver processing, task switches, or workstation background load. Data manipulations of the ILP implementation consume approximately the same time as the system operations.

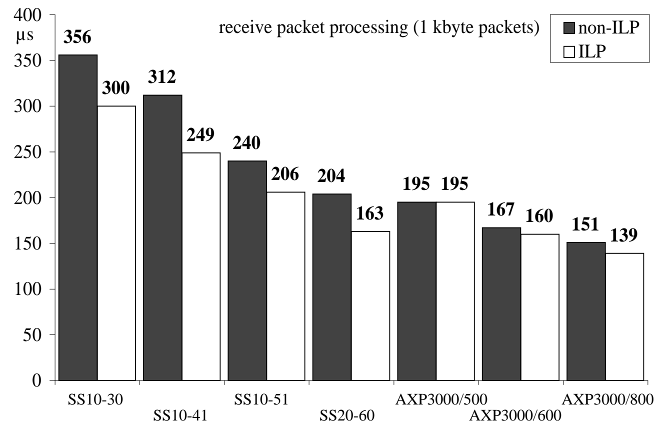


Figure 6. ILP and non-ILP receive packet processing times

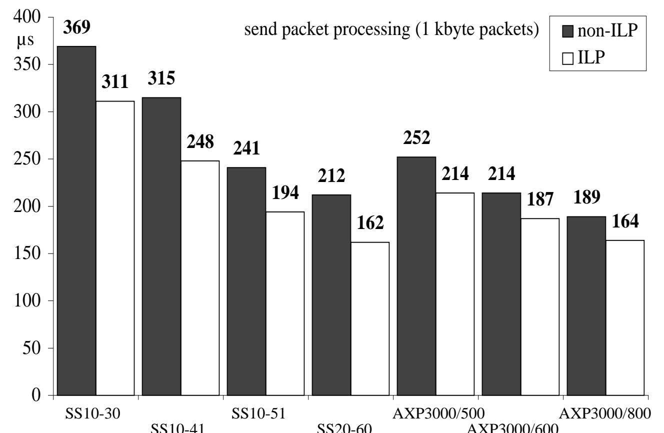


Figure 7. ILP and non-ILP send packet processing times

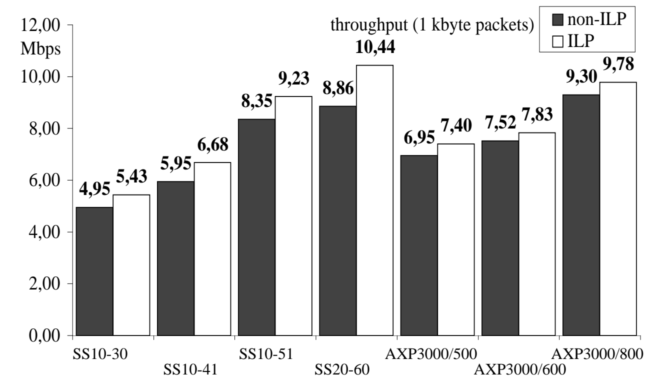
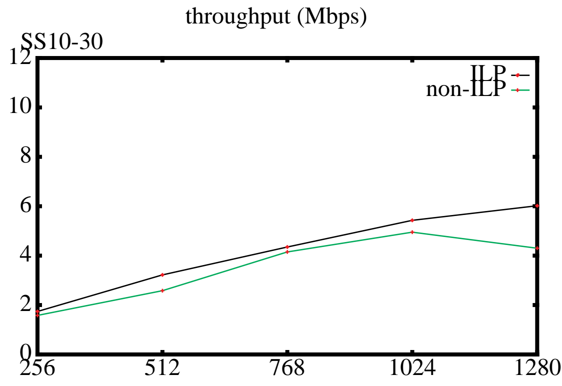
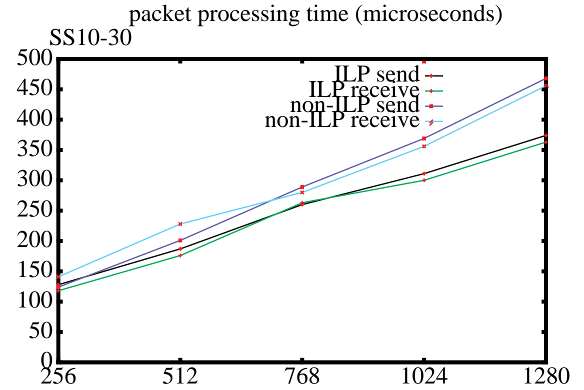
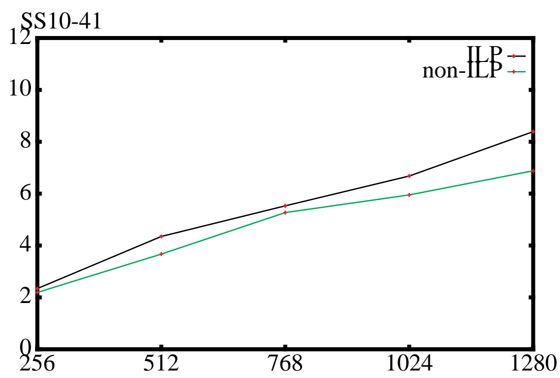
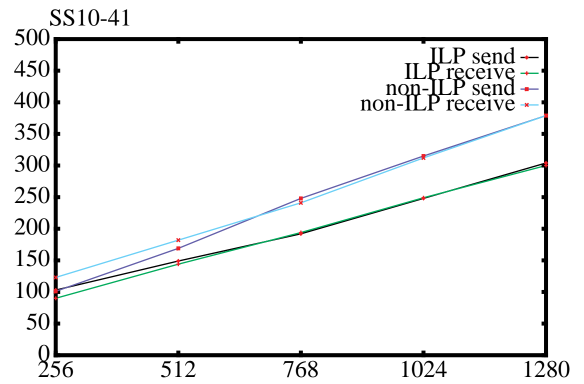


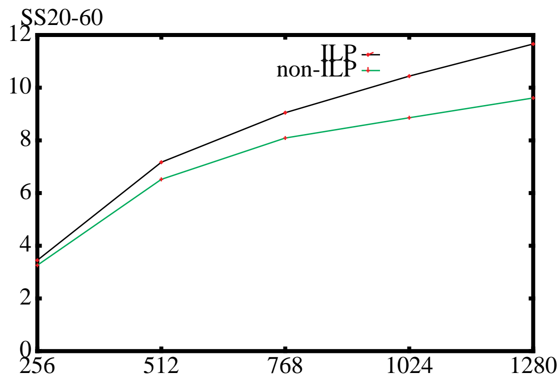
Figure 8. ILP and non-ILP throughput

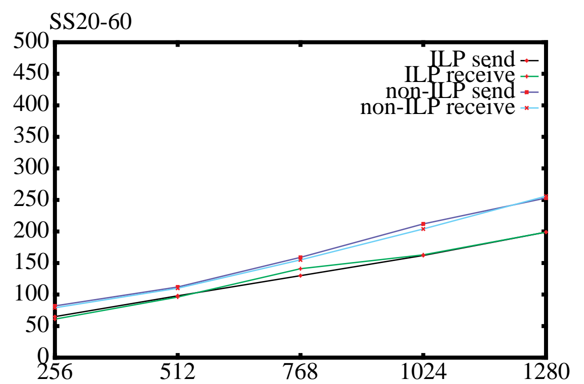












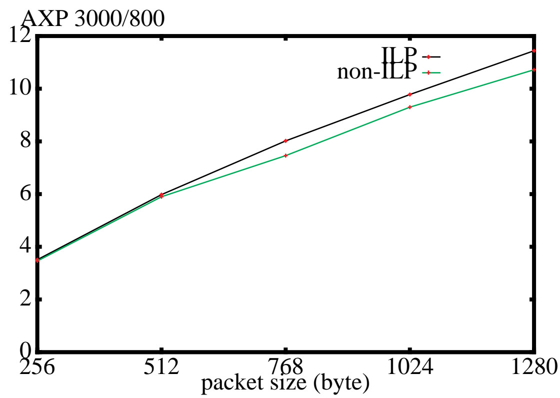


Figure 9. ILP and non-ILP throughput

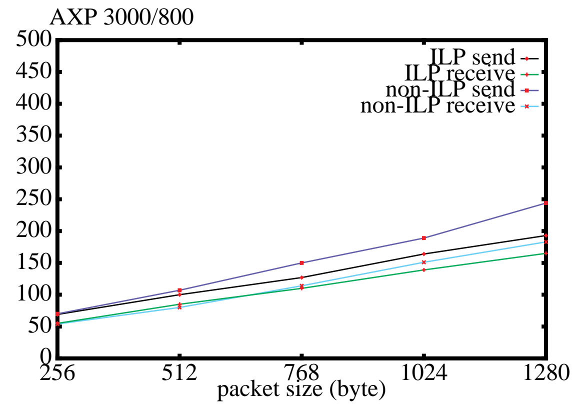


Figure 10. ILP and non-ILP packet processing times

In particular, the operating system on DEC Alpha workstations (specifically OSF/1 version 1.3 and 2.0) causes a very high overhead in the experiment. The system overhead of the SPARCstation 20-60 running Solaris 2.3 is lower and ILP improvements are more clearly visible there. Using more advanced systems, e.g. zero-copy network adapters [13][14][15] and dedicated operating system support with less system overhead, could raise the benefits from ILP further. Therefore, we expect ILP benefits to become more significant in systems with more efficient network adapters and improved integration of communication systems into the operating systems.

Another issue which has significant impact on the total throughput is the complexity of the data manipulations. Replacing the encryption/decryption algorithm by a very simple algorithm similar to the one used in [2] yields similar total improvements in packet processing times (70 µs less for sending a 1kbyte packet and 64 µs less for receiving a packet on a SPARCstation 10-30) compared with the simplified SAFERK64 algorithm (Figure 11). The relative improvements (32 and 40%) are significantly higher. Furthermore, the throughput difference between ILP and non-ILP is higher than with the more complex encryption function (Figure 12).

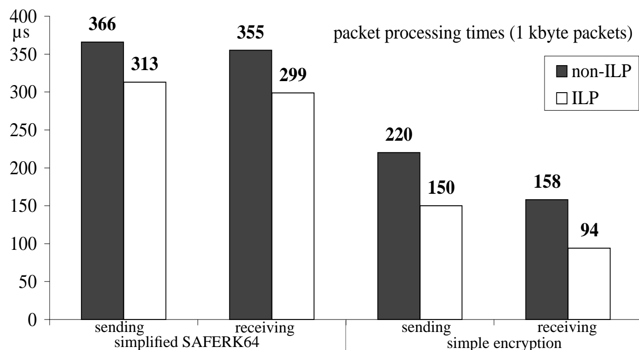


Figure 11. Packet processing of ILP and non-ILP implementation with different encryption functions

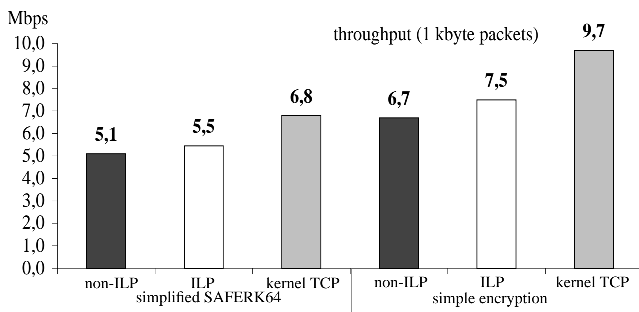


Figure 12. Throughput of ILP and non-ILP implementation with different encryption function

Figure 12 compares the throughput performance of the user-level implementations (ILP and non-ILP) and the non-ILP implementation with TCP in kernel space. In the experiment, 1 kbyte messages have been exchanged. As expected, the implementation using the BSD TCP kernel version is faster than both user-level implementations. Protocols in user space do not usually achieve kernel performance, but nevertheless they have several advantages [20][21][22][23]. However, the performance gap could be minimized by optimizing the data path, which is usually the critical path of protocol implementations. For receiving a packet the ILP implementation including TCP processing, decryption, and unmarshalling was even faster than only decryption and unmarshalling on top of kernel BSD TCP. One reason for better BSD TCP kernel throughput is that the code is more optimized and acknowledgment packets do not cross the user/kernel domain as it does in a user-level TCP implementation. A solution to this problem would be to separate control processing and data manipulations supported by different packets for control messages and data.

4.2 Analysis of memory and cache access

To understand the reasons for the performance gain illustrated in Section 4.1, the memory access and cache behavior of the application using the ILP stack and the non-ILP stack have been simulated. These simulations have been performed using a cache simulator called "cachesim", which is part of the shade analyzing tools from SUN Microsystems [18]. The tool calculates the number of memory accesses and the number of cache misses. The "atom" tool [19] has been used here to investigate the memory behavior on DEC AXP workstations. It simulates the memory system of AXP 3000/500 (150 MHz) models with 8 KB on-chip (first-level) data (write-through), 8 KB instruction cache and 512 KB external (second-level) cache.

The main reason for the performance gain on the SUN SPARCstations is the significant reduction in the number of memory accesses (Figure 13). ILP reduces the number of memory accesses by $25.7*10^6$ ($13.7*10^6$ 4-byte-reads and $12*10^6$ 4-byte-writes less) for sending 10.7 Mbyte of data. This means that the ILP implementation reads 55 Mbyte less and writes 48 Mbyte less than the non-ILP implementation, which is equivalent to saving more than 4 copy operations. For receiving the same amount of data, ILP decreases the number of memory accesses by $17.6*10^6$ because of the lower number of 4-byte-write ($8.3*10^6$ accesses less) and 4-byte-read ($8.4*10^6$ accesses less) accesses. Here, ILP writes and reads 33 Mbyte less data than the non-ILP implementation, which is equivalent to saving 3 copy operations.

ILP was initially thought as a technique which would benefit from the presence of caches. A surprising result is that the relative amount of first-level data cache misses is higher in the ILP case. First-level instruction cache misses and second-level data cache misses are negligible in the experiments on the SUN SPARCstations. For sending, the cache miss ratio is slightly higher in the ILP case. On the receiving side, the cache miss ratio increased from 4.7% to 18.7% (Figure 14). The results show that even with a careful implementation technique, a relative high rate of cache hits can be achieved without integrating several functions into a single loop, but only by processing the different loops close to each other.

On the sending side, the cache miss ratio increase for ILP can be explained by the fact that there are always a fixed number of write cache misses for writing the data into the TCP retransmission buffer, which cannot be avoided. The cache miss ratio increases because ILP reduces the total number of memory accesses in a more significant way than the number of cache misses.

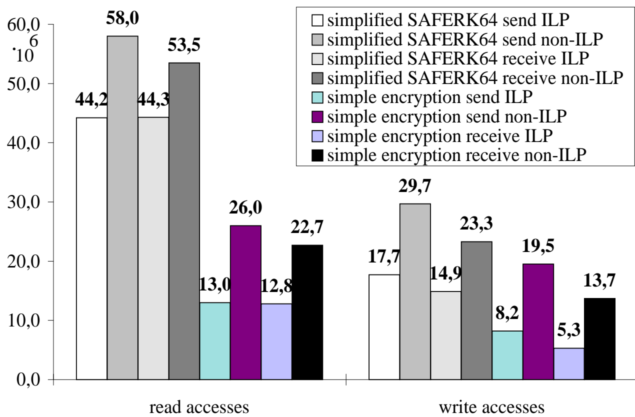


Figure 13. ILP and non-ILP memory access

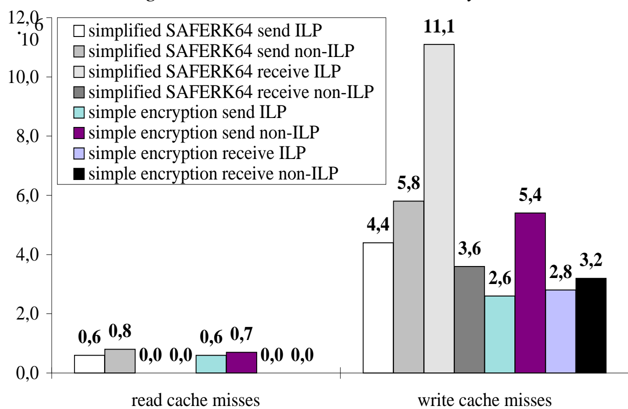


Figure 14. ILP and non-ILP cache misses

A more detailed analysis shows that the number of 4-byte-cache misses for ILP is half of that for non-ILP on the sending side. However, the number of 1-byte-cache misses increases significantly from $0.03*10^6$ to $2*10^6$. On the receiving side, the number of write cache misses increases even more (from $3.6*10^6$ to $11.0*10^6$). This difference is caused by the number of 1-byte-cache misses. The reason for the increased number of 1-byte-cache misses are characteristics of the simplified SAFERK64 encryption algorithm. The encryption and decryption functions manipulate data on a 1-byte basis and they write single bytes into the memory. The cache misses are higher for receiving, because the decryption implementation requires more variables for intermediate results than for encryption. The algorithm uses two tables (pre-calculated logarithm and exponential tables with values for each possible byte) and a byte vector, which must be accessed for each byte to manipulate. In the non-ILP case, these memory areas can remain in the cache while a message is being processed. In the ILP case, these tables must be loaded more frequently into the cache because two operations for encrypting two 8-byte units do not follow each other.

Using the very simple encryption function mentioned in Section 4.1 (which uses constant values instead of tables for manipulating the data) yields in a lower number of cache misses. On the sending side, ILP decreases the number of cache misses by 50%. On the receiving side, the number of cache misses decreases slightly in contrast to the increase of the number of cache misses for ILP with the simplified SAFERK64 encryption algorithm.

The results show that the use of simpler but less realistic encryption algorithms results in a better performance and more ideal cache behavior than more complex and more realistic ones. The type of data manipulations (in this case: the encryption algorithm) strongly influences the success of ILP. If possible, one should therefore design data manipulation functions that are well fitted for ILP, e.g. by avoiding single-byte cache misses.

Another surprising result is that the non-ILP implementation achieves a relatively good cache behavior. Nearly all operations operate on the first-level cache or on the second-level cache and do not require access to the main memory. This is the reason for the fact that the absolute difference of packet processing times decreases with increasing processor speed, but the relative difference (in %) increases.

Similar effects have also been observed on the DEC AXP workstations. For sending the memory system time is 0.494s for ILP and 0.539s for non-ILP. The whole execution time for the application program is 2.466s for ILP and 2.725s for non-ILP. For receiving the difference for the memory system time was nearly the same for the ILP (0.292s) and the non-ILP (0.295s) case. The application program execution time for receiving is 2.335s for ILP and 2.427s for non-ILP.

Similar to the SUN SPARCstations simulations, ILP improvements on DEC AXP workstations are more significant for sending than for receiving. The overhead for read cache misses could only be reduced for sending. The additional memory system time caused by writes decreases for sending and for receiving.

In contrast to the SUN SPARCstations, a significant number of cache misses can be observed on the DEC AXP workstations because of the smaller instruction cache. In the ILP case, the number of instruction cache misses is higher than in the non-ILP case and it consumes 24-28% of the memory system time. The higher instruction cache misses are an important reason for the lower ILP benefits on the DEC Alpha workstations.

5 Conclusion

This paper presents an experimental ILP implementation of a three level protocol suite based on a user-level TCP and its performance evaluation. ILP reduces the number of memory accesses up to 30%, but the relative amount of cache misses could not be reduced compared with a carefully designed non-ILP implementation. ILP throughput improvements are limited and depend heavily on several issues such as the complexity of data manipulations, the communication subsystem architecture, and the host environment characteristics. In our experiments, these issues decrease the throughput gain to 10-20% in contrast to the 50% gain achieved for simple loop experiments [1]. ILP is very sensitive to various issues, which makes its use debatable in existing communication systems and workstations.

The main limitation of ILP is that it is only applicable with certain types of protocol functions (non-ordering constrained functions) and protocol architectures (header size must be known before data manipulation processing). Another major drawback of ILP is the reduced flexibility, because the use of macros instead of function calls is required to avoid performance loss. Macros do not allow a protocol implementation to be adapted dynamically to changing application requirements or to varying network characteristics.

The implementor has to decide depending on application and system characteristics whether it is worth to apply ILP with all advantages and drawbacks. Using advanced protocol features such as non-layered architectures [24], fixed size headers, trailers for data dependent fields, different packet types for control information and data, uniform processing unit sizes for different data manipulation functions could be advantageous for ILP. These features should be studied in future protocol designs.

Acknowledgments

Many reviewers helped us to improve our paper by valuable comments. We want to thank Ernst Biersack, Jean-Chrysostome Bolot, Jon Crowcroft, Philipp Hoschka, Christian Huitema, Antony Richards, Ellen Siegel, and Martina Zitterbart. Thanks also to Anna Hoglander and Vincent Roca for their support in the design and the implementation of the user-level TCP.

References

[1] Clark, D.D.; Tennenhouse, D.L.: Architectural Considerations for a New Generation of Protocols, ACM SIGCOMM 1990, pp. 200-208

[2] Abbott, M. B.; Peterson, L.L: Increasing Network Throughput by Integrating Protocol Layers, IEEE/ACM Transactions on Networking, Vol. 1, No. 5, October 1993, pp. 600-610

[3] Partridge, C.; Pink, S.: A Faster UDP, IEEE/ACM Transactions on Networking, Vol. 1, No. 4, August 1993, pp. 429-440

[4] Gunningberg, P.; Partridge, C.; Sirotkin, T.; Victor, B.: Delayed Evaluation of Gigabit Protocols, Proceedings of the 2nd MultiG Workshop, June 1991

[5] Feldmeier, D.C.: Multiplexing Issues in Communication System Design, ACM SIGCOMM 1990, pp. 209-219

[6] Tennenhouse, D.L.: Layered Multiplexing Considered Harmful, Protocols for High-Speed Networks I, 1989, North-Holland, pp. 143-148

[7] Feldmeier, D.C.; McAuley, A.J.: Reducing Protocol Ordering Constraints to Improve Performance, Protocols for High-Speed Networks III, 1992, North-Holland, pp. 3-18

[8] Huitema, C.: MAVROS: Highlights on an ASN.1 compiler, INRIA technical report

[9] Massey, J.: SAFER K-64: A Byte-Oriented Block-Ciphering Algorithm, Lecture Notes in Computer Science 809, Springer-Verlag, 1993, pp. 1-17

[10] Feldmeier, D.C; Karn, P.R.: UNIX Password Security - Ten Years Later, in: Advances in Cryptology - CRYPTO '89, Lecture Notes in Computer Science 435, Springer-Verlag, 1990, pp.44-53

[11] Hoglander, A.: Experimental Evaluation of TCP in User Space, INRIA technical report, September 1994

[12] Druschel, P.; Peterson, L.L.: Fbufs: A high-bandwidth cross-domain transfer facility. Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, December 1993, pp. 189-202

[13] Metzler, B.; Miloucheva, I.: Design and Implementation of a Flexible User Protocol Interface, Proceedings of the 1st International Workshop on High Performance Protocol Architectures, December 15-16, 1994, Sophia-Antipolis, France

[14] Ahlgren, B., Gunningberg, P.: A minimal copy network interface architecture supporting ILP and ALF, Proceedings of the 1st International Workshop on High Performance Protocol Architectures, December 15-16, 1994, Sophia-Antipolis, France

[15] Sterbenz, J.P.G, Parulkar, G.M.: Axon: A High-Speed Communication Architecture for Distributed Applications, IEEE Infocom' 90

[16] Diot, C.; Huitema, C; Turletti, T.: Multimedia Applications should be adaptive. Submitted to the 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems, August 23-25, 1995, Mystic, Connecticut

[17] Oechslin, P.; Leue, S.: Enhancing Integrated Layer Processing Using Common Case Anticipation and Data Dependence Analysis, Proceedings of the 1st International Workshop on High Performance Protocol Architectures, December 15-16, 1994, Sophia-Antipolis, France

[18] Sun Microsystems Inc.: Introduction to Shade, April 1993

[19] Digital Equipment Corporation: ATOM Reference Manual, December 1993.

[20] Jain, P.G.; Hutchinson, N.C.; Chanson, S.T.: A Framework for the Non-Monolithic Implementation of Protocols in the x-Kernel, Usenix, August 1994, High-Speed Networking Symposium, pp. 13-30

[21] Thekkath, C.A.; Nguyen, T.D.; Moy, E.; Lazowska, E.D.: Implementing Network Protocols at User Level, IEEE/ACM Transactions on Networking, Vol. 1, No. 5, October 1993, pp. 554-565

[22] Maeda, Ch.; Bershad, B. N.: Protocol Service Decomposition for High-Performance Networking, 14th ACM Symposium on Operating Systems Principles, December 5-8, 1993

[23] Biersack, E.W.; Rütsche, E.; Unterschütz, T: Demultiplexing on the ATM Adapter: Experiments with Internet Protocols in User Space, Proceedings of the 1st International Workshop on High Performance Protocol Architectures, December 15-16, 1994, Sophia-Antipolis, France

[24] Crowcroft, J.; Wakeman, I.;Wang, Z.: Layering Considered Harmful, IEEE Network, Vol. 6, No. 1, January 1992

[25] Sun Microsystems, Inc: XDR: External Data Representation standard, RFC 1014, June 1, 1987

Annex

TABLE 1. Packet processing and throughput of ILP and non-ILP implementation

system platform	packet size	ILP	non-ILP	ILP send	ILP receive	non-ILP send	non-ILP receive
		throughput		packet processing time (μs)			
SUN	256	1.74	1.58	128	118	124	141
SPARCstation	512	3.22	2.58	187	176	201	228
10-30	768	4.35	4.15	260	263	289	280
SunOS4.1.3	1024	5.43	4.95	311	300	369	356
	1280	6.02	4.3	374	363	468	456
SUN	256	2.34	2.19	103	90	101	123
SPARCstation	512	4.35	3.67	149	144	169	182
10-41	768	5.53	5.27	192	194	248	241
SunOS4.1.3	1024	6.68	5.95	248	249	315	312
	1280	8.39	6.88	304	300	379	379
SUN	256	3.02	2.64	77	72	91	88
SPARCstation	512	5.41	4.69	124	116	147	147
10-51	768	7.78	7.01	158	158	202	195
SunOS4.1.3	1024	9.23	8.35	194	206	241	240
	1280	9.48	8.65	239	248	301	310
SUN	256	3.45	3.26	65	61	82	79
SPARCstation	512	7.17	6.52	98	96	112	110
20-60	768	9.05	8.09	130	141	159	155
Solaris 2.3	1024	10.44	8.86	162	163	212	204
	1280	11.66	9.61	199	199	253	256
DEC	256	2.52	2.53	100	73	103	73
AXP 3000/500	512	4.43	4.30	135	109	149	120
150 MHz	768	6.07	5.72	174	156	195	163
OSF/1 1.3	1024	7.40	6.95	214	195	252	195
	1280	8.59	8.07	252	227	302	237
DEC	256	2.57	2.59	85	74	86	73
AXP 3000/600	512	4.36	4.39	122	93	137	109
175 MHz	768	6.36	6.12	146	127	162	140
OSF/1 2.1	1024	7.83	7.52	187	160	214	167
	1280	8.98	8.56	227	191	256	201
DEC	256	3.51	3.46	69	55	70	54
AXP 3000/800	512	5.98	5.90	100	85	107	80
200 MHz	768	8.02	7.46	127	110	150	114
OSF/1 2.1	1024	9.78	9.30	164	139	189	151
	1280	11.44	10.72	193	165	244	183