Limitations and Implementation Experiences of Integrated Layer Processing

Torsten Braun[1]

INRIA, 2004 route des Lucioles, F-06902 Sophia-Antipolis, email: t.braun@ieee.org

Abstract. Integrated Layer Processing (ILP) has several limitations on current system environments. Performance evaluations show that the main benefits from ILP are rather a reduced number of memory accesses than an improved cache hit rate. Data manipulation characteristics have significant impact on the cache behavior and on the achievable performance gain of ILP. Future protocol characteristics and efficient implementation concepts for communication systems with CPU expensive data manipulations are proposed.

1 Introduction

The performance of processors has been increasing faster than the performance of memory during the last few years. The resulting memory bottleneck can be reduced by avoiding memory access, e.g. by eliminating copy operations from protocol implementations. Another approach is to use on-processor caches, which are almost as fast as registers. Data and instruction caches are currently in the range of a few kbytes, but it is expected that on-chip caches of future processors will grow into the range of several Mbytes [1].

Integrated Layer Processing (ILP) tries to avoid memory access and to use caches in an efficient way. ILP is an implementation concept which allows to implement all data manipulation steps in one integrated processing loop (ILP loop) [2]. 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. 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- or 64-bit-word.

In contrast to that, a non-ILP implementation reads a complete packet or data unit and performs one function after the other. Because intermediate packets must be 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 in this case often possible to keep the data within the cache between two different functions. Especially, application layer framing (ALF) [2] is advantageous to avoid cache misses. ALF is a precondition for applying ILP and means that the application breaks the data into suitable aggregates called application data units (ADUs), and the lower levels preserve these frame boundaries as they process the data. Therefore, no segmentation, blocking, or other size changing functions are allowed. Another advantage of the fact that an ADU is the logical unit of *all* protocol functions is that a ADU can be processed independent of other ADUs. This prevents buffering packets in packet queues as it is for example necessary for reassembly. Buffering is required if packets are split into several packets or if several

1. sponsored as a research fellow by the Commission of the European Communities under the Human Capital and Mobility Programme HCM (no. ERBCHBGCT930254)

packets have to combined into a single one. ALF increases the possibility that data are within the cache while being processed by a pipeline of consecutive protocol functions. Especially with small packet sizes and the expected increase of future on-processor caches the probability increases that a packet fits into an on-processor (first-level) cache.

By applying ILP to simple data manipulations like data copying and checksum calculations, performance gains of 50% have been achieved [2]. The advantages by integrating data manipulations may increase with the number of integrated data manipulation functions [3]. Other experiments proved that expensive manipulations such as DES (data encryption standard) can hide ILP performance gains [4].

2 ILP Limitations

ILP can only be applied between operations that combine or split PDUs such as (de)multiplexing, concatenation, and segmentation. These operations must be avoided to ensure that single data units do not interfere with other data units and preserve their frame boundaries according to ***ALF***. Other reasons to avoid multiplexing are quality-of-service support for single applications and various other efficient protocol implementation techniques.

Another major limitation is that ILP can only be applied with ***non-ordering constrained data manipulations*** such as TCP checksum or block cipher encryption algorithms. An ordering constrained function means that a packet must be processed serially according to the definition in [6]. Examples of ordering-constrained functions are cyclic redundancy code (CRC) calculation and stream cipher encryption algorithms, which require that data are processed in the correct order. Processing the second part depends on the result of the first part, i.e. the data part cannot be processed before knowing the result of processing the part containing header fields. However, even with non-ordering constrained functions, it is necessary to know how many bytes to be processed will be before a given block, because both operations need to be aligned on certain boundaries. Because of this fact, ILP can only be applied if the header size is known before processing the data. The header size must either be constant or it is necessary that it can be at least pre-calculated dependent on the protocol state.

Unfortunately, data manipulations have also a significant impact on the contents of header fields, e.g., checksum fields, alignment bytes for encryption, and length indicator fields. 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-1 data manipulation function. To overcome this problem there are the following possibilities:

- Divide a packet into several pieces and process the packet in a non-consecutive order beginning with the first part containing only user data. The order, in which the different pieces are processed depends on the relations between the pieces. This concept is a generalization of the approach of segregated messages [3], which make a clear separation between protocol headers and application data. ILP is only applied to application data.
- In some cases the header modifications required by the data manipulation function can be delayed and performed after other fields of the header have been composed. The delayed results can be updated by lower functions if the results are delivered

from the upper function via a separate interface. For example, it is possible to calculate the TCP checksum even while the data are being copied into a network FIFO buffer, if the FIFO buffer is large enough to store the complete packet and if it is possible to modify the TCP checksum field at the output of the FIFO buffer. This approach requires that the lower level function must have a detailed knowledge of the higher level function and that the lower functions get the result of the upper function early enough to be able to proceed processing.

Ordering constraints on the treatment of data occur because protocol processing consists of ***interactions between data manipulation functions and control tasks*** (such as header and connection processing). Often, data manipulation functions deliver a result which must be interpreted by a control function as a correct or corrupted information. The „3-stage-approach“ proposed in [3] to solve these ordering constraints divides protocol processing into „initial control operations“, „integrated data manipulations“ (or ILP loop), and a „final protocol stage“ (Figure 2).

Processing unit sizes may differ for different functions. An XDR marshalling procedure operates usually on 4-byte units, while encryption functions usually manipulate the data on 8-byte units. „Word filters“ are used in [3] to solve the mismatch which occurs when data pass from one data manipulation function to another one. Word filters output data as soon as data are ready, but do not take into account whether it is more efficient for the next function to process the data on a larger size. Furthermore, hardware constraints such as the bus width should be considered to choose the appropriate processing unit size. Considering a protocol stack with encoding on 4 bytes, encryption on 8 bytes, and checksum calculation on 2 bytes, the word filter mechanism passes data in 4-byte units whatever the size of data manipulated by each function is. To evaluate the checksum on a 4 byte word, 2 write operations are required. The word filter concept could be optimized to take advantage of each basic processing unit size, in order to minimize size modifications when passing data from one function to another. Passing data in 8 byte processing units from encryption to checksum reduces the write operation for checksum calculation to one.

Integrating code from different protocol layers may violate the implementation ***modularity***. Preserving modularity in ILP implementations may be achieved by an automatic synthesis tool, e.g., based on a macro preprocessor [3], or by an ILP stub compiler to generate the protocol code.

3 ILP Experiments and Results

To estimate the achievable ILP benefits, a complete protocol stack consisting of three data manipulation functions has been implemented. This Section describes the implementation experiments and results briefly. For details see [7].

3.1 Experimental protocol stack

A file transfer application is used as an example application. The message formats are described using ASN.1. The appropriate (un)marshalling routines have been generated with an ASN.1 stub compiler. XDR is used as data representation. The marshalled message is encrypted using a fast encryption algorithm based on the SAFERK64 algorithm [8]. This algorithm has been chosen, because DES can hide totally the ILP performance gain [4]. To reduce the still too high overhead, the algorithm has further been simplified. The encrypted message is finally copied into a TCP buffer. TCP calculates the checksum by another read operation before the data is copied into a kernel buffer.

Received messages are decrypted, unmarshalled, and delivered to the application. To increase the probability of cache hits, marshalling and encryption follow each other.

The TCP implementation is divided into a part running within the kernel and a library to be linked to each application running in user space. For sending, the main task of the kernel part is passing the messages received from the user-level TCP to IP. For receiving, the kernel part demultiplexes IP packets to the corresponding TCP connection, i.e. to the corresponding application. The TCP is tailored in order to achieve a good ILP performance, although it is fully compatible with other TCP implementations:

- TCP header options are avoided to ensure a ***fixed-size header***.
- TSDUs are mapped to a single TPDU at TCP level according to the ***ALF*** concept.
- A connection is only used for uni-directional data transfer. In one direction mainly data packets are transferred, while the other direction contains acknowledgment packets only. This results in a ***separation of control and data packet processing*** to simplify header prediction.

3.2 ILP Implementation

For ILP, the data manipulations on the send path have been integrated into the copy procedure from application memory to the TCP buffer. 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 for sending compared to the non-integrated implementation. However, the data manipulations must have a detailed knowledge about the data structures of the TCP buffer. For receiving, ILP saves one read access and one copy.

To achieve good ILP performance macros have been used instead of function calls for data manipulations. However, macros do not allow the dynamic adaptation of data manipulation functions depending on varying application needs or quickly changing network characteristics such as congestion situations. Function calls and function pointers instead would support a dynamically adaptable implementation, but experiments showed that substituting macros by function calls results in the loss of all performance benefits gained by ILP in the first place. The overhead for function calls is very high in general and the number of function calls necessary for ILP is much higher.

3.3 Performance Results

The measurement results show that the performance gap between ILP and non-ILP implementations increases for larger packets [7]. The ILP benefits are higher when data manipulations dominate protocol processing. Relative performance increase by ILP in terms of throughput is always lower (approximately 10%) than the performance benefit considering only the packet processing time (approximately 20%). System operations such as IP and network driver processing consumed approximately the same time as packet and protocol processing in user space. Another issue with significant performance impact is the complexity of the data manipulations. Replacing the encryption algorithm by a very simple algorithm similar to the one used in [3] yields significantly lower packet processing times (approximately 35% less) and higher throughput (approximately 15% more) for ILP compared to non-ILP.

The main reason for the ILP performance gain is the significant reduction in the number of memory accesses, mainly by saving copy operations. Because ILP was initially considered as a technique which would benefit from the presence of caches, it is surprising that the relative amount of first-level data cache misses is higher in the ILP case. 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 ALF and processing the different loops close to each other.

One reason for the higher ratio of first-level cache misses for ILP is that there are always a fixed number of write cache misses for writing the data into the final 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. Another reason is that the simplified SAFERK64 encryption algorithm uses two tables (pre-calculated logarithm and exponential tables with values for each possible byte) and a byte vector for the key, 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-words do not follow each other.

As expected, the implementation using the TCP kernel version is faster than the user-level implementations. Protocols in user space do not usually achieve kernel performance, but nevertheless they have several advantages [9]. However, the performance gap could be minimized by optimizing the data path, which is usually the critical path of protocol implementations. ILP packet processing is even faster than decryption and unmarshalling on top of kernel TCP. One reason for better throughput with a kernel TCP is that the kernel TCP code is more optimized and acknowledgment packets do not cross the user/kernel domain as it does in a user-level TCP implementation.

4 Advanced Communication System Concepts

The experiences with the ILP implementation and the obtained results are the basis for proposals about advanced protocol and implementation architectures to allow more efficient communication system implementations.

4.1 Protocol Architecture

ILP implementations would be simpler by using uniform processing unit sizes for the different data manipulations. Moreover, future protocols should be designed following the proposals below. Although there are several new transport protocol approaches with the proposed features [5,10,11], none of them is used in the OSI or Internet world:

- ***Fixed size headers*** should allow fast header prediction and efficient header parsing.
- ***Separate control and data packets*** simplify header prediction and allow processing of control and data packets in different system domains. E.g., data packets could be processed in user space, while control packet processing could be processed within the kernel or even on an intelligent network adapter. This implementation strategy would also avoid to cross the user/kernel boundary for acknowledgments. Further, a clear separation of control and data packets supports parallel implementations.
- Protocols should provide ***demultiplexing support***, e.g. by end system unique connection identifiers and sequence numbers, to allow incoming data to be copied directly into the final application buffers.

4.2 Implementation Environment

The workstation environments used today hide the achievable ILP benefits to a significant degree. Often, data are copied several times while travelling from the application space through the kernel space and finally to the network adapter. Also the operating system overhead and the missing ***integration of communication systems and applications*** are reasons for performance degradation. Using advanced systems with zero-copy network adapters and dedicated low overhead operating system could raise the ILP benefits further. Avoiding system copies requires modifications of the operating system and the network adapter architecture. To avoid unnecessary copy operations incoming data should be stored by the system at the final destination memory. Demultiplexing packets to the application buffer requires protocol support as discussed in Section 4.1. ILP will certainly become more important in systems with more efficient network adapters and improved integration of communication systems into the operating systems [12,13].

Another problem with protocol implementation occurs if the required data manipulations are very costly. If data manipulation processing dominates, and moving the data between memory makes only a small fraction of total processing, ILP results only in small benefits, which may even be invisible. The only reasonable solution is the use of ***dedicated hardware*** for costly data manipulations. Hardware components are used in current workstation technology, e.g. for expensive encryption functions such as DES or for data compression.

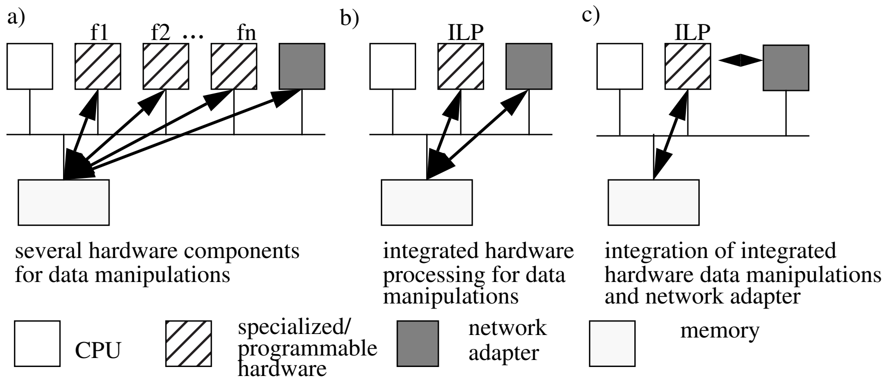

Fig. 1. Hardware support for data manipulations

Especially in the case of several data manipulations implemented in hardware, data must be transferred for each hardware component from the main memory to the hardware component and back to the main memory (Figure 1a). The data transfer over the bus will be the main bottleneck in data manipulation processing in that case. Especially in such a scenario, ILP could become a promising approach by integrating several data manipulations into a single ILP loop and processing it on a single hardware component. In this case, the data are transferred only once from the main memory to the data manipulation hardware, the data manipulation hardware performs the ILP loop, and finally the data is transferred to the destination memory (Figure 1b). Figure 1c shows a further optimization by connecting the network adapter and the data manipulation hardware. This configuration results in a single data transfer over the system bus. Another advantage of a hardware solution is that data or instruction cache misses because of context switches can not occur. The user data as well as other data necessary for data manipulations are always present in the memory of the hardware component. Furthermore, no (connection) state information is required on the specialized hardware processor. The complexity of the hardware unit is fully independent from the number of applications, connections etc.

Programmable hardware devices fit very well into the implementation approach. They allow to configure and to load exactly those combinations of data manipulations, which are required by the workstation applications. A possible implementation would be a stub compiler integrating data manipulations into (un)marshalling routines. Such a stub compiler could produce hardware description code such as VHDL code instead of C procedures generated by conventional stub compilers. The output of the hardware stub compiler must be synthesized with a synthesis tool and downloaded to the programmable hardware processing unit. The data description language for such a hardware stub compiler should be the same as for a software stub compiler, e.g. ASN.1 or XDR language. Dependent on the expected complexity of the data manipulations software or hardware (un)marshalling procedures could be generated by selecting either the hardware or the software stub compiler (Figure 2). The software and the hardware solution can even coexist in a single environment. While all initial and final processing steps are performed in software, the ILP loop is implemented in software and/or hardware.

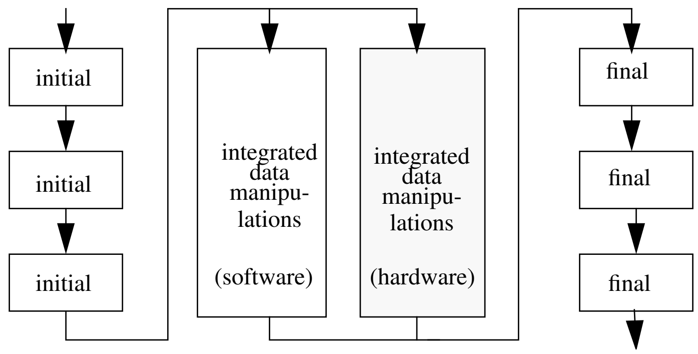

Fig. 2. Alternative integrated HW/SW data manipulations following the 3-stage-approach

The hardware implementation also allows an efficient implementation of the non-ILP/ALF approach, which has a slightly worse performance compared to the pure ILP approach, but which provides the possibility of a dynamically changeable implementation. In the non-ILP/ALF approach a packet is transferred to the hardware processing unit and one data manipulation after the other is processed. Another advantage of the latter approach is that limitations about the type of data manipulations do not exist. Also ordering-constrained functions can be implemented. Because of the lower degree of limitations combined with the expected high performance, this approach seems to be a very promising concept for implementations of costly data manipulations.

5 Conclusions

Although remarkable performance improvements of up to 35% for packet processing times could be achieved with ILP, there remain several limitations and problems. Often, the benefits of ILP by improved packet processing are nearly invisible in the total throughput performance. ILP limits the degree of flexibility and ordering-constrained data manipulations prevent the use of ILP. Further, the protocol architectures used today in communication systems do not support ILP very well. In general, this makes the use of a software ILP implementation on the workstation CPU debatable, especially when data manipulations are very costly and computations are more expensive than memory access. Therefore, a more flexible implementation concept is proposed, which allows to implement integrated data manipulations either in software or in hardware dependent on data manipulation characteristics and application requirements.

References

1. Müller-Schloer, C.; Thurner, E.: RISC, Superskalarität und die Grenzen, Praxis der Informationsverarbeitung und Kommunikation, PIK 3/94, Vol.17, pp. 166-177
2. Clark, D.D.; Tennenhouse, D.L.: Architectural Considerations for a New Generation of Protocols,ACM SIGCOMM 1990, pp. 200-208
3. Abbott, M. B.; Peterson, L.L: Increasing Network Throughput by Integrating Protocol Layers, IEEE/ACM Transactions on Networking, Vol. 1, No. 5, pp. 600-610, October 1993
4. Gunningberg, P.; Partridge, C.; Sirotkin, T.; Victor, B.: Delayed Evaluation of Gigabit Protocols, Proceedings of the Second MultiG Workshop, June 1991
5. Braun, T.: PATROCLOS: A Flexible and High-Performance Transport Subsystem, 4th International IFIP Workshop on Protocols for High-Speed Networks, Vancouver, August 1994, pp. 183-201
6. Feldmeier, D.C.; McAuley, A.J.: Reducing Protocol Ordering Constraints to Improve Performance, 3rd IFIP Workshop on Protocols for High-Speed Networks, Stockholm, 1993
7. Braun, T.; Diot, C.: Protocol Implementation Using Integrated Protocol Processing, ACM SIGCOMM 1995
8. Massey, J.: SAFER K-64: A Byte-Oriented Block-Ciphering Algorithm, Lecture Notes in Computer Science 809, Springer-Verlag, 1993, pp. 1-17
9. 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
10. Strayer, W.T.; Dempsey, B.J.; Weaver, A.C.: XTP: The Xpress Transfer Protocol, Addison Wesley, Reading, Mass.
11. Sabnani, K.; Netravali, A.; Roome, R.: Design and Implementation of a High Speed Transport Protocol, IEEE Transactions on Communications, Vol. 38, No. 11, November 1990, pp. 2010-2024
12. Druschel, P.; Peterson, L.L.: Fbufs: A high-bandwidth cross-domain transfer facility. Proceedings of the 14th ACM Symposium on Operating Systems Principles, December 1993, pp. 189-202
13. 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