The Applicability of Integrated Layer Processing

Bengt Ahlgren, Mats Björkman and Per Gunningberg

Abstract— **In this paper we review previous work on the applicability and performance of Integrated Layer Processing (ILP). ILP has been shown to clearly improve performance when integrating simple functions, but the situation has been less clear for more complex functions and complete systems.**

We discuss complications when applying ILP to protocol stacks, the requirements of ILP on the communication subsystem, caching aspects, the importance of the processor registers, and a model for predicting the performance of data manipulation functions.

We conclude that the main drawback of ILP is its limited applicability to complex data manipulation functions. The performance to expect from an ILP implementation also depends heavily on the protocol architecture and the host system architecture.

I. Introduction

It is well known that the communication performance of workstation class computers is limited by the performance of their primary memory systems [1], [2]. The reason is that software implementation of protocol data manipulation, such as checksum calculation, needs to access every byte of message data in memory. Since the access latency of memory systems using DRAMs is decreasing at a much slower pace than CPU speeds are increasing [3], [4], the bottleneck is getting more and more severe relative to the CPU performance of future computer systems.

Integrated Layer Processing, ILP, is an implementation technique for data manipulation functions in communication protocols presented by Clark and Tennenhouse [5]. The purpose with ILP is to relieve the memory bottleneck by reducing the number of memory accesses for message data and thus increase communication performance. The potential reduction comes from combining the manipulation functions of several protocol layers into a pipeline in a single processing loop. For each iteration of the loop the manipulations are run in succession on a small quantity of data, typically a word or a couple of words. Message data is pipelined via registers between the manipulations and need not be accessed from memory by each function.

But in most cases, ILP does not come for free. ILP adds complexity to the implementation. The cost of implementation can therefore increase. The complexity can also reduce the performance to the point that it completely offsets the performance benefit of ILP.

This work is supported in part by the CEC DG III Esprit LTR project 21926 HIPPARCH.

B. Ahlgren is with the Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden. E-mail: Bengt.Ahlgren@sics.se.

M. Björkman and P. Gunningberg are with the Uppsala University, Dept. of Computer Systems, Box 325, S-751 05 Uppsala, Sweden. E-mail: Mats.Bjorkman@docs.uu.se and Per.Gunningberg@docs.uu.se.

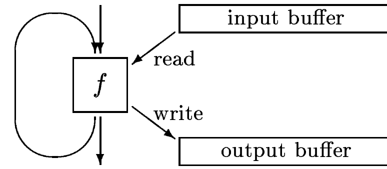

Fig. 1. A data manipulation function "f".

In this paper we review previous work on ILP, our own and other researchers', and analyze the benefits and drawbacks of ILP. The main questions we will try to answer are: When is a performance gain to be expected from applying ILP? When is ILP applicable?

The paper is organized as follows. Section 2 explains what ILP is and the complications that arise when ILP is applied to a protocol stack. Section 3 discusses the importance of having a communication subsystem designed for ILP. Section 4 analyses the effects of processor cache behavior on data manipulation functions. Section 5 looks at the data manipulation functions in more detail and defines a set of parameters describing the functions and also describes a method for measuring the performance of functions with arbitrary characteristics. Section 6 defines a performance model for integrated and sequential implementations of data manipulation functions. The model can be used to predict and compare the performance of the respective implementation given a description of the data manipulation functions and the target computer system. Section 7 finish the paper with a summary of our conclusions on the applicability of Integrated Layer Processing.

II. Integrated Layer Processing

A. *Data manipulation functions—Sequential vs. integrated implementations*

Communication protocol processing can be divided into two parts: protocol control and data manipulation. The protocol control part processes the information in message headers and maintains the state needed by the protocol, for example, connection state. The control processing is usually specified and implemented as a finite state machine. The data manipulation part processes the message data. Examples of data manipulation functions are checksums, encryption and presentation encoding. They all have in common that they process each byte of the message data and possibly produce new data. This is illustrated in Figure 1. The data to be manipulated is stored in buffers. The input and output buffers are often the same buffer in a real implementation. The data manipulation function f is implemented as a loop which reads a block of data from

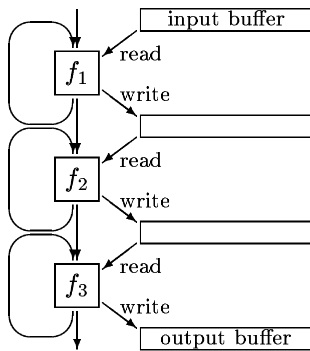

Fig. 2. Sequential composition of data manipulation functions.

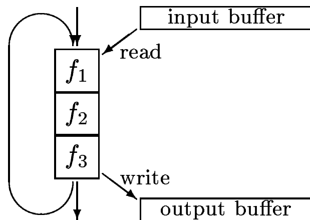

Fig. 3. Integrated composition of data manipulation functions.

the input buffer, computes the function value and writes the result to the output buffer. The minimum size of the block depends on the function.

In a traditional implementation of a protocol stack, the data manipulation functions are executed independently in their respective layer, as illustrated in Figure 2. We call this implementation a *sequential* implementation. Each function follows sequentially and reads and, possibly, writes all data in the buffer. The functions need not be immediately adjacent as in the figure. They can even be located in different address spaces. An example of this is the implementation of the TCP/IP protocol stack with an RPC protocol in a typical Unix system. It has presentation layer encoding in the application and the UDP or TCP checksum in the operating system kernel.

The idea with Integrated Layer Processing [5] is to combine the data manipulation functions of several protocol layers into one processing loop as illustrated in Figure 3. The purpose is to avoid reading and writing the message data in the buffers several times. We call this implementation an *integrated* implementation. It reads a block of data from the input buffer and applies all functions in succession on the data. The assumption is that the transformed data is stored in processor registers between the functions and need not be accessed from memory by each function. After the last manipulation function the resulting block of data is written to the output buffer and the loop continues with the next block until the input buffer is finished. In this particular example with three functions each producing new data, the number of memory accesses for message data is reduced to $\frac{1}{3}$ of the sequential implementation.

ILP has been successfully used to fold together checksum computation with a copy operation by Clark and Tennenhouse [5], Jacobson [6], Partridge and Pink [7], and Dalton *et al.* [8]. The performance increase is typically in the order of 50 %. Abbott and Peterson [9] and Gunningberg *et al.* [10] have shown that ILP gives a significant performance increase also for other simple functions.

The relative performance gain can however be small for complete systems and for complex functions, as shown by Braun and Diot [11] and Gunningberg *et al.* [10].

B. *Complications with applying ILP*

There are a number of problems or complications that arise when ILP is applied to a protocol stack. In some circumstances the complications are true limitations that prohibit a fully integrated implementation. Most complications, however, have the property that they make an integrated implementation of the protocols more complex, but they do not preclude integration *per se*. The complexity may make the integrated implementation too costly to be justified. The complexity may also completely defeat the performance gain that was the purpose with integration, and therefore make integration undesirable.

Clark and Tennenhouse [5] define ILP in the context of Application Layer Framing, ALF. ALF is an architectural principle where the data units of the application (ADUs, application data units) are preserved through the whole communication system, and where the application data units can be independent of each other. It is the responsibility of the application to form data units of suitable size for the lower layers. The independence makes it possible for the application to process received data units out of order. A communication system designed according to the ALF concept is more suitable for applying ILP than traditional communication systems of non-ALF design. We do not generally assume ALF in the following discussion, but when it matters, we point that out.

This section is partly based on the work by Clark and Tennenhouse [5], Gunningberg *et al.* [10], Abbott and Peterson [9], and Braun and Diot [11].

B.1 Different or varying block sizes

The block size of a data manipulation function is the minimum amount of data that the function can process. For example, the Internet checksum has a block size of two bytes and DES encryption has a block size of 8 bytes.

With fixed and the same input as output block sizes it is straightforward to combine the functions. Abbott and Peterson [9] use "word filters" which coerces the block size of all functions to one word. Functions with a larger block size are rewritten to buffer the input until it has collected the number of words needed for processing. The complication with this approach is that additional state must be kept that must be tested in the inner ILP loop. This reduces performance. Braun and Diot [11] as well as the authors of this paper use the least common multiple of the block sizes as the block size of the integrated function. For most functions this approach works well, but the least common multiple could in odd cases be an inconveniently large number.

A data manipulation function can also expand or reduce the amount of data, i.e., the input block size is not the

same as the output block size. The situation gets even more complex when a data manipulation function has a variable block size that depend on the data content, as, for example, is the case with many compression functions. In the general case, a variable block size has to be handled by testing if there is enough input data for a function, and restarting the ILP loop from the beginning if more data is needed. This adds considerable complexity to the integrated implementation, complexity that most likely completely defeats the ILP performance gain.

Fry and Ghosh [12] have made experiments with dynamically combining data manipulation functions at runtime. Their idea is to have a library of data manipulation functions from which the functions can be combined at runtime and executed in an integrated fashion. One benefit with their approach is that the modules in the library are reused by all applications or protocols needing them. The modularity of the system is also in a natural way preserved.

B.2 One protocol's data is another protocol's header

In traditional protocol architectures, protocols commonly add headers to messages going down a protocol stack, and conversely remove the same headers from messages going up. The protocol headers of higher layer protocols are user data to lower layer protocols. The data manipulation functions of higher layer protocols are then applied to a smaller part of the complete message than the functions of lower layer protocols. The protocols at different layers have a different view of what data to manipulate.

This means that only the part of the message covered by all manipulation functions can be processed by an integrated implementation. The remaining parts for lower layers, consisting of higher layer protocol headers, have to be handled in the normal sequential way. This is also the approach proposed by Abbott and Peterson [9]. They introduce an abstract data type called "segregated message" which makes the start of the application data visible to the whole protocol stack.

B.3 Protocol ordering constraints

Protocols in a protocol stack commonly have constraints on the order in which the different protocol functions, including data manipulation functions, are executed. There are three kinds of ordering constraints:

- constraints on ordering between functions in the same protocol,
- constraints on ordering between functions in protocols at different layers, and
- constraints on processing order within a data manipulation function.

The first two kinds are basically the same—the protocol functions just belongs to different layers. As an example, a protocol may require that a checksum is calculated over a received message before deciding if an acknowledgment should be sent for that message.

Examples of the third kind of constraint are all encryption function used in chaining mode, as well as CRCs and message digest functions such as MD-5. They all have to process the bytes of a message in a certain order.

B.3.a Limiting constraints. All ordering constraints form a partial order on the protocol functions. To capture the third kind of constraint the processing of the data manipulation functions are split up block by block in the ordering. In this way, the ordering constraints between the processing of the blocks become visible. Integration adds more ordering constraints between the blocks of the data manipulation functions. If this would add a loop, breaking the partial order, integration is not possible.

A good example which would create a loop in the partial order is a protocol stack with TCP and the proposed IPv4/IPv6 "IP Authentication Header" [13] using the MD-5 message digest algorithm. The TCP checksum needs to be computed and put in place in the TCP header before MD-5 can start any processing of the rest of the message. It is the combination of two ordering constraints that causes the loop and therefore makes integration of these particular data manipulation functions impossible. The first function must be completely computed before the next one starts. A pragmatic way to avoid the problem in this particular example is to put the TCP checksum in a trailer instead, but that requires changing the TCP protocol specification.

Such combinations of ordering constraints is not uncommon in existing protocols. In the above example, the same problem occurs when UDP is used instead of TCP, as well as when using the "IP Encapsulating Security Payload (ESP)" [14] with any chaining encryption algorithm instead of the IP authentication header. Another example is from the Pretty Good Privacy [15] security package. The RSA-signed MD-5 message digest is put in a header which is compressed with Lempel-Ziv and encrypted by IDEA together with the plaintext. The compression and encryption can not be integrated with the MD-5 message digest because the signature must be completed before the compression is started.

The "three stage approach" is a method proposed by Abbott and Peterson [9] to solve the problems with ordering. The method limits the design space for a protocol stack so that it always is possible to add the ordering constraints imposed by integration. The protocol processing is divided in three stages: an initial stage, a data manipulation stage and a final stage. The initial and final stages are executed serially, but the data manipulation stage is executed as one integrated loop. The approach does not allow chained data manipulation functions that run over both headers and data of higher layer protocols.

B.3.b Complicating constraints. There are other kinds of ordering constraints that complicate the integrated implementation. A protocol in the middle of the stack may want to send an acknowledgment. The protocol has to wait with this until the ILP loop is completed. A similar situation is when a checksum verification fails and the message should be discarded. The verification can not be done until the ILP loop is completed. These situations are handled by the "three stage approach", but makes the integrated

implementation more complex compared to the sequential.

B.4 Conflict with layered engineering

Layered engineering is used in most existing protocol architectures. The purpose is to simplify the design and implementation of the communication system by making it modular. ILP is in conflict with the layered engineering principles, because the data manipulation functions from several protocol layers are implemented together in the same software module.

Gunningberg *et al.* [10] propose to use *delayed evaluation* of the data manipulation functions to be able to both keep the layered design and also execute the functions in an integrated loop. The layer where the manipulation function belongs, delays the evaluation of the function until it can be evaluated integrated together with other data manipulation functions.

Abbott and Peterson [9] propose to use an automatic synthesis tool to build the integrated implementation. The benefit is that the layered design and specification can be kept while a tool automatically takes care of the details of the integrated implementation. Braun and Diot [16] use the same approach.

B.5 Multiplexing and segmentation

Multiplexing and segmentation are two protocol functions that make ILP more difficult to apply.

Different data manipulation functions may be used above a multiplexing point in a protocol stack depending on who sends or receives a message. If there is a multiplexing point in the middle of the protocol stack, and there are data manipulation functions both below and above the multiplexing point, the receiving side must take the demultiplexing decision before entering the ILP loop, that is, potentially at the bottom of the protocol stack. Otherwise, the correct ILP loop can not be selected, since there must be one ILP loop for each variant of manipulation functions in the layers above the multiplexing point. Additional ordering constraints may be imposed, if, for instance, the information used to demultiplex need to go through a data manipulation function before it can be used.

The alternative is to terminate the integration at the multiplexing point and have one ILP loop for the layers below, and additional ILP loops for each protocol variant above. This latter approach is easier, but performs less good.

Segmentation of a message at one layer into two or more messages at a layer below also complicates the application of ILP. The data manipulation functions at the layers below the segmentation point do not process the same amount of data as above.

In a communication system designed to the ALF principle, the application data units are preserved by the lower layers. Segmentation is in other words not performed. It is the responsibility of the application to break its data unit into suitable units for the lower layers.

III. ILP and the Communication Subsystem

In this section we will discuss how ILP fits into advanced communication subsystem architectures.

In many systems, the communication subsystem spans multiple address spaces and protection domains. It is the case when one part is located in the application address space and the other part is located in the operating system address space. For example, in Unix systems, the application protocol is often placed in the user address space while the transport protocol and lower layer protocols are often located in the kernel address space.

Many traditional operating system APIs for communication, such as the Unix system call socket interface (`send()`, `recv()`, etc.), copies data between operating system buffers and application program data structures. This interface gives flexibility to the application. It can choose to send or receive data from/to any location in its address space. It can choose how the data is aligned (e.g., word, longword, or page boundary) because the application has complete control of its data structures. Furthermore, the separation of address spaces protects the integrity of the system from malicious users. However, this copying of data from one place in memory to another is costly. It may even offset the gain achieved by an integrated implementation where the aim is to reduce the number of memory accesses. A successful approach is to combine this copying with a data manipulation function, such as checksumming, as is reported on in [5], [6], [7], [8]. An even better alternative is to avoid this copying altogether. There are some reports on how to reduce this copying which is the case in minimal-copy data path architectures [17], [1], [18]. Our own work on minimal-copy architectures is described in more detail in [19], [20]. We will here discuss how ILP fits with a minimal-copy API architecture.

A. *Minimal-copy communication subsystems*

In minimal copy architectures, the communication subsystem "move" data, instead of copying data, between domains by the means of passing pointers to shared data structures, page remapping, application program interfaces that support sharing and the handling of protection mechanisms.

The approach of passing buffer and data structure pointers via the API may seem simple, but it has a number of implications. The data buffers still have to be moved between the two protection domains. Our work is based on the ideas presented by Druschel and Peterson [21]. They have designed a buffer system called "fbufs" which is a mechanism for transferring page-size buffers between protection domains. Fbufs is a combination of virtual page remapping and shared virtual memory. We have made an experimental implementation of a minimal copy subsystem, based on the fbufs ideas, complemented with a new API (extension to SunOS sockets) but without an ILP loop. Measurement results and our SunOS implementation are described in [20].

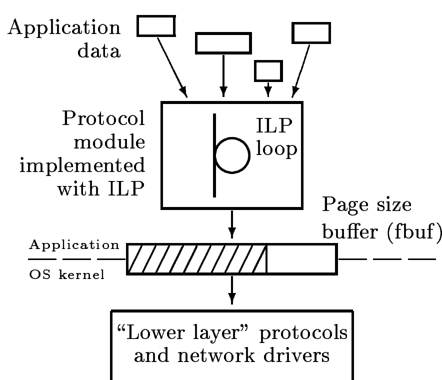

Fig. 4. Outbound data path.

B. *An ILP loop in the minimal copy data path*

A communication subsystem will have two ILP loops, one loop for outbound data (transmit) and one for inbound data (receive). An ILP loop either produces new data or leaves the data unchanged depending on the functions integrated in the loop. We assume that the loop always produces new data, which means that a new buffer must be allocated for the newly produced data. When all data in the incoming buffer is consumed, the buffer is discarded.

In the following discussion we will assume that the ILP loops will be located in the user address space. Then the ILP loops will be involved in two data movement interfaces: 1) the application-to-ILP-loop interface and 2) the operating system API-to-ILP-loop interface. In a minimal-copy architecture the ILP loop should be combined with one of these data movement interfaces, typically with the reading of data structures from the application.

The ILP loops could also be located in the system address space. The difference is then that application data will cross the operating system boundary before it reaches the ILP loop in the inbound direction and afterwards for the other direction. A discussion on buffer implications for this case will be similar to the following discussions on ILP loops in the user address space.

We will now describe the data path through the communication subsystem architecture beginning with the outbound side (see Figure 4).

B.1 Outbound data path

The application-to-ILP-loop interface attaches the application to the top of the ILP loop. The loop reads data directly from the application data structures. The interaction between the application and the ILP loop is tightly controlled by the application, since only the application has knowledge of the application data structures. The input data structures are in the general case scattered in the virtual address space of the application. The address of a data structure is determined by the compiler, linker, stack, heap, etcetera.

The communication subsystem gives the ILP loop an empty sequential buffer, in which the ILP loop produces the outbound data. The buffers should be designed for crossing the operating system boundary (e.g according to fbuf ideas) and aligned for efficient ILP loop writing. The location of the buffer should also be controlled in order to avoid cache conflicts with input data and ILP state. The buffers from the protocol module are transfered over to the lower layer protocols through the operating system boundary. The driver finally makes the network adapter send the data and then deallocates the buffer.

B.2 Inbound data path

The inbound data path starts at the network adapter hardware. The adapter identifies to which data path an incoming packet belongs, and selects a buffer allocated for that path by the network driver. The input buffer requirements are very similar to the requirements of the outbound data path. The buffers should also here be designed for crossing the operating system boundary and aligned for efficient ILP loop reading.

After the adapter has placed the data in buffers, there is no way of rearranging data in a buffer without copying. The ILP loop will therefore receive data in buffers exactly as the adapter places the data in memory. As a consequence, any data alignment requirement from the ILP loop must be pushed down to the network adapter hardware. When receiving data, the adapter must place the data part of the packet at the required alignment. This effect can in some circumstances be achieved by off-setting the start of the buffers so that the data part begins where desired.

IV. Data Buffers and the Processor Cache

In this section we will discuss cache effects on ILP performance, with the focus on data caching.

Cache memory behavior is crucial for the performance of modern processors. In order to make predictions on the performance of the memory subsystem, we must know the behavior of the cache system and the probabilities of cache conflicts. In order to improve performance, we must control how caches are used.

A. *Cache architectures*

There are several design parameters of a cache that affect the possibility for conflicts besides its size. A cache is organized in a number of *lines*. Each line can hold a number of sequential bytes from primary memory. A common line size is 32 bytes.

The cache can be *direct mapped*, which means that the low bits of the data item's address directly indexes the cache. The alternative to direct mapping is varying levels of *associativity.* In an associative cache, a *set* of cache lines can be kept at each cache index. The set size determines the level of associativity, e.g. a 4-way set associative cache has a set size of 4. These caches do not suffer from conflicts in the same degree as direct mapped caches, but the drawback is that associative caches are slower and more expensive.

A processor can have separate (or *split*) caches for instructions and data. The alternative is a *unified* cache holding both instructions and data. The computer system can also have several *levels* of caches, where the first level is

the smallest, fastest and closest to the CPU. A typical high performance processor of today has split first level caches and a fairly large (256 KB to 1 MB) unified second level cache. The first level caches can have some associativity, but the second level cache is usually direct mapped.

B. Conflicts

Conflicts in the data cache occur when a cache line is to be fetched, and the indexed position (or, in the case of an associative cache, all positions in the indexed set) is already occupied. The (one) old cache line is then evicted from the cache, and the new line can be placed in the cache.

Depending on whether the conflicting cache line comes from the same application or not, we divide conflicts into two types, *internal* conflicts and *external* conflicts.

For external conflicts, the communication subsystem can typically not control other applications' use of the cache. However, since the aim of ILP is to lower the number of memory accesses, the number of potential cache misses will also be lowered.

Internal data cache conflicts in the communication subsystem can have a major impact on performance [22]. This can happen e.g. in a direct mapped cache if the input and output data buffers overlap in the cache, or if either of the buffers overlap with tables or constants used by the data manipulating functions. For systems with set-associative caches, the risk of overlapping data actually resulting in cache conflicts is of course lower.

We have measured the performance effects of internal conflicts between input and output data buffers for the Abbott and Peterson BSWAP/PES/CKSUM ILP loop [9]. This loop is an ILP loop in which a byte swap, a pseudo encryption and a checksum calculation is combined.

Figure 5 is a code listing of the original integrated implementation of the loop. We have made one change to this implementation. We use a block size of two words which saves the "if" statement and gives better performance. Compared to the original one word implementation, we received between 25% (for cached message data) and 35% (for non-cached message data) improvement in performance when using a block size of two words. We discuss the block size further in Section VI-D.

For this experiment we used a SPARCstation 2. It has a 64 KByte unified data and instruction cache with 32 byte lines. The cache is virtually indexed and direct mapped. The design is a few years old and therefore is not completely representative for state-of-the-art systems. It has, however, some characteristics in common with today's designs. Most, if not all, of today's systems have unified and direct mapped second level caches.

The results as presented in figure 6 show that the cost of internal conflicts can be substantial. For a full presentation of our cache conflict results, we refer to [22].

Our conclusion is that, in order to eliminate (or lower the risk of) internal cache conflicts, the placement of data buffers in memory should be controlled by the communication subsystem.

```
for (; i > 0; i--) {
    /* READ A WORD OF INPUT */
    DataWord = *inputBuffer++;

    /* BSWAP */
    DataWord = ((DataWord & 0x00FF00FF) << 8) |
               ((DataWord & 0xFF00FF00) >> 8);

    /* PES */
    if (!awaitingSecondWord) { /* DataWord is first */
        firstWord = DataWord;  /* word of a pair    */
        awaitingSecondWord = TRUE;
    }

    else {  /* DataWord is second word of a pair */
        secondWord = DataWord;
        awaitingSecondWord = FALSE;
        DataWord = (firstWord & 0xF0F0F0F0) |
                   (secondWord & 0x0F0F0F0F);
        /* CKSUM */
        sum += (DataWord & 0xFFFF) + (DataWord >> 16);

        /* WRITE A WORD OF OUTPUT */
        *outputBuffer++ = DataWord;

        DataWord = (firstWord & 0x0F0F0F0F) |
                   (secondWord & 0xF0F0F0F0);
        /* CKSUM */
        sum += (DataWord & 0xFFFF) + (DataWord >> 16);

        /* WRITE A WORD OF OUTPUT */
        *outputBuffer++ = DataWord;
    }
}
```

Fig. 5. Abbott & Peterson's integrated BSWAP/PES/CKSUM loop.

C. Effects of cache hit rate

Ideally in an ILP implementation, data blocks are accessed from the memory system only once, subsequent references being made to data in registers. Also, ideally, each cache line is fetched from main memory only once, subsequent memory accesses to data in the same cache line being serviced from the cache.

Braun and Diot [11] observed that employing ILP lowers the cache hit rate (ratio). This is because the total number of memory accesses is lowered since an ILP implementation often can keep data in registers between accesses. The absolute number of cache *misses* however remains the same, or is slightly lowered because of a better locality of reference in the ILP case. Thus, this lowering of cache hit rate does not imply a heavier load on the memory subsystem.

Figure 6 also shows the performance effects of cache hit rates. The measurements with cold caches show the performance for a cache hit rate of 0 (every cache line has to be fetched from main memory), and the measurements with warm caches show the performance for a cache hit rate of 1 (every cache line is already in cache).

It is important to note that the warm cache case is an ideal case where the data buffer already resides in the cache when the communication subsystems gets to work on it. This can happen e.g. if (for outgoing messages) the sending application has accessed data just before handing the buffer to the communication subsystem, or (for incoming messages) if the host architecture supports DMA transfers

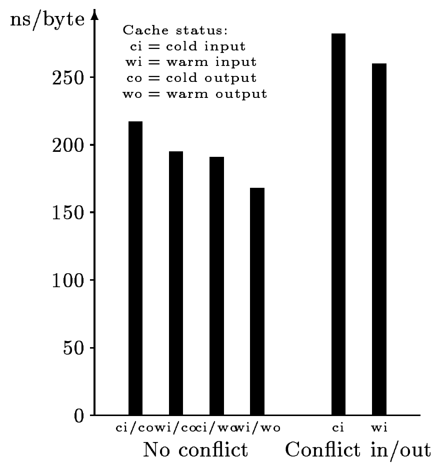

Fig. 6. Abbott & Peterson's BSWAP/PES/CKSUM loop: No cache conflict and conflict between input and output buffers.

directly to cache from the network interface. In the general case, however, message data is typically *not* found in cache by the communication subsystem. Hence, the cold cache case is expected to be the common case.

Measurements reported on in the next section show that the impact of varying cache hit rates on performance is small compared to other factors (e.g. the availability of registers). The cache miss cost is furthermore a (more or less) constant cost for a given application on a given system. In the measurements reported on in the next section, this constant cost can be clearly seen from the figures 7 and 8.

D. Discussion

For caching and ILP, we conclude:

- Warm or cold cache is not a very big issue. As will be shown in the next section, the cost of a cold cache is typically not a dominating cost, and it is typically more or less constant for a given implementation.
- The cost of local conflicts is a big issue. As measurements presented in this section have shown, local conflicts can have a major impact on performance.
- Local conflicts can however be controlled. By controlling the location of buffers and other data in the cache, local conflicts can be avoided.

V. Characterizing Data Manipulation Functions

In this section, we characterize data manipulation functions by its major performance-affecting parameters, and use an experimental method to predict ILP performance.

The key question when considering the employment of ILP is: When does actually an integrated implementation result in fewer memory accesses compared to a sequential implementation? Part of the answer is that there is more to consider than the memory accesses to the message data. The CPU registers clearly play an important role. If the result of the integration is that one of the manipulation functions can fit less of its frequently needed information in registers, there will be extra memory accesses for this information. These accesses have to be weighted against the saved memory accesses to message data.

A. Data Manipulation Functions

Since the purpose with an integrated implementation is to save memory accesses, we must capture the characteristics that define the memory behavior of the function.

The following paragraphs describe the parameters we have identified as relevant.

- **Input data block:** The size of the input block is fundamental to a data manipulation function. The whole block must be present in order to do the processing. The function processes one block of data in each iteration. For example, the TCP/IP checksum has an input block size of 2 bytes, the DES encryption has 8 bytes and the MD-5 message digest has 16 bytes.
- **Output data block:** Depending on the particular function, the size of the output data block may be the same, less than or greater than the size of the input data block. A data compression function is an example of a function that produces less data than input. Some functions, like checksums, do not produce output data at all.
- **State:** Many data manipulation functions keep a state between iterations. This is the case for the TCP/IP checksum and the DES encryption algorithm in CBC mode. The checksum algorithm adds a 16 bit data block to the accumulated sum of previous blocks, i.e., the sum is the state. The state need to be stored somewhere—preferable in registers, since it is accessed for each iteration. The size of the state is therefore important for the memory behavior of the function. The final state is usually the result of functions that do not produce output data.
- **Constants:** Data manipulation functions can use constant values in its computation. A constant is explicit in the code and is accessed at least once per loop iteration. A constant is therefore a candidate for register allocation by the compiler. As such, constants compete for register space with the function state. A difference is that constants in memory need only be read, while the state is updated and must be both read and written. A constant can however be used more than once, so the compiler must analyze the number of references to be able to make a good register allocation.
- **Table look-ups:** Tables are similar to constants in that they contain constant values. The difference is that only a fraction of the entries in a table are accessed for each iteration. A table is also usually too large for the processor registers and must therefore be accessed from memory. The interesting parameters are the number of table lookups per iteration and the size of the table.
- **Temporaries:** A data manipulation function will use a number of temporary variables for intermediate results. The temporaries need not be explicit in the original source code. The compiler usually puts temporary variables in registers.

- **Instructions:** We distinguish between three logical categories of machine level instructions: *computation*, *memory* and *loop overhead*. The computation instructions are arithmetic-logic and control instructions used to compute the state or the output data of the data manipulation function. The memory instructions transfer data between the CPU registers and the memory or the cache. The loop overhead instructions consist of instructions for incrementing the buffer pointers, decrementing and testing the loop variable and a conditional branch back to the beginning of the loop.

B. Experimental Method

We use an experimental method to investigate how data manipulation functions with varying demands for CPU registers respond to integration.

B.1 Generating data manipulation functions

We have implemented a generator program that takes a set of parameters as input. The generator program produces 'synthetic' code which behaves according to the parameters, but does not do anything else useful. This method gives us complete control over the selection of parameter values and thus over the behavior of the manipulation functions. We actually implemented two generator programs, one producing an integrated implementation and the other producing a sequential implementation from the same set of parameters.

We have also implemented a measurement program whose core is a loop, or a set of loops, in which different code easily can be inserted, in particular the parameterized synthetic data manipulation code generated by the generator program mentioned above. In the integrated case, there is a single loop. In the sequential case, there is a set of loops, one loop per function. The integrated loop, or each sequential loop, runs once over an input buffer and possibly produces data in an output buffer.

The measurement program gives us complete control over how the input and output buffers are allocated and located in the cache. In the measurements, all buffers, the stack and global variables are allocated in such a way that they do not conflict in the cache, i.e., so they do not compete for the same cache location.

The program also controls the cache temperature of the buffers. In the measurements below, 'warm cache' means that all buffers are warm, and 'cold cache' means that all buffers are cold and, in the sequential case, that the buffers also are flushed from the cache between each loop.

B.2 Measuring data manipulation functions

We experiment with four cases: an ILP loop and sequential loops with warm and cold caches respectively. In a real implementation the cases with warm cache correspond to a situation where input data is already touched, for example by an application, and that the output buffer already is in cache, for example it could be a buffer from a previous run of the loop which is reused. Furthermore, in the sequential case, the next manipulation function will use this warm buffer as its input buffer. This is an ideal, best case situation, where none of the output buffers are evicted before it is accessed by the next function. It may only be achievable when manipulation loops are run directly after each other.

In the cases with cold cache, input data to the manipulation loop has to be fetched from primary memory. In the sequential case all output buffers are also evicted from the cache before the next manipulation function is run. This could be the situation in a real implementation when the data manipulation functions are separated in the protocol stack. In most implementations we expect that some cache blocks are evicted and some stay in cache. The performance will then be somewhere in between the two extremes. By comparing these cases, we can estimate the effects of caching on performance.

We use the processing time per byte (in nanoseconds) as the performance measure of a data manipulation function. This measure depends on the number of instructions executed and the average number of clock cycles needed per instruction. The number of cycles depends not only on the instruction, but also on parallelism and data dependencies in the instruction pipeline, and how many cycles the CPU has to wait when referenced data is in cache or in main memory.

We discovered, not surprisingly, that the measurement results are very dependent on the particular compiler used. On both our platforms, SunOS and HP-UX, we used the stock 'cc' compiler. We carefully checked the code produced by the compilers in order to make sure that it was what we expected.

C. Factors Affecting ILP Performance

We have studied three factors that affect the relative performance of an ILP implementation compared to a sequential implementation. The first factor is the *number of computation instructions* per data manipulation function. The results are the expected and show that the number of computation instructions per function does not affect the absolute performance difference.

The other two factors are *function state size* and the *number of functions*. Both factors affect the number of CPU registers that is used to hold the state needed in the loops. The results show that the register usage has a large impact on performance. The following subsection reports on the impact of function state size on performance. For a full presentation of the results for the other factors we refer to [23].

D. State

The state of a data manipulation function is the information in the calculation that needs to be retained between each iteration. For example, the state of a checksum function is the partially calculated checksum.

In the presented experiment we vary the size of the state to see how it affects throughput while keeping other parameters constant. The values of the constant parameters are: 3 functions, 10 arithmetic instructions per function and a 1 word block size.

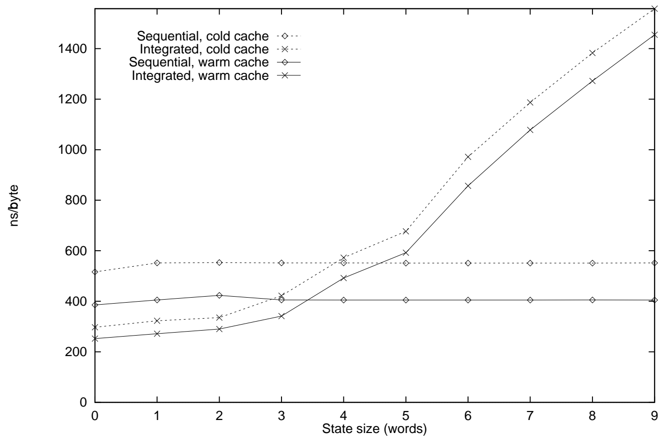

Fig. 7. The processing cost of increasing state size (SPARCstation 2).

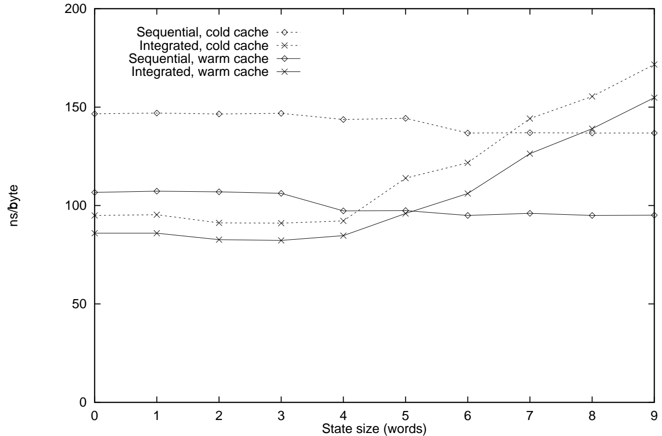

Fig. 8. The processing cost of increasing state size (HP 9000-735/99).

The expected behavior is that when the state size is increased over some threshold defined by the number of available CPU registers, parts of the state must be stored in memory. Thus, the cost of having additional state will increase, i.e., the throughput will decrease. It is worth noting that the state stored in memory will be allocated on the stack and is most likely to be cached since it is accessed at least once for each block. In our experiments we know that the state will be cached, since we control the placement of code, data and stack to avoid cache conflicts.

The integrated implementation with 3 functions has an aggregated state which is 3 times that of the sequential implementation. The register threshold will therefore be reached for an aggregate state size 1/3 of the threshold state size for the sequential implementation. For each word the state of each of the functions is increased over this threshold, the integrated implementation will need to load and store 3 extra words from/to memory, 1 per function.

When state sizes are further increased, eventually also the states of the sequential functions will not fit in registers. After this point, the additional cost for further increased state sizes will be the same for ILP as for the sequential implementation (1 extra load and store for each additional word of state). This is however not shown in the measurements, because when this happens, ILP already performs so much worse than a sequential implementation that ILP is not a viable implementation alternative.

As an illustrating example, assume a system with 9 available registers and 3 functions to integrate. If each function has its state size increased from 9 words to 10 words, this adds a total of 3 loads and 3 stores for each data block to a sequential implementation as well as to an ILP implementation. However, while the sequential implementation can keep the rest of the state for each function in registers, ILP already before the increase has to perform 19 loads and 19 stores on the aggregated state (27 words) for each data block.

The experimental results presented in Figures 7 and 8 clearly show the expected behavior. The unit on the X-axis is the size in words of the state for each of the three functions.

For the SPARCstation 2 (Figure 7), the cost in nanoseconds is almost constant for the sequential implementations. The cost for the integrated implementations is almost constant for state ≤ 2 words. When the state reaches 3 words, the number of available registers are not enough to hold the aggregated state. There is actually room for a larger state in the CPU registers, but the compiler does not do a very good job with register allocation. The better than expected performance for state $= 5$ is due to the compiler 'accidentally' doing a better job allocating the registers. The non-optimal register allocation does however not affect the general result, it only shifts the threshold point.

The HP 9000-735/99 (Figure 8) shows a similar behavior. The CPU has more available registers and the compiler allocates registers better, so the threshold for the integrated implementations is higher than for the SPARCstation. The reason for the slight decrease in cost for larger state size is that the compiler suddenly thinks it should slightly unroll the loop.

We can also note that the effect of warm versus cold cache is a constant factor, smaller for ILP than it is for the sequential implementations.

The first conclusion from this experiment is that the performance of ILP starts to decrease rapidly when the aggregate state size does not fit in registers. The second conclusion is that the cache temperature is a constant factor.

E. Discussion

We have shown that the aggregated state size of the data manipulation functions is a crucial factor influencing the performance gain, or loss, for an integrated implementation compared to a sequential. When the aggregated state does not fit in processor registers, the integrated implementation quickly becomes much slower than the sequential implementation as the state size is increased.

The results presented in this section can be generalized to other systems with different characteristics. The faster the CPU is relative to the speed of the memory system, the higher will the relative cost of loading and storing data

be, thus accentuating the desire to avoid loads and stores. The number of CPU registers available will move the state threshold at which ILP starts to behave worse than a sequential implementation. The more available registers, the larger the state threshold.

The cache architecture, i.e., the size, associativity and whether unified or not, will mainly affect the probability of cache misses. In the presented experiments we have controlled the direct-mapped caches of our measurement systems to avoid conflicts. The results we present are thus similar to results from systems with associative caches (where conflicts do not occur). Not controlling direct-mapped caches will result in probabilities of having cache conflicts. Cache conflicts would further accentuate the difference between situations where state fits in registers and situations where it has to be loaded and stored from memory, since the cost of accessing conflicting locations will increase.

VI. Modeling the Performance of Data Manipulation Functions

In this section we present a performance model that predicts the performance of integrated and sequential implementations of data manipulation functions. The model has been presented in more detail in a previous paper [24].

The previous section showed the importance the CPU registers have for the resulting performance of an integrated implementation. In the model we are therefore particularly interested in the memory behavior and the role of the CPU registers. The model calculates the number of clock cycles per message byte given a set of architecture parameters and a set of function parameters describing the computer system and the data manipulation functions, respectively.

The important contribution of our performance model is that it does not only consider memory accesses for message data, but also memory accesses for function state that does not fit in processor registers.

Abbott and Peterson [9] have also presented a performance model. The main difference compared to our model is that their model does not include memory accesses for function state. It can therefore not be used to study the situation we are interested in when the aggregated function state does not fit in the processor registers. As a consequence, in their model the integrated implementation always performs better than the corresponding sequential implementation.

The motivation behind this work is to fully understand how different parameters affect the performance of the two implementation techniques. This knowledge can be useful for tools ("ILP compilers" [16]) that automatically generate integrated implementations.

A. Model parameters

Table I lists the input parameters of the model. The parameters are of two kinds. The *architecture* parameters specify the characteristics of the target computer system. The *function* parameters specify the characteristics of the desired data manipulation functions.

TABLE I
Model parameters.

Architecture parameters	
R	number of available CPU registers (word size)
A_C	access time of first level cache (clocks/word)
A_M	access time of memory (clocks/word)
H	cache hit rate
CPI	cycles per instruction
L	loop overhead (instructions/loop)
C	CPU clock frequency
W	word size (bytes)
Function parameters	
n	number of functions
s_i	state size for function i (words)
a_i	computation instr. for function i per input block
b_i^{in}	input block size for function i (bytes)
b_i^{out}	output block size for function i (0 if no output)

We use the same set of function parameters in the model as we used to generate the synthetic functions in the previous section. These parameters are a subset of the parameters described above in Section V-A.

The number of CPU registers (R) denotes the number of registers that the compiler has available for the function state. This parameter is thus compiler dependent in addition to being CPU dependent. We assume that this number is invariant when the other parameters of the model are varied. The memory access time (A_M) is the average access time for reading or writing memory sequentially through the cache(s) of the system when the cache initially is cold. The effects of cache line fills and cache line write-backs are included in this average. We assume that the read and the write access times are the same.

The CPI parameter models the superscalarity of the system. A superscalar processor can execute more than one instruction in a single clock cycle. Non-superscalar processors have $CPI \geq 1$. Superscalar processors can have $CPI < 1$, but the value is usually very dependent on the mix of instructions.

The loop overhead (L) depends on the instruction set of the CPU, but can also depend on the compiler. The loop overhead consist of incrementing two buffer pointers (or one pointer for checksum-like functions without output buffers), decrementing and testing the loop variable, and a conditional branch back to the beginning of the loop.

B. Model formulas

The purpose of the performance model is to accurately model the register, cache and memory behavior of the integrated and sequential implementations of a set of data manipulation functions. The number of memory accesses are therefore central in the model. There are two kinds: accesses for message data, denoted DA (*data accesses*), and accesses for function state, denoted SA (*state accesses*). Both of these are expressed as the number of word accesses (load or store) per byte of message data.

The space in this paper does not allow presenting all details of the model. We refer the interested reader to [24]. In particular, we omit the straightforward DA formulas

and instead concentrate on the more interesting formulas for state accesses.

The number of state accesses for the integrated implementation is:

$$SA^{\mathrm{ILP}} = \frac{2}{b^{\mathrm{ILP}}} \times \begin{cases} S^{\mathrm{ILP}} \le R & : \quad 0 \\ S^{\mathrm{ILP}} > R & : \quad S^{\mathrm{ILP}} - R \end{cases} \qquad (1)$$

The important factor in this equation is the parameter R, the number of CPU registers available for function state, and its relationship to S^{ILP}, which is the aggregated state of all functions. *If the whole state fits in the registers, there are no memory accesses. Otherwise, part of the state has to be stored in memory.* This is the key behavior of the model that is expressed by Equation 1. We assume that the state in memory has to be loaded from and stored back to memory once per iteration of the ILP loop, thus the "2" in the numerator of the fraction in Equation 1. b^{ILP} in the denominator is the block size common for all functions in the integrated implementation. It is the least common multiple of the block sizes of all manipulation functions.

The SA formula for the sequential implementation is a sum over the functions:

$$SA^{\mathrm{SEQ}} = \sum_{i=1}^{n} \left(\frac{2}{b_i^{\mathrm{in}}} \times \begin{cases} s_i \le R & : \quad 0 \\ s_i > R & : \quad s_i - R \end{cases} \right) \qquad (2)$$

The expression inside the sum is essentially the same as for the integrated case, but the state and block sizes are for the respective original functions instead of for the integrated function.

The message data and state accesses can either be satisfied by the processor cache or the real memory. We assume that the state accesses will always be present in the first level cache of the processor. This is a reasonable assumption, since the state is accessed on every iteration of the loop. The message data is on the other hand not always present in the cache. The cache hit rate parameter (H) therefore affects the accesses to message data, but not accesses to function state.

The cache hit rate parameter was not present in the original model presented in [24]. The introduction of this parameter makes it possible to merge the two formulas from [24] for the cached and memory cases, respectively. The merged formula for the number of clock cycles per byte of message data is:

$$\begin{aligned} \text{clocks} \quad = \quad & CPI \times \left(\sum_{i=1}^{n} \frac{a_i}{b_i^{\mathrm{in}}} + LO \right) + \\ & \Big(H \times A_C + (1-H) \times A_M \Big) \times DA + \\ & A_C \times SA \end{aligned} \qquad (3)$$

The formula is used for both the sequential and integrated cases with the respective substitution of the sequential and integrated variants of the DA and SA formulas.

The second term is the time to access the message data. Depending on the cache hit rate (H), the access time ranges from A_C, the cache access time, and A_M, the memory access time. The third term is the access time of the function state. As discussed above, it is always accessed from the cache.

The first term of Equation 3 is the number of clock cycles for the non-memory instructions. The sum is the number of computation instructions per byte for all functions. LO is the number of instructions of loop overhead per byte of message data. It is computed from the loop overhead parameter, L.

The sum of the computation instructions and the loop overhead is then multiplied by CPI, the cycles per instruction parameter, to get the time to execute all instructions. The total formula then represents the number of clock cycles to execute the instructions and access the memory.

Finally, if the time in seconds is desired instead of clock cycles, it can be calculated by dividing the number of clock cycles by the CPU clock frequency.

C. *Comparison with measured results*

In this section we compare the modeled performance first to the BSWAP/PES/CKSUM combination of functions and then to the measurements of the synthetic data manipulation functions from Section 5.

C.1 The Abbott & Peterson loop

The first comparison is both with a real implementation of the BSWAP/PES/CKSUM function combination and with a synthetic implementation with the same characteristics. The code listing of the original integrated implementation was presented earlier in Figure 5. It consists of a byte-swap, a "pseudo-encryption" and the Internet checksum. We have also here modified the implementation to use a two-word block size.

Table II presents the measured performance of the real and synthetic implementations compared with the modeled performance for three computer systems. The modeled performance values compare well to the measured real implementation for all three systems. The largest deviation is about 10 % (for the HP, sequential with warm cache).

TABLE II
COMPARISON OF THE MEASURED PERFORMANCE OF THE REAL AND SYNTHETIC IMPLEMENTATIONS WITH THE MODELED PERFORMANCE USING THE BSWAP/PES/CKSUM LOOP.

System	impl. type	Performance, ns/Byte: cold cache Seq.	cold cache ILP	warm cache Seq.	warm cache ILP
HP 9000-735/99	real	95	49	54	37
	synthetic	95	49	54	37
	modeled	*87*	*52*	*49*	*37*
SPARCstation 2	real	344	174	230	125
	synthetic	351	186	236	137
	modeled	*372*	*178*	*247*	*128*
SPARCstation model 20/71	real	78	39	54	32
	synthetic	82	39	57	33
	modeled	*81*	*40*	*58*	*31*

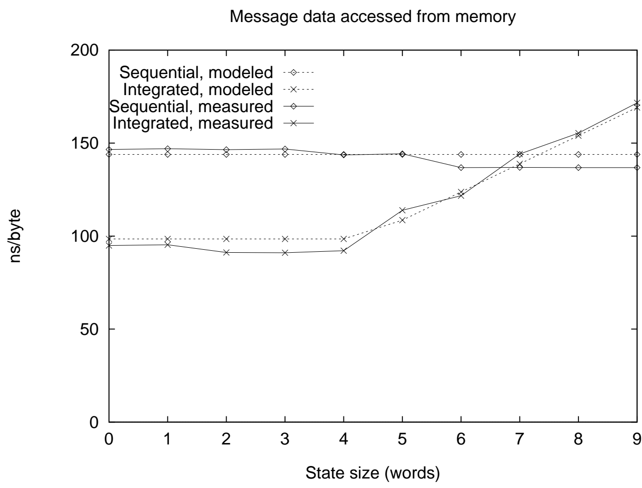

Fig. 9. Comparison of measured performance with modeled performance for the HP 9000-735/99.

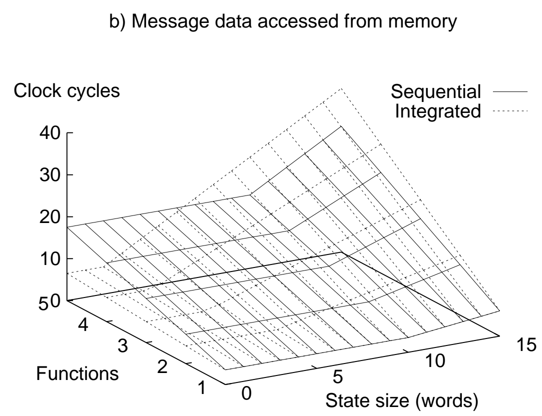

Fig. 10. Modeled performance for different state sizes (per function) and number of integrated functions.

C.2 Synthetic loops

The second comparison is with measurements of synthetic data manipulation functions on the HP 9000-735/99. We have varied the state size and the number of functions. The other function parameters have been kept constant: 3 functions, 4 byte input and output block sizes and 10 computation instructions per input block.

As can be seen in Figure 9, the measured and modeled performance correspond very well. We can see that the model captures the register exhaustion just as intended for the integrated implementation when the state size grows above 4 words. For a state size of 5 words, the aggregated state is $3 \times 5 = 15$ words, which does not fit in the available 13 registers.

D. Model analysis

How do different factors affect the performance of an integrated implementation compared to a sequential? In this section we will discuss the role of the parameters in the model and how they affect the performance of the two implementation techniques.

D.1 Function parameters

D.1.a State size and number of functions. The function state sizes and the number of functions defines the aggregated state size for the integrated implementation. When the aggregated state size exceeds the number of available registers, the excess state will typically be allocated on the stack. This results in additional memory referencing instructions.

If the aggregated state fits in registers, the integrated implementation will always be faster than the sequential. In the model the number of state accesses will then be zero, and both the number of data accesses and the loop overhead will be higher for the sequential implementation.

This is the general performance behavior that the model expresses and which is illustrated by the examples in Figure 10. The respective surface shows the performance in clock cycles per byte of message data for different number of functions and state sizes. The data shown is dominated by the memory and cache performance. There are only three instructions of computation per function and one instruction of loop overhead. The memory access time used to produce the graph is 5 CPU clock cycles per word.

In the left part of the graph, where the state size is small, the modeled performance of the integrated implementation is better than the sequential implementation. There are knees on all curves at the point where the aggregated state size completely fills the available registers (10 in this example). To the right of the knees, there are additional memory references for the part of the function state that overflows the registers. These references are all assumed to be satisfied by the cache. The intersection between the integrated and the sequential surface are the points where these additional state accesses for the integrated implementation cost the same as the additional message accesses and loop overhead in the sequential implementation.

D.1.b Block size. The block size of the data manipulation functions may not seem interesting at first from a performance point of view. Previous research have focused on the practical aspect on how to integrate functions with different block sizes.

There are also other considerations for the block size in the case when the function state is too large for the processor registers. When the block size is increased, the additional loads and stores of the function state is amortized over the larger block size. This effect is seen in Figure 11. The slope above the knees is decreasing rapidly when the block size increases. For the case in the figure, the integrated implementation is faster than the sequential when the block size reaches 32 bytes, regardless of state size.

In this example, the input and output block sizes are the same. The number of available registers are 10 for a block size of 4 bytes. For all larger block sizes, the number of available registers is decreased with the number of additional registers needed for the larger input block. The other model parameters used to generate this graph are: 3 instructions per 4 bytes of message data, 3 functions, 1 cycle cache access time, 1 CPI and 1 instruction of loop

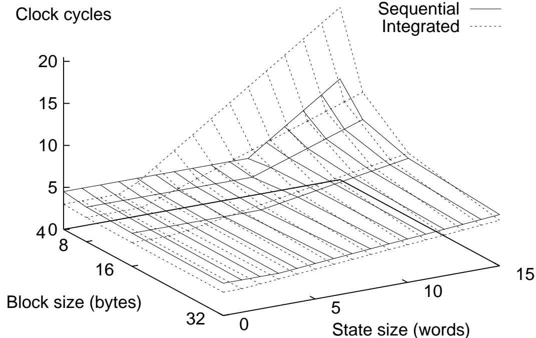

Fig. 11. Modeled performance for different state sizes (per function) and block sizes.

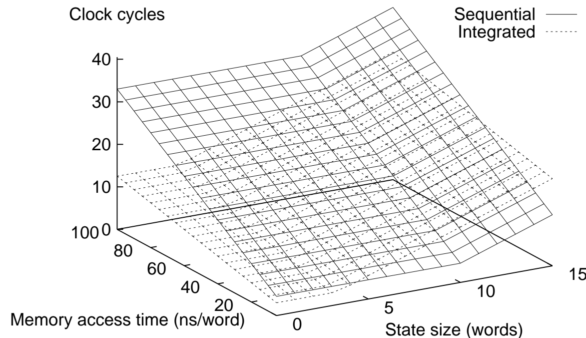

Fig. 12. Modeled performance for different state sizes (per function) and memory access time.

overhead. The figure shows the case when message data is in the cache. When the message data is not present in the cache, the absolute difference between the surfaces becomes larger with a constant, but the shapes of the surfaces are the same.

D.2 Architecture parameters

D.2.a Number of registers. The number of processor registers available for function state defines the threshold point when the excess function state is allocated on the stack. Processors with more available registers increase the applicability of the ILP technique to more functions and to functions with larger state.

As an example, the MD-5 message digest function has a state (the chaining variables) that needs 4 registers. MD-5 has, however, a 64 byte block size, which potentially uses another 16 registers, and 64 unique constants.

Another example is the Safer family of encryption algorithms. It has an 8 byte block size, but processes the message data one byte at a time and therefore needs 8 registers to hold the input block. The whole expanded encryption key is used for each input block. The expanded key is 104 bytes for the minimum recommended 6 "rounds" of the algorithm, which potentially use 26 registers. Actually, because of the byte operations, one register per byte is desirable.

Both of these functions are examples where the CPU registers can not hold everything that is needed in the inner processing loop. If these functions are integrated with another function having state, the registers will overflow even more.

D.2.b Memory access time and cache hit rate. The memory access time and the cache hit rate are very important for the performance difference between the integrated and sequential implementations. The slower the memory and the lower the cache hit rate, the more is gained with integration.

The actual performance experienced in a real system depends heavily on the cache hit rate for accesses to message data. Sequential implementations are more sensitive to varying cache hit rates than integrated implementations, since sequential implementations access data several times.

In extreme cases with low cache hit rates (for message data) and slow memory, the integrated implementation may always be faster than the sequential implementation, also in the cases when the function state overflows the registers. This is illustrated in Figure 12.

D.2.c Cache access time. The cache access time determines the slope of the performance curve for increasing state sizes above the register threshold point. A reasonable assumption is that the cache access time scales with the CPU clock frequency of future processors. This means that the cached performance scales proportionally with higher clock frequency for both the integrated and sequential implementations.

D.2.d CPU clock frequency. When the clock frequency of the processor increases, the memory speed get relatively slower. As we have previously mentioned, one of the main motivations behind ILP is the fact that processor speeds are increasing at a much higher rate than memory latency is decreasing. This will make ILP more and more applicable.

A higher clock speed will have the same effect on the relative performance of an integrated and a sequential implementation as a higher memory access time has.

VII. Conclusions

This paper has reported on the applicability of Integrated Layer Processing (ILP). ILP aims at improving performance by reducing the number of memory accesses, but the employment of ILP includes some performance tradeoffs. We have summarized previous work in the area, and the important questions we have tried to answer are "When is ILP applicable?" and "When is a performance gain to be expected from applying ILP?". Our main conclusions are:

- **Constraints:** Constraints on the implementation as imposed by the protocol architecture may limit the applicability of ILP. Some protocol architectures have constraints that rightout exclude the application of ILP. More common however are constraints that make an ILP implementation so much more complex than a traditional sequential implementation that the performance gain from the memory access reduction is lost.
- **Copying:** In systems where data is copied several times between buffers, the relative performance gain of ILP is small. Since data copying is a form of data manipulation suitable for inclusion in an ILP loop (shown

by several successful implementations as reported on in section 3), we advocate the use of ILP in combination with a minimal copy architecture that avoids copying whenever possible.

- **Caching:** *Cache hit rates* has previously been found to be lower for ILP than for a corresponding sequential implementation [11]. This is however due to the lower overall number of memory accesses for ILP, and is thus not affecting the performance of ILP negatively. From our experiments, we conclude that the cache temperature (i.e. whether buffer data is in cache or not when the buffer reaches the data manipulation function) is typically *not* a main performance affecting factor for ILP. For sequential implementations, the cache temperature is more important, since each data manipulation function then must fetch data from the memory system.

  Internal cache conflicts, however, may have a more serious impact on performance, e.g. if the input and output buffer of a function happens to be mapped to the same location in a direct mapped cache. The conclusion from our experiments is that the layout of data buffers (and other data) in the cache should be controlled in order to avoid such internal conflicts. This is true for ILP as well as for sequential implementations.
- **Performance affecting factors:** We have found a main factor affecting ILP performance to be the availability of processor registers. The compound state of the integrated functions is likely to be larger than each separate function's state. Hence, ILP is more likely to experience a performance hit as the compiler runs out of available registers. When this happens, ILP performance degrades quickly with increasing state sizes, since the overflowing state has to be stored in memory between iterations in the ILP loop.
- **Performance modeling:** We have presented a model that captures the main factors affecting ILP performance. Using this model, we are able to predict the performance to expect when applying ILP.

To summarize our findings, the applicability of ILP, as well as the performance to expect from an ILP implementation, depends heavily on both the protocol architecture and the host system architecture.

The compound state size of the ILP loop is a main performance affecting factor. Functions requiring a large state to be kept between iterations are therefore typically not suitable for ILP implementation. Hence, ILP is probably mainly useful when integrating simple functions into simple loops, and thus the use of automated tools for this integration may be overkill.

A performance model like the one presented in section 6 can help a communication subsystem implementor to determine whether an ILP implementation is likely to increase performance or not, before attempting such an implementation.

References

[1] Peter Druschel, Mark B. Abbott, Michael A. Pagels, and Larry L. Peterson, "Network subsystem design," *IEEE Network*, vol. 7, no. 4, pp. 8–17, July 1993.

[2] Jonathan M. Smith and C. Brendan S. Traw, "Giving applications access to Gb/s networking," *IEEE Network*, vol. 7, no. 4, pp. 44–52, July 1993.

[3] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, 1994, ISBN 1-55860-281-X.

[4] Wm. A. Wulf and Sally A. McKee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, Apr. 1995.

[5] David D. Clark and David L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *SIGCOMM '90 Conference Proceedings*, Philadephia, Pennsylvania, Sept. 24–27, 1990, pp. 200–208, ACM SIGCOMM Computer Communication Review, 20(4).

[6] Van Jacobson, "Talk about the "witless" host network interface," INTEROP, San Jose, California, Oct. 1991.

[7] C. Partridge and S. Pink, "A faster UDP," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, Aug. 1993.

[8] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley, "Afterburner," *IEEE Network*, vol. 7, no. 4, pp. 36–43, July 1993.

[9] Mark B. Abbott and Larry L. Peterson, "Increasing network throughput by integrating protocol layers," *IEEE/ACM Transactions on Networking*, vol. 1, no. 5, pp. 600–610, Oct. 1993.

[10] Per Gunningberg, Craig Partridge, Teet Sirotkin, and Björn Victor, "Delayed evaluation of of gigabit protocols," in *Proceedings of the 2nd MultiG Workshop*, Electrum, Stockholm-Kista, Sweden, June 17, 1991.

[11] Torsten Braun and Christophe Diot, "Performance evaluation and cache analysis of an ILP protocol implementation," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 318–330, June 1996.

[12] Michael Fry and Atanu Ghosh, "Some issues for dynamic synthesis of ALF/ILP systems," *The Australian Computer Journal*, vol. 28, no. 2, pp. 61–65, May 1996.

[13] R. Atkinson, "IP authentication header," Internet RFC 1826, Aug. 1995.

[14] R. Atkinson, "IP encapsulating security payload (ESP)," Internet RFC 1827, Aug. 1995.

[15] Philip R. Zimmermann, *The Official PGP User's Guide*, MIT Press, 1995, ISBN 0-262-74017-6.

[16] Torsten Braun and Christophe Diot, "Automated code generation for integrated layer processing," in *IFIP Protocols for High Speed Networks*, Sophia-Antipolis, France, Oct. 28–30 1996.

[17] Shin-Yuan Tzou and David P. Anderson, "The performance of message-passing using restricted virtual memory remapping," *Software—Practice and Experience*, vol. 21, no. 3, pp. 251–267, Mar. 1991.

[18] Peter A. Steenkiste, "A systematic approach to host interface design for high-speed networks," *IEEE Computer*, vol. 27, no. 3, pp. 47–57, Mar. 1994.

[19] Bengt Ahlgren, Per Gunningberg, and Kjersti Moldeklev, "Increasing communication performance with a minimal-copy data path supporting ILP and ALF," *Journal of High Speed Networks*, vol. 5, no. 2, pp. 203–214, 1996.

[20] Bengt Ahlgren, Mats Björkman, and Kjersti Moldeklev, "The performance of a no-copy API for communication," in *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Mystic, Connecticut, USA, Aug. 23–25 1995.

[21] Peter Druschel and Larry L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, USA, Dec. 5–8, 1993, pp. 189–202, ACM SIGOPS Operating Systems Review, 27(5).

[22] Bengt Ahlgren, Mats Björkman, and Per Gunningberg, "Towards predictable ilp performance—controlling communication buffer cache effects," *The Australian Computer Journal*, vol. 28, no. 2, pp. 66–71, May 1996.

[23] Bengt Ahlgren, Mats Björkman, and Per Gunningberg, "Integrated layer processing can be hazardous to your performance," in *IFIP Protocols for High Speed Networks*, Sophia-Antipolis, France, Oct. 28–30 1996.

[24] Bengt Ahlgren, "A performance model for integrated layer processing," in *Seventh IFIP Conference on High Performance Networking*, White Plains, NY, USA, Apr. 28–May 2, 1997.