Integrated Layer Processing Can Be Hazardous to Your Performance*

Bengt Ahlgren

Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden.

E-mail: Bengt.Ahlgren@sics.se

Mats Björkman

Per Gunningberg

Uppsala University, Dept. of Computer Systems, Box 325, S-751 05 Uppsala,

Sweden. E-mail: Mats.Bjorkman@docs.uu.se, Per.Gunningberg@docs.uu.se

Abstract

Integrated Layer Processing (ILP) has been presented as an implementation technique to improve communication protocol performance by reducing the number of memory references. Previous research has however not pointed out that in some circumstances ILP can *significantly increase* the number of memory references, resulting in lower communication throughput.

We explore the performance effects of applying ILP to data manipulation functions with varying characteristics. The functions are generated from a set of parameters including input and output block size, state size and number of instructions. We present experimental data for varying function state sizes, number of integrated functions and instruction counts.

The results clearly show that the aggregated state of the functions must fit in registers for ILP to be competitive.

Keywords

ILP, Integrated Layer Processing, performance, protocol implementation

*Supported by the Swedish National Board for Industrial and Technical Development, under Esprit Basic Research Action project HIPPARCH EC-AUS004.

W. Dabbous et al. (eds.), *Protocols for High-Speed Networks V*

© Springer Science+Business Media Dordrecht 1997

1 INTRODUCTION

Data manipulation in software is a major bottleneck for communication performance, because it involves memory accesses to every byte of the message data (Druschel, Abbott, Pagels and Peterson, 1993; Smith and Traw, 1993). Since the speed of CPUs is increasing at a faster rate than the latency of memory systems is decreasing, this bottleneck is becoming more severe (Patterson and Hennessy, 1994; Wulf and McKee, 1995).

Data manipulation functions all have in common that they read message data, make some computation that transform the data, and possibly write new message data. Examples of data manipulation functions are checksum calculation, presentation encoding and encryption.

Integrated Layer Processing, ILP, is a protocol implementation technique introduced by Clark and Tennenhouse (1990). The purpose with ILP is to reduce the number of memory accesses needed by the data manipulation functions. 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.

ILP has been shown to considerably improve throughput for simple functions (Abbott and Peterson, 1993; Gunningberg, Partridge, Sirotkin and Victor, 1991; Partridge and Pink, 1993) and to improve throughput less for more complicated functions and when used in complete systems (Braun and Diot, 1995; Gunningberg, Partridge, Sirotkin and Victor, 1991).

We will show that depending on the characteristics of the manipulation functions, an integrated implementation can perform better than or much worse than a corresponding sequential implementation. The key question 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.

We use an experimental method to investigate how data manipulation functions with varying demands for CPU registers respond to integration. We generate 'synthetic' data manipulation functions from a set of parameters, such as state size, input and output block size, and the number of arithmetic instructions. The method gives us complete control over the selection of parameters and thus over the behavior of the manipulation functions.

The main contribution of this paper is that an integrated implementation can perform significantly worse than a sequential implementation when the CPU registers are too few to hold the aggregated state from all functions.

The rest of the paper is organized as follows. The next section describes data manipulation functions and ILP in detail and defines terminology used in later sections. Readers familiar with the topic may want to skip directly to Section 3, where we describe the experimental method. Section 4 presents the results of the experiments and discusses the factors affecting the relative performance of integrated implementations compared to sequential implementations. Section 5 discusses the generalization of the results to computer systems in general and the last section contains the conclusions.

2 DATA MANIPULATION FUNCTIONS AND ILP

Protocol processing can be divided into two parts: protocol control and data manipulation. The control part processes protocol headers and controls the state of a connection. The data manipulation functions operate on the user data part of a message. The manipulation function processes blocks of data from a message input buffer until it is empty. The transformed data is written to an output message buffer. The size of the input block depends on the function, and can vary considerably. The TCP/IP checksum uses two bytes, a byte swap function uses a word, an XDR encoder uses multiples of four bytes, an MPEG encoder uses 256 bytes, etc.

Below we have illustrated with pseudo-code a data manipulation loop. The loop successively reads input blocks of data from the input buffer until it reaches the end of the buffer.

```
for (i=0; i<input_buffer_size; i+=input_block_size) {
    read(input_block)
    /* MANIPULATION_INSTRUCTIONS */
    write(output_block)
}
```

The performance of a data manipulation function depends on the access time to input and output buffers as well as on the processing time for the manipulation instructions. On a modern RISC processor there is a large delay penalty when data has to be fetched from primary memory. This penalty can be substantial for simple data manipulation functions. It is therefore important that frequently used variables and data structures in the manipulation are put in registers by the compiler.

A data manipulation can be described by:

- **An input data block:** The size of the input block is fundamental to the function. The whole block must be present in order to do the processing. Message data is most likely found in primary memory unless it has been recently accessed. The input block is loaded from primary memory through the cache to the registers for processing.
- **An output data block:** The manipulation may expand or compress the input data block or produce an output block of the same size, depending on the particular function. The block is normally stored to the output buffer at the end of a manipulation. Some functions, like checksums, do not have an output data block.
- **A state:** Some manipulation functions use the state from previous iterations or some initialized context. 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. Since the state variables are used for each input block it is desirable that they stay in registers from the processing of one block to the next. The state must be carried over to other messages when the data manipulation needs several messages to finish.
- **Table look-ups:** One difference between a table and the state is that a table is not updated. A table is expected to be large and to have relatively few accesses to it. It is therefore likely that most of the table will stay in primary memory and only the cache lines which are accessed by table lookups will be brought into cache. These lines will however evict other cache lines in an unpredictable way, which will affect performance. If these lines are not accessed in a long time they will eventually be evicted.

- **Constants:** Constants are explicit in the code. The difference we make between a constant and a table entry is that the table is sparsely accessed and too large to fit into registers. The constants are accessed each time the manipulation is performed. They are therefore candidates for register allocation by the compiler.
- **Temporaries:** A data manipulation function will also use a number of temporaries which are not visible outside the function. Depending on their usage, a compiler may assign them to registers.
- **Instructions:** A data manipulation function has both control instructions and arithmetic-logic instructions. When referring to the number of instructions in a data manipulation in this paper we mean the number of arithmetic-logic instructions. The code for a manipulation loop is typically small enough to fit into cache. For unified caches, some instructions may be thrown out due to conflicts with data, such as table access data, which will prolong the processing time. Function calls in the data manipulation code reduce locality and increase the risk of address conflicts in the cache, see (Mosberger, Peterson and O'Malley, 1995) for a discussion on this effect.

2.1 Integrated Layer Processing

The basic idea behind ILP is to perform all the manipulations in one or two processing loops, instead of performing several manipulation loops sequentially as is most often done today. With an *ILP loop* we mean a loop that consists of an input data block read from memory, followed by a 'pipeline' of several manipulations and an output data block write. In such a pipeline each manipulation successively transforms this block of data until the last manipulation is applied. Thereafter the fully transformed block is written back.

The rationale for this loop integration is that the number of time consuming memory accesses to input and output data buffers in the pipeline can be reduced. Instead, these accesses will be to the transformed block which is assumed to be located in registers. This is achieved at the cost of a larger aggregated state since all manipulation function states must be kept for the next input block to the pipeline.

For design of and implementation details on ILP loops we refer to Braun and Diot (1995), Abbott and Peterson (1993) and to Gunningberg et al (1991).

An ILP loop is illustrated below. Assume n manipulations to be integrated in the loop. They are sequentially ordered, such that manipulation i is executed before manipulation $i + 1$.

```
for (i=0; i<input_buffer_size; i+=input_block_size) {
    read(input_block)
    /* MANIPULATION_INSTRUCTIONS from function 1*/
    /* MANIPULATION_INSTRUCTIONS from function 2*/
    /* ... */
    /* MANIPULATION_INSTRUCTIONS from function n*/
    write(output_block)
}
```

For very simple functions with few instructions, such as the TCP/IP checksum, byte alignment, etc, the time to read and write to memory dominates the execution time. For other functions, like encryption and some presentation encodings, the manipulation time may dominate. The

relative speed-up with an ILP loop for simple functions can be substantial while it is marginal for complex functions.

3 EXPERIMENTAL METHOD

We use the time (in nanoseconds) to process a byte 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. It is the job of the compiler to: (1) reduce the number of instructions, (2) schedule the instructions to avoid a stalled pipeline, (3) allocate registers in the most efficient way and (4) to generate code and data addresses with good locality. A skilled programmer can certainly structure code and data in order to lower the number of cache misses and for efficient register usage.

3.1 Generating data manipulation functions

To be able to study data manipulation functions with varying characteristics we want to automatically generate the desired functions from a set of parameters, such as state size, block size and count of arithmetic instructions (described in the previous section). We therefore have implemented a generator program that takes such 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. We actually implemented two generator programs, one producing an integrated implementation and the other producing a sequential implementation from the same set of parameters.

The generator programs produce C code to make the method more portable to different architectures. The drawback is that much care must be taken to make sure that the particular C compiler produces the desired machine code. When generating code that does not do anything useful, it is easy to accidentally produce ‘dead code’ that is optimized away by the compiler. The strategy of the generator programs is to minimize the use of temporary variables and make sure that the output is dependent on all input and on the result of all arithmetic instructions.

To validate that the generator programs produce code that is compatible to real data manipulation functions with respect to throughput, we have made a comparison with the BSWAP/PES/CKSUM combination of functions from Abbott and Peterson (1993). The parameters of the BSWAP/PES/CKSUM functions were used as input to the generator programs and the result was compared to the real implementations (see Table 1) on our two measurement systems, the HP 9000-735/90 and the SPARCstation 2. On the HP, the synthetic implementations performed virtually identical to the corresponding real implementations. On the SPARCstation there was a small discrepancy which we traced the reason for to the C compiler making a slightly different register allocation.

3.2 Measurement method

We have 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 described above. In the integrated case, there is a single loop. In the sequential case, there is

Table 1 Performance comparison of synthetic and real implementations of the BSWAP/PES/CKSUM loop.

Mbit/s	cold cache		warm cache	
	Seq.	ILP	Seq.	ILP
HP real	83.9	163.8	147.3	217.7
HP synthetic	83.9	163.8	147.3	218.5
SS-2 real	23.2	46.0	34.8	64.3
SS-2 synthetic	22.8	43.0	33.9	58.3

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 program measures the time over single invocations of a loop using the standard `gettimeofday()` system call. In both our measurement systems, SunOS 4.1.x and HP-UX Rel. 9, this call has a $1\ \mu\text{s}$ resolution. The results presented below are median values of 1000 measurements corrected for measurement overhead. The median value is often only one or two microseconds larger than the minimum measured value. The maximum value can, however, be almost any number because of interrupts and scheduling of other processes. The mean value is therefore less interesting. The buffer size is chosen so the measurement is approximately in the range $500\ \mu\text{s}$ to 5 ms. The lesser value is large enough to give accurate correction for measurement overhead, and the larger is small enough to be possible to run without getting disturbed by interrupts.

The program gives us complete control over how the input and output buffers are allocated and located in the cache. In the measurements in this paper, 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. For the SPARCstation 2, which have a unified instruction and data cache, we have also made sure that the executed code, both in the measurement program and in the `gettimeofday()` system call, do not conflict with the buffers, the stack or the global variables.

The program also controls the cache temperature of the buffers. If 'warm' is selected for a buffer, the program reads the buffer before the measurement loop to load the buffer into the cache. If 'cold' is selected, it flushes the buffer from the cache. 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.

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.

3.3 Dependency on the compiler

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.

The two compilers each have their peculiarities. The SunOS compiler is not very good at register allocation. This results in more variables than necessary allocated on the stack and code with additional loads and stores of these variables. This shows up in the measurements as a performance decrease when there are many local variables. The HP compiler on the other hand sometimes decides to slightly unroll loops by duplicating the loop body two or three times. This usually shows up in the measurements as a small performance increase.

When these features of the compilers affect the measurements described in the next section, it is pointed out and the effects are discussed.

4 FACTORS AFFECTING ILP PERFORMANCE

We have studied three factors that affect the relative performance of an ILP implementation compared to a sequential implementation. The experiments and results for each of the factors are presented in the following subsections.

The first factor is the *number of instructions* per data manipulation function. The results are the expected and show that the number of arithmetic 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 clearly show that this has a large impact on performance.

4.1 Instructions

In this experiment we vary the number of arithmetic instructions for each data manipulation function and observe the processing time per byte. The other parameters are kept constant. The input block size is set to 1, the number of state variables is set to 2 and the number of functions is set to 3. The aggregate state of the functions is small enough to fit into registers.

We vary the number of arithmetic instructions in each function, from 4 up to 100 instructions. For simplicity, we use the same number of instructions in each function. We have measured ILP loops as well as sequential loops for both warm and cold caches.

We expect the processing cost to be linear to the number of arithmetic instructions in the loops, since these arithmetic instructions only add to the CPU execution time, and do not affect the number of memory accesses. The code for the loops fits in the cache and are expected to stay there after an initial load.

In Figures 1 and 2 the measurement results for SPARCstation 2 and HP 9000-735/90 are

Figure 1 The processing cost of increasing number of arithmetic instructions per function (SPARCstation 2).

Figure 2 The processing cost of increasing number of arithmetic instructions per function (HP 9000-735/90).

presented. The unit on the X-axis is the number of arithmetic instructions in each function. The unit on the Y axis is the processing cost measured in ns/byte.

The results are generally what can be expected. The processing time for a byte basically scale linearly with the increasing number of arithmetic instructions needed for the manipulation. Both systems exhibit small deviations from the linear behavior. This is mainly due to the low-level instruction scheduling in the processor, and how well the compiler can take advantage of this scheduling.

In each figure there are four plots for the cases ILP loop with cold and warm cache and sequential loops with cold and warm cache. The plots are parallel as expected, with better performance for warm cache than cold. The ILP loops always perform better than the sequential loops for the measured selection of parameters.

The difference in processing cost between the ILP loop and the sequential loops should be: (1) the absolute gain for ILP when loads and stores to memory are reduced, and (2) the loop overhead avoided by ILP by having one single loop instead of one per function. For three functions, two loads and two stores can be avoided using ILP. The net gain will be higher for cold cache than for warm cache, since a cold load from main memory is more expensive than a load from cache.

The ILP gain for the SS2 is about 220 ns/byte for cold cache (135 ns/byte for warm) and for the HP about 30 ns/byte for cold cache (20 ns/byte for warm). This gain is substantial for an ILP loop with a small number of arithmetic instructions. For example, for ten instructions the throughput gain for ILP is 65% for the SS2 and 60% for the HP. For large manipulation functions, at the other end of the spectrum, the gain is more marginal. For example, for 100 instructions the gain is 11% for the SS2 and 4% for the HP.

The benefit of warm caches can also be observed in the figures. The SS2 benefits more from a warm cache than does the HP. The sequential case gain more from a warm cache, as expected since there are two more loads and stores for every input block as compared to the ILP case.

For ILP on the SS2 the gain from warm cache is about 44 ns/byte. On the HP the penalty for accessing the input block in primary memory is surprisingly small for a high performance RISC computer. It varies between 10 and 15 ns/byte for the ILP case and 25 to 40 ns/byte for the sequential case. The penalty is slightly higher for functions with few arithmetic instructions. This is probably caused by a stalled instruction pipeline. For few arithmetic instructions, the compiler has problems to schedule the loads and stores in an optimal way. That is not the case for ILP, because the compiler then has more arithmetic instructions to schedule between the load and store instructions.

The first conclusion from this experiment is that the absolute gain with ILP is not affected by the number of arithmetic instructions. The second conclusion is that the cost of a cold cache is constant, and that for the HP, this cost is small.

4.2 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 this 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.

The expected behavior is that when the state size is increased over some threshold defined by

Figure 3 The processing cost of increasing state size (SPARCstation 2).

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 below, because when this happens, ILP already behaves 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 3 and 4 clearly show the expected behavior.

Figure 4 The processing cost of increasing state size (HP 9000-735/90).

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 3), 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 9000-735/90 (Figure 4) 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, as in the previous experiment, 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.

4.3 Functions

The previous experiment showed how state size affects performance for a fixed number of integrated functions. This experiment will show how the number of integrated functions affect performance for different state sizes.

We vary the number of functions for a selected set of state sizes while keeping the other parameters constant. The values of the constant parameters are: 10 arithmetic instructions per function and a 1 word block size.

We assume that each of the integrated functions have the same state size. Since the factor affecting performance for ILP in this experiment is the sum of the function state sizes rather than the size of each individual function, these results are for ILP applicable to any set of functions where the sum of the state sizes add up to the same total size (when other function characteristics remain the same). For the sequential implementations, the results are applicable to the same sets of functions as for ILP above, as long as none of the executed functions have a state size larger than the number of available registers.

The expected behavior in this experiment is that as long as each function, or integrated set of functions, can keep its active state in registers, an additional function will introduce to a sequential implementation the additional costs of the extra loads and stores needed to access the data blocks in the message buffer, the loop overhead for the new function, and the arithmetic instructions executed in the added function. For the ILP implementation, the only overhead will be the arithmetic instructions executed in the added function. Hence, if ILP can keep its aggregate state in registers, ILP will perform better than a sequential implementation when adding a new function.

However, as was discussed in the previous experiment, the aggregate state of an ILP implementation is the sum of the states of all the integrated functions. Thus, the aggregated state is more likely to grow beyond the register capacity for an ILP implementation than it is for a sequential implementation. When this happens, the extra cost for ILP will be an additional load and store operation for each item that does not fit in registers. When this cost is larger than the additional cost of a sequential implementation (as above), ILP will perform worse than a sequential implementation.

Below we present experimental results for the two measured systems. In order to make the figures clearer, we have chosen to include only measurements with warm cache. As in the previous experiments, measurements with cold cache show a similar behavior as with warm cache, only with larger absolute costs because of the additional cache misses.

In each figure we show the behavior for four interesting state sizes. These sizes are different for each system because of the different number of available registers. For the sequential implementations, the performance difference between state sizes is so small that for clarity we show only one (the worst behaving) sequential implementation.

The experimental results presented in Figures 5 and 6 show that the behavior is the expected. The unit on the X-axis is the number of integrated functions.

On the SPARCstation 2, the increase in cost for an additional function is almost linear for the sequential implementations. For ILP, the cost is dependent on the total state size reached. At some point, the total state size exceeds the number of available registers. After this point, the cost for ILP grows faster than before the point. For state sizes larger than 2 per function, ILP costs then grow faster than sequential costs.

The expected growth of ILP costs should be proportional to the state size in each function. However, we can again see that the SPARCstation compiler is not very good at allocating

Figure 5 The processing cost of increasing the number of integrated functions (SPARCstation 2).

Figure 6 The processing cost of increasing the number of integrated functions (HP9000-735/90).

registers. For example, for ILP state size 5 words, we would expect the tangent of the linearly growing cost to cut the sequential cost somewhere between 1 and 2 functions. The failure of the compiler to use the available registers optimally adds extra load and store costs to all points above 1 function. This effectively moves the plotted line upwards in the figure.

The HP 9000-735/90 also exhibits the expected behavior. For state sizes 4 and above, ILP costs grow faster than sequential costs. On the HP, the compiler does better register allocation, resulting in more linearly increasing costs.

The conclusion of this experiment accentuates the results from the previous experiment in showing that when the ILP aggregated state cannot fit in registers, ILP performance declines even though we in these experiments make sure that the state not in registers remain in the cache between iterations.

5 DISCUSSION

The results presented in this paper 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.

6 CONCLUSIONS

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.

Complexity in the form of many arithmetic instructions does, however, not affect the absolute performance difference between an integrated and a sequential implementation.

REFERENCES

- Abbott, M.B. and Peterson, L.L. (1993) Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October.

- Braun, T. and Diot, C. (1995) Protocol implementation using integrated layer processing. In *SIGCOMM '95 Conference Proceedings*, pages 151–161, Cambridge, MA, USA, August 28–September 1. ACM SIGCOMM Computer Communication Review, 25(4).
- Clark, D.D. and Tennenhouse, D.L. (1990) Architectural considerations for a new generation of protocols. In *SIGCOMM '90 Conference Proceedings*, pages 200–208, Philadelphia, Pennsylvania, September 24–27. ACM SIGCOMM Computer Communication Review, 20(4).
- Druschel, P., Abbott, M.B., Pagels, M.A. and Peterson, L.L. (1993) Network subsystem design. *IEEE Network*, 7(4):8–17, July.
- Gunningberg, P., Partridge, C., Sirotkin, T. and Victor, B. (1991) Delayed evaluation of of gigabit protocols. In *Proceedings of the 2nd MultiG Workshop*, Electrum, Stockholm-Kista, Sweden, June 17.
- Mosberger, D., Peterson, L.L. and O'Malley, S. (1995) Protocol latency: MIPS and reality. Technical Report TR 95-02, Department of Computer Science, University of Arizona.
- Partridge, C. and Pink, S. (1993) A faster UDP. *IEEE/ACM Transactions on Networking*, 1(4), August.
- Patterson, D.A. and Hennessy, J.L. (1994) *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers. ISBN 1-55860-281-X.
- Smith, J.M. and Traw, C.B.S. (1993) Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July.
- Wulf, W.A. and McKee, S.A. (1995) Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, April.

BIOGRAPHY

Bengt Ahlgren received his MSc degree in Computer Science from Uppsala University in 1989. He joined the PhD program at the Department of Computer Systems at Uppsala University in 1990 and is currently working towards a PhD with expected graduation in 1996.

He has been employed full time as a researcher at the Swedish Institute of Computer Science (SICS) since 1989. The first few years he worked on communication security. Later work has been focused on high speed communication and the performance of protocol implementations and host/network interfaces.

Mats Björkman is an Assistant Professor at Uppsala University, Department of Computer Systems, since 1995. He received the MSc degree from Uppsala University in 1986 and the PhD degree from Uppsala University in 1993. He was a Research Associate at the University of Arizona in 1993 to 1994, where he worked in the x-Kernel project. Dr Björkman's research interests include high speed protocol implementations, especially host operating systems performance issues, and performance aspects of distributed systems.

Per Gunningberg is an Associate Professor at Uppsala University, Department of Computer Systems, Sweden, since 1995. He received the M.Sc. degree from University of California Los Angeles in 1981 and the Ph.D. from Uppsala University in 1983. Prior Uppsala University he was a researcher at the Swedish Institute of Computer Science. He has also been a Visiting Assistant professor at the UCLA Computer Science Department for one and a half year. His interests include protocol implementations, distributed operating systems and dependable computing.