Optimizing Layered Communication Protocols

Mark Hayden, Robbert van Renesse

Dept. of Computer Science, Cornell University

Abstract

Layering of protocols offers several well-known advantages, but typically leads to performance inefficiencies. We present a model for layering, and point out where the performance problems occur in stacks of layers using this model. We then investigate the common execution paths in these stacks and how to identify them. These paths are optimized using three techniques: optimizing the computation, compressing protocol headers, and delaying processing. All of the optimizations can be automated in a compiler with the help of minor annotations by the protocol designer. We describe the performance that we obtain after implementing the optimizations by hand on a full-scale system.

1 Introduction

We work with extensively layered group communication systems where high-level protocols are often implemented by 10 or more protocol layers. Layers provide many advantages, but introduce serious performance inefficiencies. This has led us to investigate compilation techniques which make layered protocols execute as fast as non-layered protocols, without giving up the advantages of using modular, layered protocol suites. This problem encompasses both systems issues involved in protocol performance, and compiler issues involved in designing optimization methods that can be automated.

The key idea of the approach we present is in the careful selection of the "basic unit of optimization." For optimization, our method automatically extracts a small number of common sequences of operations that occur in protocol stacks, called *event traces*. We provide a facility for substituting optimized versions of these traces at runtime to improve performance. We show how these traces can be mechanically extracted from protocol stacks and that they are amenable to a variety of optimizations that dramatically improve performance. We recommend event traces be viewed as orthogonal to protocol layers. Protocol layers are the unit of development in a communication system: they implement functionality related to a single protocol. Event traces, on the other hand, are the unit of execution. Viewing the system in this fashion, we can understand why it is important to focus on protocol layers in development, but on event traces when optimizing execution (see fig. 1).

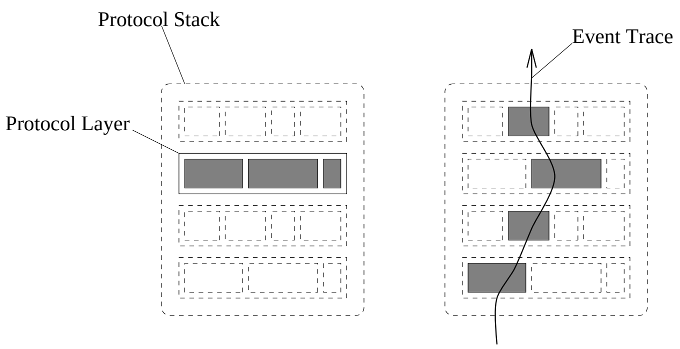

Figure 1: *Comparison of protocol layers and event traces. Layers are the basic unit of functionality. Traces are the basic unit of execution.*

In addition to the high performance of the optimized protocols, our methodology benefits from its ease of use. The protocol optimizations are made after-the-fact to already-working protocols. This means that protocols are designed largely without optimization issues in mind. The optimizations require almost no additional programming. Only a minimal amount of annotation of the protocol layers is necessary (the annotation consists of marking the start and end of the common paths of the source code), and this annotation is only made to the small portions of the protocols which are in the common path. In addition, the optimizations place few limitations on the execution model of the protocol layers.

The optimizations described in this paper have been implemented as part of the Ensemble communication system. We briefly describe Ensemble here. For an application builder, Ensemble provides a library of protocols that can be used for quickly building complex distributed applications. An application registers 10 or so event handlers with Ensemble, and then the Ensemble protocols handle the details of reliably sending and receiving messages, transferring state, detecting failures, and managing reconfigurations in the system. For a distributed systems researcher, Ensemble is a highly modular and reconfigurable toolkit. The high-level protocols provided to applications are stacks of small protocol layers. These protocol layers each implement several simple properties: they are composed to provide sets of high-level properties such as total ordering, security, virtual synchrony, etc.... Individual layers can be modified or rebuilt to experiment with new properties or change the performance characteristics of the system. This makes Ensemble a very flexible platform on which to do research. Ensemble is in daily use at Cornell.

The remainder of the paper proceeds as follows. In section 2, we describe the layering model in which we present the optimizations. In section 3, we present the general approach to extracting common paths from protocol stacks. In section 4, we present the optimizations for computation, for message headers, and for delaying operations. In section 6, we present the performance of these optimizations which have been used in the Ensemble communication system.

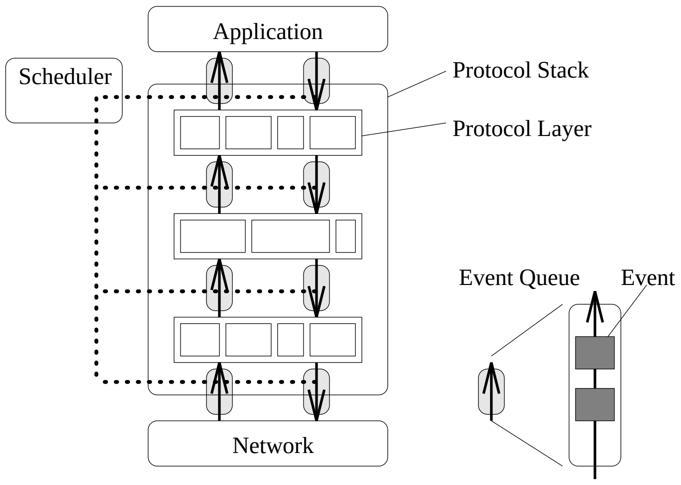

Figure 2: *Elements of the layering model.*

2 Layering Model

This work relies on a model of protocol layering, the design of which is central to the optimizations we present. We will describe our layering model here and in the remainder of the paper point to ways in which its features help enable the optimizations we describe (see fig. 2). The layering model consists of a set of components:

Event: Events are records used by protocol layers to pass information about messages. Most events contain a reference to a message, though not all do so.

Event queue: Events are passed between layers through directional event queues. Events placed in one end of an event queue are removed from the other end in FIFO order.

Protocol layer: A protocol layer implements a small protocol as an event driven automaton. An instance of a protocol layer consists of a local state record and handlers for processing events passed to it from adjacent layers. A layer interacts with its environment only through the event queues connecting it to adjacent protocol layers. For example, layers do not make any system calls or access any global data structures (other than memory management data structures).

Protocol stack: Protocol layers are composed to create protocol stacks, which are typically visualized as linear vertical stacks of protocols. Adjacent protocol layers communicate through two event queues, one for passing events from the upper layer to the lower layer, and another for the other direction.

Application and Network: The application communicates with the top of the protocol stack: messages are sent by introducing *send* events into the the top of the stack, and are received by through *receive* events that are emitted from the top. The network communicates with the bottom of the protocol stack. *Send* events that emerge from the bottom cause a message to be transmitted over the underlying network and a *receive* event to be inserted into the bottom of the stack of the destination.

Scheduler: The scheduler determines the order of execution of events in a protocol stack. The scheduler must ensure that events are passed between adjacent layers in FIFO order and that any particular layer is executing at most one event at a time. Also, all events must eventually be scheduled.

As an example, consider a simple case of sending a message. The application inserts a *send* event into the top of a protocol stack. The event is passed to the topmost protocol layer, which executes its handler on the event. The layer then updates its state and emits zero or more events. In a simple scenario, the same event gets passed from one layer to the next all the way to the bottom of the protocol stack. When the event emerges from the stack, the network transmits the message. The destination inserts a receive event into the bottom of the protocol stack. Again, in a simple scenario the event is repeatedly passed up to the top of the protocol stack and is handed to the application. In more complex situations, a layer can generate multiple events when it processes an event. For instance, a reliable communication layer may both pass a *receive* event it receives to the layer above it, and pass an *acknowledgment* event to the layer below.

This model is flexible in that the scheduler has few restrictions on the scheduling. For example, the model admits a concurrent scheduler where individual layers execute events in parallel. The optimizations we present do not make use of concurrency, but they do leverage off of the flexibility of the model.

2.1 Advantages and Disadvantages of Layering

The optimizations are primarily targeted at removing the overhead introduced by using layered communication protocols. This begs the question of why it is we are interested in layered communication protocols if their use causes serious performance degradation. Layered protocols have many advantages, some of which we list below:

- Layered protocols are modular and can often be combined in various ways, allowing the application designer to add or remove layers depending on the properties required. This way you only pay for what you use.
- When carefully decomposed into small layers, high-level protocols can be much more rapidly developed and tested than large, monolithic protocols.
- There are many cases where different protocols are substitutable for one another. The variations may each have different behaviors under different workloads. In a layered system, application designers can select the suite of protocols most suited to their expected work load. In addition, Ensemble supports changing protocol stacks underneath executing applications, so the application can tune its protocol stack to its changing work load.

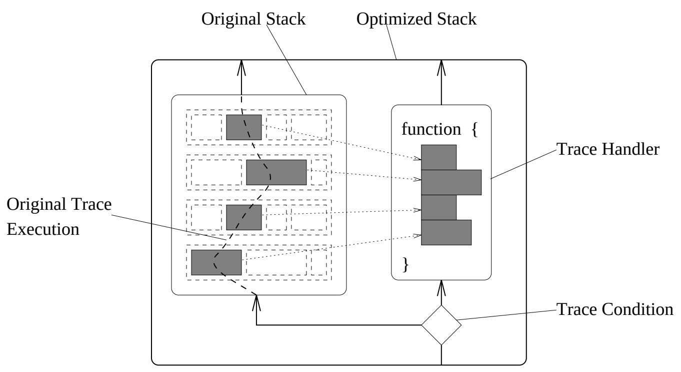

Figure 3: *Event traces, trace handlers, and trace conditions. The original protocol stack is embedded in an optimized protocol stack where events that satisfy trace conditions are intercepted and execute through heavily optimized trace handlers. Pictured is the original execution of the event trace and the interception of that trace with a trace handler. In a full example, multiple traces are optimized with each trace having its own trace condition and handler. In addition there would be traces starting both at the bottom and the top of the protocol stack.*

- We are also interested in verifying protocols. The current state of verification technology requires that the unit of verification be as small as possible. Small, layered protocols are just within the range of current verification technologies, whereas large, monolithic protocols are certainly outside this range.

The disadvantages of layered systems consist primarily of overhead, both in computation, and in message headers, caused by the abstraction barriers between layers. Because a message often have to pass through many (10 or more) protocol layers, the overhead of these boundaries is often more than the actual computation being done. Different system have reported overheads for crossing layers of up to $50\mu s$[RBM96]. The goal of this paper is to mitigate these disadvantages so that the use of layers is a win-win situation.

3 Common Paths in Layered Systems

Our methodology focusses on the common execution paths of communication systems. The first step in optimizing the common path involves identifying it. The old adage, “90% of the time is spent in 10% of a program,” says that most programs have common paths, even though it is often not easy to find the common path. However, carefully designed systems can often do a good job in exposing this path. In layered communication systems, the designer is often able to easily identify the normal case for individual protocols and these cases can be composed together to arrive at global sequences of operations. It is these sequences, or event traces, that we focus on as the basic unit of execution and optimization. For each event trace, we identify a condition which must hold for the trace to be enabled, and a handler that executes all of the operations in the trace. For the purposes of optimization, we introduce three more components to the layering model (see fig. 3):

Event trace: A sequence of operations in a protocols stack. In particular, we use the term to refer to the traces that arise in the normal case. An event trace begins with the introduction of a single event into a protocol stack. The trace continues through the various layers, where other events may be spawned either up or down. Often, a trace may be scheduled in various ways. It is assumed that one of these is chosen for a particular trace.

Trace condition: The condition under which a particular event trace will be executed. This consists of a predicate on the local states of the layers in a protocol stack and on an event about to be introduced to the stack. If the predicate is true then the layers will execute the corresponding trace as a result of the event.

Trace handler: The sequence of operations executed in a particular event trace. If the trace condition holds for a trace handler, then executing the handler will be equivalent to executing the normal operations within the protocol layers.

As an example, consider a type of event trace that occurs in many protocol stacks. When there are no abnormalities in the system, sending a message through a protocol stack often involves passing a *send* event directly through the protocol stack from one layer to the next. If messages are delivered reliably and in order by the underlying transport, then the actions at the receive side involve a *receive* event filtering directly up from the network, through the layers, to the application. Such an event trace is depicted in fig. 3. Both the send and receive event traces are called *linear* traces because (1) they involve only single events, and (2) they move in a single direction through the protocol stacks.

Complex event traces. Many protocol stacks have event traces that are not linear. Non-linear traces have multiple events that are passed in both directions through the protocol stack. Non-linear event traces are important because they occur in many protocol stacks. Without support for such traces, these stacks could not be optimized. Examples of such protocols include token-based total ordering protocols, broadcast stability detection protocols, and hierarchical broadcast protocols. We will describe an example routing protocol in more detail.

A *hierarchical routing protocol* is one in which a broadcast to many destinations is implemented through a spanning tree of the destinations. The initiator sends the message to its neighbors in the tree, who then forward it to their children, and so on until it gets to the leaves of the tree which do not forward the message. Some of the traces in a hierarchical routing protocol would include the following, the first two of which are linear and the last is non-linear:

1. Sending a message is a linear trace down through the protocol stack.

2. If a receiver is a leaf of the routing tree, then the receipt is a linear trace up through the stack.

3. If a receiver is not a leaf of the tree, the receipt will be a trace where: (1) the receive event is passed up to the routing protocol, (2) the receive event continues up to the

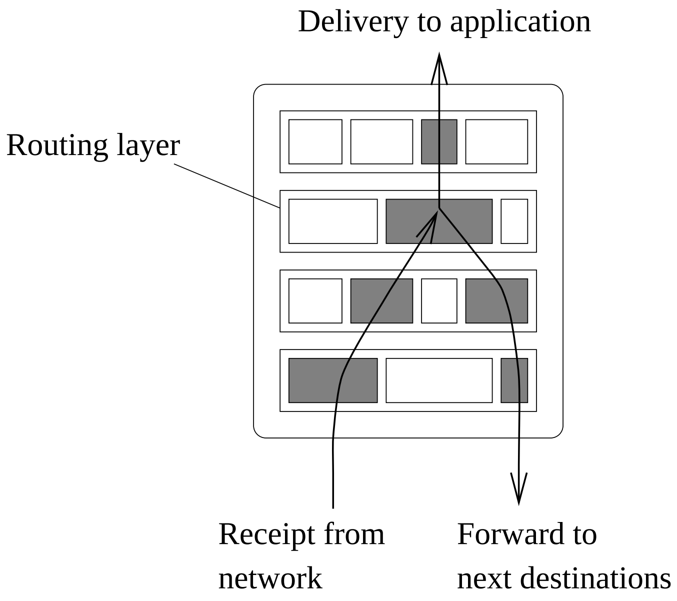

Figure 4: *A complex, non-linear trace in a routing protocol stack. A message is receive from the network and passed to the routing layer. The routing layer forwards a copy down to the next destination and passes a copy to the network.*

application, and (3) another send event is passed down from the routing protocol to pass the message onto the children at the next level of the tree (see fig. 4).

Determining and composing event traces. Determining event traces requires some annotation by protocol designers. They must identify the normal cases in the protocol layers, mark the conditions that must hold, and the protocol operations that are executed. From this, the traces can be generated by composing the common cases across multiple layers. Note that entire layers are not being annotated and no additional code is being written: the annotation is done only for the common cases, which are usually a small portion of a protocol layer.

Intercepting event traces. Given the event traces of a protocol stack, we can build alternative versions of the code executed during those traces and modify the system so that before introducing an event into a protocol stack, we check to see whether one of the event conditions is enabled. If not, then the event is executed in the protocol stack in the normal fashion, and checking the conditions has slowed the protocol down a little. If a trace condition holds, then the normal event execution is intercepted and we execute the trace handler instead. The performance improvement then depends on the percentage of events for which the trace condition is enabled, the overhead of checking the conditions, and how much faster the trace handler is.

The use of a trace handler assumes that there are no events pending in any of the intervening event queues. If there were a pending event, the trace handler would violate the model because the events in the trace would be executed out of order with regard to the previously enqueued event. Our solution to this problem relies on the flexibility of the layering model, and works by using a special event scheduler that executes all pending events to completion before attempting to bypass a protocol stack. This ensures that there are no intervening events.

Soundness of the transformation. The transformation of the protocol stack maintains correctness of the protocols becausetrace handlers execute exactly the same operations as could occur in the normal operation of the protocol layers. If the original protocols were correct, then the trace protocols are as well. This is leveraging off of the layering model in several ways. The first is that the layers are allowed to interact with their environment only through event communication. Thus we know that all of a layer's protocol is contained in its event communication and state updates. The second is that the layering model allows the operations executed by the layer to be executed in any fashion as long as it is equivalent to a legal scheduling of the events.

4 Optimizing Event Traces

The previous section described the general technique of extracting common paths as event traces. This section describes how these event traces are then optimized. We divide these optimizations into three classes: members of the first improve the speed of the computation; the second compress the size of message headers; and the third reorder operations to improve communication latency, without affecting the amount of computation.

4.1 Optimizing Computation

The first optimizations are those that improve the performance of the computation in event handlers. The general approach here is to carry out a set of transformations to the protocol stack in order that traditional compilation techniques can be effectively applied.

First pass: extract the source code. The first step extracts the source code for the trace condition and trace handler from the protocol layers. We break the operations of a stack into two types: protocol and layering operations. Protocol operations are those that are directly related to implementing a protocol. These include operations such as message manipulations and state update. Layering operations are those that result from the use of layered protocols. These include the costs of scheduling the event queues and the function call overhead from all the layers' event handlers. Layering operations are not strictly necessary because they are not part of the protocols. Given an event trace and annotated protocol layers, we use the annotations to textually extract from each layer the protocol operations for the trace.

Second pass: eliminate intermediate data structures. The second step removes the explicit use of events in the protocol layers. In the layering model, event records are used to pass information between protocol layers in a stack. They contain temporary state about a message that follows the message through the layers and event queues. Each event must be allocated, initialized, and later released. It is not necessary to explicitly use events because event traces encompass the life of the initial event and all spawned events. The contents of the event record can instead be kept in local variables within the trace handler, which compilers are often able to place in registers.

Third pass: inline functions. The third step completely inlines all functions called from the trace handler. The payoff for inlining is quite large because the trace handlers form almost all of the execution profile of the system. Normally, code explosion is an important concern when inlining functions. However, code explosion is not an issue in this case because of several factors. There are only a small number of trace handlers which are normally not too large: the inlining is focussed on a small part of the system so code growth will not be large. Also, the functions called from trace handlers are normally simple operations on abstract data types, such as adding or removing messages from buffers. These functions are not recursive and do not call many other nested functions, so fully inlining them will typically add only a fixed amount of code.

Fourth pass: traditional optimizations. The fourth step is to apply traditional optimizations to the trace handlers. These can be very effective because the previous passes create large basic blocks which compilers can optimize a great deal. Also, constant folding and dead-code elimination can be effective due to the elimination of event records. For instance, if one protocol layer marks an event record's field with some flag to cause an operation to happen at another layer, this constant value can be propagated through the trace handler so that the flag is never set at the first layer or checked at the second layer.

4.2 Compressing Protocol Headers

The second class of optimization reduces the size of headers. The protocols layers in a stack push onto a message headers which are later popped off by the peer layer at the destination. We divide these headers into three classes, two of which we aim to compress:

Addressing headers: These headers are used for routing messages, including addresses and other identifiers. They are treated opaquely: i.e., protocols are only interested in testing these headers for equality. These can be compressed through so-called path or connection identifiers, as described below.

Constant headers: These include headers that are one of several enumerated constant values and specify the "type" of the message. For instance, a reliable transmission protocol may mark messages as being "data" or as "acknowledgments" with a constant header, and from this the receiver knows how to treat the message. These headers are compressed by our approach when they appear in the common path.

Non-constant headers: These include any other headers, such as sequence numbers or headers used in negotiating reconfigurations. These are not compressed.

These optimizations are based on the use of connection identifiers[vR96, Jac90, Kay95]. Connection identifiers are tuples containing addressing headers which do not change very often. All the information in these tuples are hashed into 32-bit values which are then used along with hash tables to route messages to the protocol stacks. MD5 (a cryptographic one-way hash function) is used to make hashing collisions very unlikely, and other techniques can be used to protect against collisions when they occur. The use of connection identifiers compresses many addressing headers into a single small value and all messages benefit from this compression. Although the main goal of header compression is to improve bandwidth efficiency, small headers also contribute to improved performance in transmitting the messages on the underlying network and in the protocols themselves because less data is being moved around.

We extend connection identifiers to contain an additional field called the "multiplexing index." This field is used to multiplex several virtual channels over a single channel. This use of connection identifiers allows constant headers to be compressed along with addressing headers. This is done by statically determining the constant headers that are used in a trace handler and creating a virtual channel for that trace handler to send messages on. The constant headers are embedded in the code for the receiving trace handler.

This optimization significantly reduces the header overhead of the protocol layers. Even though each of the constant headers are quite small, the costs involved in pushing and popping them becomes significant in large protocol stacks. In addition, by encoding these constant values in the trace code, standard compiler optimizations such as constant folding and dead code elimination are possible. As an example, consider protocols in Ensemble. In many protocol stacks (including ones with more than 10 protocol layers), traces often only contain one non-constant field. Without trace optimizations, these headers add up to 50 bytes. With compression, the total header size decreases to 8 bytes. 4 bytes of this is a connection identifier. The other 4 bytes is a sequence number (for instance). There is not much room for improvement. This 8 byte header can be compared with those in similar communication protocols, such as TCP (40 bytes, 20 bytes for TCP with header compression)[1], Isis[BvR94] (over 80 bytes), and Horus[RBM96] (over 50 bytes).

Managing multiple formats. Two related problems arise when additional header formats are introduced to protocol stacks that expect only a single format. The first problem occurs when a trace condition is not enabled for a message received with compressed headers (for example, out-of-order messages may not be supported by trace handlers). The message must be passed to the normal execution of the protocol but the message is not in the normal format. The second problem occurs when a trace handler inserts a message into a buffer and a protocol layer later accesses the message. The solution in both cases is to lazily reformat messages. Messages are reformatted by functions which regenerate constant fields and move non-constant fields to their normal location in the message. These reformatting functions can be generated automatically. For the first problem, the message is reformatted before

[1]Comparison with TCP and the TCP with header compression is more complicated than this. For instance, 20 bytes of the 40 byte TCP header is the IP header, which is overhead Ensemble also has when running over IP. In addition, TCP header compression is targeted toward serial links which allows more headers to be compressed than is possible in the general case.

being passed to the normal protocol stack. The protocol layers get the message as though it were delivered in the standard format.

In order to manage buffers contains messages in different formats, each message is marked as normal or compressed. Compressed messages are buffered along with their reformatting function. When a protocol accesses a compressed message, it first calls the function to reformat the message. For most protocols, the normal case is for a message to be buffered and later released without further accesses. Lazy reformatting is efficient in these cases because messages are being buffered in compressed form and no additional operations are carried out on the message. Handling these buffers requires some modification of the protocol layers, but the modification is required only in layers with message buffers, and even then the modification is very simple. The reformatting function needs to be stored with compressed messages, but this cost is offset by the decreased size of messages being buffered.

4.3 Delayed processing

The third class of optimizations improves the latency of the trace handlers without decreasing the amount of computation. The approach in [vR96] relies heavily on this class of optimization, whereas in our work this optimizations is made in addition to others that are more significant in our case. When a message is sent, there are some operations (such as determining a message's sequence number) which must be done before the message may be transmitted, and some which may be delayed until after the transmission (such as buffering the message). The effect of reordering operations is to decrease the communication latency. Similarly, at the receiver, some operations are delayed until after message delivery. Note that reordered operation do not affect the amount of computation done, they only give the appearance of lower latency.

Fully automating delayed operations is a difficult problem, requiring some form of data flow analysis to determine which operations can be delayed while still retaining correctness. However, protocols can be annotated to specify which operations can and which cannot be delayed.

5 Comments on Using the ML Programming Language

The Ensemble system is implemented entirely in the ML programming language. Ensemble is derived from a previous system, Horus[RBM96], written in C, and embodies a great deal of experience gained from Horus. ML has encouraged structural changes that have improved performance: with the optimizations in this paper, Ensemble is much faster than Horus, even though C programs generally execute faster than ML programs. Ensemble benefits from a design which has tremendously improved performance, and the use of ML has been essential in being able to rapidly experiment and refine Ensemble's architecture in order to make these optimizations. The use of ML does mean that our current implementation of Ensemble is somewhat slower than it could be, but this has been more than made up for by the ability to rapidly experiment with structural changes, and thereby increase performance through improved design rather than through long hours of hand coding the entire system in C. At any rate, advances in ML compilation technology are steadily eroding this cost factor.

The use of a functional-style language was a cause of concern for us initially. This led us to design Ensemble so as to avoid features which do not "translate" back into an efficient implementation in C. This restriction on the design does not mean that these features were not used at all, but that they were only used in limited ways. The constructs of ML that we were concerned about were exceptions, higher-order functions, and garbage collection. For instance, many ML idioms for loop constructs use higher order functions. This use of higher order functions is typically compiled efficiently by ML and has a simple efficient translation into C, so higher-order functions were used in this fashion. By limiting the use of features of ML, we were confident that if we ran into performance problems from ML, we could translate the system back to C with minimal effort.

The performance of ML has not been a problem. The compiler we use (Objective Caml[Ler96]) generates adequately efficient code for us. We believe this is partly because we avoid expensive language constructs. Further optimizations are squeezed out of Ensemble by translating trace handlers by hand into C in order to improve performance even more. Such translations benefit from the trace optimizations because only the trace handlers need to be translated. We found that this translation gave an improvement of about a factor of 2, depending on the platform. Advances in ML compiler technology are expected to decrease the effectiveness of translation to C.

The issues relating to the use of garbage collection are assuaged by not using garbage collected objects in Ensemble. To be more precise, all allocation that occurs in the common path is managed explicitly outside of the purview of the ML garbage collector. Because of this, garbage collections are triggered only occasionally and do not significantly impact Ensemble's performance.

6 Experience with the Optimizations

An implementation of an automated protocol compiler for the Ensemble communication system is underway for the optimizations described above. Although currently we do not automatically generate trace protocols, we have constructed trace protocols by hand in a fashion that can be readily automated. We describe Ensemble and the protocols being optimized, and describe our experiences in using these optimizations in a full scale system.

We emphasize that the optimized versions of the protocols are part of a distributed communication system in daily use. The protocols and optimizations are not a proof-of-concept mock-up or part of a prototype system. It is important to demonstrate that proposed optimizations can be deployed in a full-size system because it gives increased confidence in the applicability to real world use. Prototype systems can demonstrate the feasibility of an idea, but leave the possibility that crucial issues have not been fleshed out, or that the methods are unwieldy when used in a real system. As an example of the kinds of applications Ensemble is used in, we describe one of them: a highly available remote process management service. This service uses groups of daemons to manage and migrate remote processes. The Ensemble protocols support reliable, totally ordered communication between the daemons for coordinating distributed operations, and the protocols manage system reconfigurations resulting from machine failures. The daemons are in daily use in the Cornell Computer Science Department.

protocol suite	code-latency	delayed operations
normal	$1500\mu s$	none
trace/ML	$41\mu s$	$28 - 63\mu s$
trace/C	$26\mu s$	$37\mu s$

Figure 5: *Performance comparisons for various protocol stacks.*

The protocols measured here implement FIFO virtual synchrony[BJ87] and consist of 10 or more protocol layers. All the performance measurements are made on groups with 2 members, where the properties are roughly equivalent to those of TCP. Actual communication is over a point-to-point (UDP or ATM) or multicast (IP Multicast) transports that provide best-effort delivery and a checksum facility. As we are interested in the overheads introduced by our protocols, our measurements are only of the code-latency of our protocols with the latencies of the underlying transports subtracted out. We focus on two measurements in this analysis. The first is the time between receiving a message from the network and sending another message on the network, where the application does minimal work. This is called the protocol code-latency. The second measurement is the time it takes to complete the delayed operations after one receive and one send operation (this is the amount of computation that is removed from the common path by delaying operations). All measurements were made on Sparcstation 20s with 4 byte messages. Measurements are given for three protocol stacks: the unoptimized protocols, the optimized protocols entirely in ML, and the optimized protocols where the trace conditions and handlers have been rewritten in C. See fig. 5. The C version of the protocol stacks has approximately $5\mu s$ of overhead in the code-latency from parts of the Ensemble infrastructure that are in ML. These could be further optimized by rewriting this infrastructure in C. There are no delayed operations in the unoptimized protocol stack.

The time line for one round-trip of the C protocol is depicted in fig. 6. Two Sparcstation 20s are communicating over an ATM network using U-net[BBVvE95] which has one-way latencies of $35\mu s$. At $0\mu s$, process A receives a message from process B off the network. $26\mu s$ later the application has received the message and the next message is sent on the network. At $61\mu s$, process B receives the message and sends the next message at time $87\mu s$. Process A completes its delayed updates by time $62\mu s$. The total round-trip time is $122\mu s$, of which Ensemble contributes $52\mu s$.

7 Previous Work

Other approaches have been made to improve the performance of layered communication protocols. Work done in our research group on this problem has been described in [vR96]. In that work, protocols are optimized through the use of pre- and post-processing to move computation overhead out of the common path. Through this approach, the latency is greatly reduced, though the computation is not. Pre- and post-processing is done through a layering model where handlers are broken into the operations to be done during and after messaging

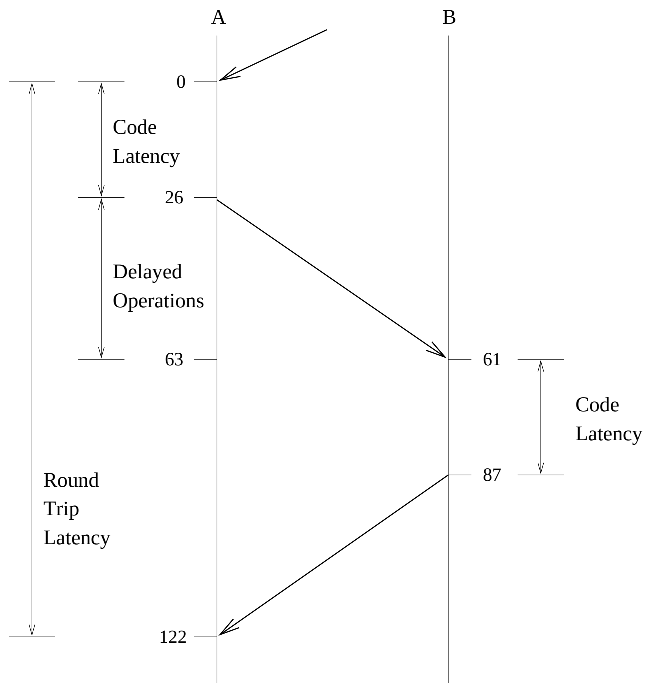

Figure 6: *Round-trip latency time-line between two processes. Vertical scale is in μs.*

operations (pre-processing for the next message is appended to the post-processing of the current message). The Protocol Accelerator also demonstrated the use of small connection identifiers to compress headers from messages and the use of message packing to achieve higher throughput.

VanRenesse's Protocol Accelerator, which is closely related prior work, achieves code-latencies of $50\mu s$ for protocol stacks of 5 layers. The total time required for pre- and post-processing one message send and one message receive operation is approximately $170\mu s$, with a header overhead of 16 bytes. This can be compared to code-latencies of $26\mu s$ in Ensemble, protocol headers of 8 bytes, and total processing overhead for a receive followed by a send of $63\mu s$, with a protocol stack that has more than twice as many layers. The advantages of the approach we have described over that of the Protocol Accelerator are:

- This approach decreases actual computation and layering overhead in addition to latency. While latencies of both approaches are comparable, the computational overhead is significantly smaller through our compilation techniques.

- These optimizations can be applied to a larger class of protocols than that of the Protocol Accelerator, including routing and total ordering protocols.

- The Protocol Accelerator approach requires structural modifications to protocols that are effectively annotations. Our approach requires significantly less annotation. We have demonstrated the use of our approach on a full-sized system.

- The primary requirement of our approach is the use of a strict layering model. The optimizations are not a necessary part of the layering model: protocol layers execute with or without the optimizations. This simplifies development because optimizations only need to be considered in the final stages of development.

Other related work has been done on Integrated Layer Processing[PHOA93, MPBO96]. ILP encompasses optimizations on multiple protocol layers. The optimizations we describe here are a form of ILP, but much of the ILP tends to focus on integrating data manipulations across protocol layers, whereas our optimizations focus on optimizing control operations and message header compression. On the other hand, ILP typically compiles iteration in checksums, presentation formating, and encryption from multiple protocol layers into a single loop to minimize memory references. Currently, none of the Ensemble protocols touch the application portion of messages. For instance, we assume the underlying network calculates checksums on the messages. The optimizations we present focus on aspects of protocol execution that are compatible and orthogonal to these other approaches, which we believe can be combined with ours.

8 Conclusion

We have demonstrated how common paths can be extracted from layered communication systems in a fashion that can be largely automated, and that doing so presents many opportunities for applying optimizations to the resulting code. We have applied these techniques to a real communication system and found them easy to use and have achieved significant performance improvements. The latencies were decreased from $1500\mu s$ to $26\mu s$.

We are examining opportunities for further improving the performance of layered protocols. These approaches all leverage off of the fact that the common paths have been extracted from the normal layer handlers as trace handlers. These handlers provide many opportunities for optimizations not carried out by traditional compilers. One possibility is to use dynamic code generation to compile trace conditions and handlers at run-time. By compiling handlers at run-time, more state is fixed, giving further payoffs from constant folding and other optimizations.

Acknowledgments

This work has also benefited from discussions with Anindya Basu, Greg Morrisett, and Scott Stoller. The distributed process management system described is by Takako Hickey. We appreciate the help of the developers of Objective Caml. Thanks for Julia Komissarchik and Robert Constable for comments on previous versions of this paper.

References

[BBVvE95] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. of the Fifteenth ACM Symp. on Operating Systems Principles*, pages 40–53, Copper Mountain Resort, CO, December 1995.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the Eleventh ACM Symp. on Operating Systems Principles*, pages 123–138, Austin, TX, November 1987.

[BvR94] Kenneth P. Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1994.

[Jac90] Van Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, Network Working Group, February 1990.

[Kay95] Jonathan Kay. *Path IDS: A Mechanism for Reducing Network Software Latency.* PhD thesis, University of California, San Diego, 1995.

[Ler96] Xavier Leroy. *The Objective Caml system release 1.02.* INRIA, France, October 1996.

[MPBO96] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of techniques to improve protocol processing latency. In *Proc. of the Proceedings of the 1996 ACM SIGCOMM Conference*, Stanford, September 1996.

[PHOA93] Larry L. Peterson, Norm Hutchinson, Sean O'Malley, and Mark Abbott. RPC in the x-Kernel: Evaluating new design techniques. In *Proc. of the Fourteenth ACM Symp. on Operating Systems Principles*, pages 91–101, Asheville, NC, December 1993.

[RBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.

[vR96] Robbert van Renesse. Masking the overhead of protocol layering. In *Proc. of the Proceedings of the 1996 ACM SIGCOMM Conference*, Stanford, September 1996.