17th ACM Symposium on Operating Systems Principles (SOSP '99)
Published as *Operating Systems Review*, **34**(5):80–92, Dec. 1999

Building reliable, high-performance communication systems from components

Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, Robert Constable[1]
Department of Computer Science
Cornell University, Ithaca, NY 14850

xliu,kreitz,rvr,ken,rc@cs.cornell.edu, jyh@cs.caltech.edu, hayden@pa.dec.com

Abstract

Although building systems from components has attractions, this approach also has problems. Can we be sure that a certain configuration of components is correct? Can it perform as well as a monolithic system? Our paper answers these questions for the Ensemble communication architecture by showing how, with help of the Nuprl formal system, configurations may be checked against specifications, and how optimized code can be synthesized from these configurations. The performance results show that we can substantially reduce end-to-end latency in the already optimized Ensemble system. Finally, we discuss whether the techniques we used are general enough for systems other than communication systems.

1 Introduction

Building systems from components has many attractions. First, it can be easier to design, develop, test, and optimize individual components of limited functionality than when the same functionality is embedded within a large monolithic system. Second, systems built from components may be more readily adapted and tuned for new environments than monolithic systems. Third, component-based systems may be extended at run-time with new components, facilitating evolution. And components can be individually formally specified and verified more easily, allowing safety-critical applications to be synthesized from such components.

The component-based approach has long been used successfully by VLSI designers. Yet, realizing the development of an efficient and correct software system from components has been difficult. One problem is that the abstraction barriers between the components impose high overheads arising from additional function calls, poor locality of data and code (leading to cache and TLB misses), redundant code that is not reachable in a particular configuration of the components (resulting in unused fields in data structures and message headers), and separate compilation (causing non-optimal use of existing optimization techniques).

Another problem is that configuring a system from components is often much harder than installing a shrink-wrapped system. Such a configuration includes the selection of components, the parameterization of the individual components, and the layout or composition of the selected components. Finally, although smaller components make it possible to prove the correctness of the algorithms or protocols used in each component, the generation and verification of the code of the components themselves is still very difficult.

In spite of these problems, we have seen a proliferation of component-based systems in the last two decades. The trend towards componentization is yielding systems that have more functionality than ever before. But these systems are generally not more reliable or faster, in spite of the tremendous improvements in hardware performance and compiler technology.

In this paper, we present encouraging, intermediate results of a cooperation between two groups at Cornell University: the Ensemble group communication group, and the Nuprl automated formal reasoning group. The project has two orthogonal goals: hardening communication protocol components and configurations of components, and improving their efficiency. This is approached by combining technology from a variety of different fields:

- (Communication protocols) Protocols are developed within the Ensemble framework. This architecture is

[1] The current address of J. Hickey is California Institute of Technology, CS Dept., M/S 256-80, Pasadena, CA 91125. The current address of M. Hayden is Compaq SRC, 130 Lytton Ave., Palo Alto, CA 94301. This work is supported in part by ARPA/ONR grant N00014-92-J-1866, ARPA/RADC grants F30602-96-1-0317 and F30602-98-2-0198, and NSF grant EIA 97-03470.

presented in Chapter 2 of [10]. Protocols are decomposed into *micro-protocols*, each specialized to do one task well in a specific environment. For different environments, the same task is implemented in different ways for improved efficiency. A particular micro-protocol implementation constitutes a component.

- (Programming languages) Components are written in Objective Caml (OCaml) [17], a dialect of ML [20]. For our project, the two important aspects of this choice are that 1) OCaml has a formal semantics that allows the code to be manipulated by formal tools, and 2) OCaml compiles to efficient machine code.

- (Specification) Specification of the behavior of both protocols and micro-protocols is done using I/O automata (IOA) [18]. IOA specifications are accessible to programmers because their syntax is close to standard programming languages such as C, and thus ease the transition between specification and code. Furthermore, IOA specifications can be composed, and have formal semantics.

- (Formal methods) For program reasoning and transformation we use the Nuprl [7] system. Nuprl is able to "understand" both the IOA specifications and the OCaml code, and can rewrite the code for the purpose of optimization. Nuprl's proof strategies can be tailored for reasoning about distributed systems, which we use for correctness proofs.

These parts each contribute to the realization of a reliable, high-performance, component-based communication system. By decomposing protocols into a large number of micro-protocol components, Ensemble provides a highly flexible communication system, but the implementation has the usual problems. The large numbers of module boundaries introduce significant latency, and application-specific protocols can be difficult to configure. We use the formal tools to address these problems. The OCaml programming language has a precise mathematical semantics that we use as the foundation for formal reasoning. IOA specifications provide the requirements for configuration and performance optimization, and we use Nuprl to automate the optimization. Eventually, we hope to leverage Nuprl for the configuration process as well.

We were able to synthesize code from the original implementations automatically, whose latency is within 50% of the best hand-optimized code generated for Ensemble. For example, we optimized a 4-layer stack that provides reliable multicast. On a 300 MHz SparcStation, the processing overhead for sending a message through the stack is 2 μs (down from 13 μs in the original Ensemble stack), and the overhead for delivery is 4 μs (down from 10 μs).

Significantly more work is required to optimize one protocol stack using our formal framework than to do so by hand. If one were to optimize a single monolithic protocol stack, such as TCP/IP, it would probably make sense to do so by hand. The two advantages to our approach are (1) the optimization is guaranteed to preserve correctness of the protocol and (2) once initial work has been done to prepare each protocol layer, it is possible to automate the optimization of all their combinations. Thus the fact that Ensemble is highly configurable is important. Ensemble protocol stacks can be combined in 1000's (if not more) ways. It would not be feasible to optimize them all by hand.

Although we have not yet completed a machine-generated proof of a distributed property (a property that involves more than one participant) of a non-trivial protocol, we have written specifications of four protocols and a manual proof of the total ordering protocol and its configuration in a virtually synchronous communication stack. This led to the discovery of a subtle bug. This effort has also hardened and simplified configuration of protocols in Ensemble, an otherwise difficult and error-prone process [11].

The Ensemble, OCaml, and Nuprl systems are described in Section 2. Section 3 describes how protocol behaviors are specified and protocol implementations and configurations are checked for correctness. In Section 4 we show how we use Nuprl in the optimization process, and we discuss the resulting performance. We review related work in Section 5. Section 6 discusses the feasibility of using our approach in other settings.

2 Ensemble, OCaml, and Nuprl

Ensemble is a high-performance network protocol architecture, designed particularly to support group membership and communication protocols. Ensemble's architecture is based on the notion of a protocol stack. The architecture is described in detail in Chapter 2 of [10]. The system is constructed from simple *micro-protocol* modules, which can be stacked in a variety of ways to meet the communication demands of its applications. Ensemble's micro-protocols implement basic sliding window protocols, fragmentation and re-assembly, flow control, signing and encryption, group membership, message ordering, and other functionality. Ensemble includes a library of over sixty of these components.

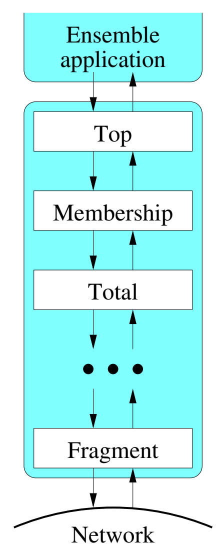


Figure 1. Ensemble architecture

Each module adheres to a common Ensemble micro-protocol interface. There is a top-level and a bottom-level part to this interface. The top-level part of the interface of a module communicates with the bottom-level part of the interface of the module directly on top of it. The interface is event-driven: modules pass event objects to the adjacent modules. Certain types of events travel down (*e.g.*, send events), while others (such as message delivery events) travel up the stack.

Ensemble is currently used in the BBN Aqua and Quo platforms, a fault-tolerant testbed at JPL, the Adapt adaptive multimedia middleware system at Lancaster University, a multiplayer game by Segasoft, and in the Alier financial database tools.

Ensemble is written in OCaml [17]. Chapter 4 of [10] explores the impact of building Ensemble in OCaml, including comparisons with Ensemble's predecessor, Horus [26], which was written in C. For instance, the code for Ensemble's protocols was found to be about 7 times smaller than functionally identical code in Horus. The reasons for this include OCaml's high-level manipulation of data structures, automated memory allocation and garbage collection, and automatic marshaling. [10] also reports that the Ensemble developers found the OCaml code easier to develop and maintain than C.

For our purposes, the main benefit of OCaml is that it has a precise definition that we can use during operations like synthesis, verification, and optimization. If performed manually, these operations can be tedious and error-prone. We use formal automation to address both problems—automation makes it possible to re-use common reasoning strategies, and formal reasoning guarantees the correctness of program transformations.

We are not the only ones using OCaml for systems programming. OCaml is also used in many systems, such as the Pentasoft financial simulation system, the Switchware active network at UPenn, a Mitre network filter, a network supervision system at France Telecom, and formal analysis of cryptographic protocols at Stanford Research International.

In the work we present here, Nuprl acts as a source-to-source translator. Unlike a compiler, every step made by Nuprl has to be accompanied by a proof, which often requires manual interaction. A semantics for the functional subset of OCaml, as well as the imperative language features that are required for implementing finite state-event systems, was developed in [14]. In particular, exceptions and support for object-orientation are currently not handled by Nuprl. Ensemble's implementation of micro-protocols only uses the corresponding subset. Tools convert OCaml code into the corresponding terms of Nuprl's input language and vice versa. This way, the Nuprl system can reason about OCaml expressions and evaluate them symbolically.

The resulting *logical programming environment* [16] provides the infrastructure for the application of formal verification and optimization techniques to the actual code of Ensemble. On this basis, we have developed formal techniques for domain-specific optimizations of layered systems [15, 10], which support the optimization of both individual layers and whole layer stacks.

In summary, our formal environment has three parts: 1) OCaml provides a precise code-level description of Ensemble, 2) IOA specifications provide high-level logical descriptions of the micro-protocol components, and 3) the Nuprl prover provides the support for automating the reasoning process.

3 Specifications and correctness

Specifications serve two purposes: 1) when we design a new protocol, the specification can be used to guide the selection of micro-protocols, and 2) by verifying an existing implementation against its specification, it becomes possible to determine if the program is *correct*. We cover these in the next two subsections.

3.1 Specification

Specifications of communication systems range along an axis from specifying the *behavior* of a system to specifying its *properties*. When specifying the behavior, we describe how the system reacts to events. For example, we may specify that the system sends an acknowledgment in response to a data message. When specifying properties, we describe logical predicates on the possible executions of the system. An example of a property is that messages are always delivered in the order in which they were sent (FIFO).

Both kinds of specifications are important. The properties describe the system at the highest level. Since the properties do not specify *how* to implement a protocol, they are easy to compose. Behavioral specification provides the connection to the code by describing how to implement the properties. Behavioral specifications can be either *concrete* or *abstract*. Concrete specifications can be directly mapped onto executable code, while abstract specifications are non-deterministic descriptions that use global variables, and are therefore not executable. The advantage of abstract specifications is that they are simple, and that global or distributed properties such as FIFOness can be easily derived. The advantage of a concrete specification is that an implementation can be easily derived from it. Below, when using the term *specification*, we will mean *behavioral specification* unless otherwise noted.

We will now describe some examples of both abstract and concrete behavioral specifications of networks and protocols. The programming model we use is that of a *state machine* with event-condition-action rules. This is not much different from an abstract data type, or a C++ or Java object, except that all interactions between components are through events. An abstract specification has a set of variables, and a set of events that, *under certain conditions*, modify these variables and allow interaction with the environment.

See Figure 2(a) for an example of an abstract specification of a FIFO network, using a variant of IOA as the specification language. The state consists of a single global queue that contains the messages that are in transit, paired with the respective destinations. The `Send` event specifies the destination of a message and the message itself, and adds the message to the queue. The `Deliver` event of a particular message at a particular destination can only trigger when that pair is at the head of the queue. If so, that pair is removed from the queue. No message loss or retransmissions, nor the fact that there is no shared memory to implement the global queue, needs to be considered. This specification is abstract because it is not executable: it uses a global variable, and the scheduling of events is not determined. Instead, each event has a condition that specifies under which circumstances the corresponding action can be evaluated.

Figure 2(b) specifies, again abstractly, a network that can arbitrarily lose and reorder messages, as well as deliver them

Specification FifoNetwork()
Variables
in-transit: queue of $\langle Address, Message\rangle$
Actions
$Send(dst : Address; msg : Message)$
condition: true
{ in-transit.append($\langle dst, msg\rangle$); }

$Deliver(dst : Address; msg : Message)$
condition: in-transit.head() == $\langle dst, msg\rangle$
{ in-transit.dequeue(); }

(a)

Specification LossyNetwork()
Variables
in-transit: set of $\langle Address, Message\rangle$
Actions
$Send(dst : Address; msg : Message)$
condition: true
{ in-transit.add($\langle dst, msg\rangle$); }

$Deliver(dst : Address; msg : Message)$
condition: in-transit.contains($\langle dst, msg\rangle$)
{}

$Drop(dst : Address; msg : Message)$
condition: in-transit.contains($\langle dst, msg\rangle$)
{ in-transit.remove($\langle dst, msg\rangle$); }

(b)

Figure 2. (a) The abstract behavioral specification of a FIFO network; (b) a network that reorders, duplicates, and loses messages.

Specification FifoProtocol($p : Address$)
Variables
send-window, recv-window, ...
Actions
$Above.Send(dst : Address; msg : Message)$
$Below.Send(dst : Address; \langle hdr, msg\rangle : \langle Header, Message\rangle)$
$Below.Deliver(dst : Address; \langle hdr, msg\rangle : \langle Header, Message\rangle)$
$Above.Deliver(dst : Address; msg : Message)$
$Timer()$

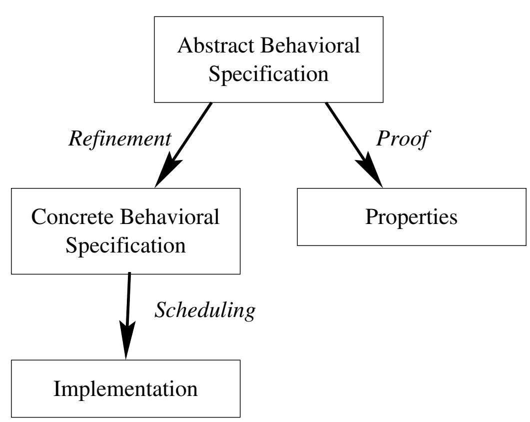
Figure 3. The prototype of a concrete behavioral specification of a communication protocol that retransmits messages, removes duplicates, and delivers messages in order. Each process that participates in the protocol runs an instance of one of these.

multiple times. In this case the state consists of an unordered set of messages. `Send` just adds a (destination, message) pair to the set, and `Deliver` delivers an arbitrary message in the set. There is a third, internal event that drops arbitrary messages from the set to model message loss.

The prototype (or *signature* in IOA terminology) of a concrete specification of a protocol that implements a FIFO network on top of a lossy network is presented in Figure 3. (Including the specification of the actions would take up too much space.) A concrete specification only involves state and events local to a single participant in the protocol. For every event the condition specifies if it can be executed. An *instance* of this specification has to run on each process that is a participant in the protocol. (Each instance has its own copy of the variables.) Note that a concrete specification has two `Send` events and two `Deliver` events. `Above.Send` is visible only to the application, while `Below.Send` is visible only to the network. There is also an internal timer event that takes care of retransmissions.

The difference between a concrete specification and an implementation is small: in an implementation, the programmer also needs to detect which conditions are true, and specify the order in which the corresponding actions should be executed. This does not affect the correctness of the implementation.

The relationship between abstract behavioral specifications, concrete behavioral specifications, implementations, and properties can be pictured as follows:



A concrete specification is derived from the abstract specification by a process called *refinement*. This involves designing a protocol that implements the abstract requirements. On the other hand, the properties of the abstract specification are derived by proof. The alternative, to derive an implementation directly from properties, or prove properties given the implementation, would be much harder.

If we wish to prove the correctness of FifoProtocol, we use an instance of LossyNetwork, and an instance of FifoProtocol for each participant in the protocol. We then compose them by tying all Below.Send events to the Send event in LossyNetwork, and similarly for the Below.Deliver events. Two events can be tied together by combining the conditions and actions of those events. We then have to show that any execution of this composed specification, which is an abstract specification, is also an execution of FifoNetwork. (An execution is basically the sequence of externally observable events of a specification.) A good example of such a proof, by hand, can be found in [11], which demonstrates the correctness of one of Ensemble's total ordering protocols. (This exercise located a subtle bug in the original implementation.)

3.2 Is Ensemble correct?

One way to answer this question is to first come up with an abstract behavioral specification of a network that implements the properties that some particular application requires. Then, we must show that the composition of the abstract specification of the actual network used, and the concrete specifications of the micro-protocols in the protocol stack, results in the same executions as that of the required abstract specification. Since there are many networks, many applications, and many micro-protocols, the correctness of Ensemble is no easy matter. But even for a single application, network, and protocol stack, this approach would be an almost impossible task.

The approach that we are taking instead is the following. For each micro-protocol p, we present two abstract specifications, $p.Above$ and $p.Below$. $p.Above$ specifies the behavior that the protocol implements, whereas $p.Below$ specifies the behavior that the protocol requires of the networking environment below it. When proving the correctness of a stack of protocols for a particular network and application, we can therefore limit ourselves to showing that, for each pair p and q of adjacent protocol layers (p below q), every execution of $p.Above$ is also an execution of $q.Below$ and vice versa. Since we are now working at the level of relatively simple abstract specifications, and $p.Above$ and $q.Below$ will generally be identical or similar, the task is tractable.

It would be nice if, given a particular application and network, we could automatically select a set of micro-protocols and generate a correct configuration. A brute force solution of trying all possible combinations obviously will not work. Instead, the Ensemble system contains an algorithm for calculating stacks given the set of properties that an application requires. This algorithm encodes knowledge of the protocol designers and appears to work quite well, but we cannot currently be sure that it always generates a correct stack. (We could check the correctness afterwards using the methodology described above, but this has not yet been automated.) Also, the heuristic only knows about approximately two dozen different properties, so it cannot generate stacks for applications with specific unique requirements.

4 Optimization

Maintaining good communication performance in a layered system is hard, and a variety of projects have tried to address this problem. For example, the VIA Giganet interface provides 10 μs one-way end-to-end latency for messages of 256 bytes or smaller [27]. A highly optimized layering system like Ensemble adds about 1 to 2 μs per layer to the latency of pure layering overhead, and having more than 10 layers is not uncommon.

Chapter 3 of [10] describes the following optimizations:

1. Avoiding garbage collection cycles. In the original Ensemble system, an OCaml object was allocated for each message sent or delivered. These relatively large objects tend to be short-lived, a scenario for which the OCaml garbage collector is not optimized. The Ensemble distribution now has its own message allocator, which is formed by a single OCaml string object. Ensemble is itself responsible for *freeing* messages. Using other careful coding techniques, creating short-lived objects was avoided in the normal case, greatly reducing garbage collection overhead.[1] In addition, by triggering garbage collection when the system is idle, it became highly unlikely that garbage collection is necessary during stack processing.

2. Avoiding marshaling. Ensemble allows any OCaml value to be sent and delivered. As a value is sent, each layer encapsulates the value into another one consisting of the header of that layer and the headers of the layers above it. The resulting data structure needs to be serialized before sending along with the message payload. (Note that there is no fixed wire format for headers in Ensemble.) Ensemble uses the OCaml value marshaler for this, which traverses the data structure, and copies all the data into a byte string. A similar operation, copying parts of the string back into dynamically allocated data structures, happens on delivery. In many common cases, the headers contain only one or two integers, and all this generality leads to substantial overhead. By providing specialized marshalers for these cases, and by using the UNIX scatter-gather capability, we can avoid this overhead.

3. Delaying non-critical message processing. For example, a message can often be sent or delivered before the message is buffered. Doing so takes the buffering overhead out of the so-called *critical path*, and reduces end-to-end message latency.

[1] OCaml now has a new compacting garbage collector that may reduce the effectiveness of these optimizations.

4. Identifying common paths and eliminating layer boundaries. A common technique for optimizing layered protocol implementations is to identify the common outcomes of if-statements, and *outline* the uncommon code (*i.e.*, move it into a different section of memory than the common path to improve locality). Using function inlining, the layer boundaries can be removed. The resulting common code path is compact and has better cache performance.

5. Header compression. Many layers need to add some information to the message header, and all layers contribute to some processing overhead for every message that is sent or received. Fortunately, most information in headers seldom changes, allowing for significant compression of headers, typically to just 16 bytes.

The first three steps do not affect the layering abstraction itself. However, the final two steps require generating special code for common cases. Finding common cases and compressing headers is far beyond the capabilities of current compiler optimization techniques, and therefore previous work [1, 2, 9, 21] involves hand-optimization or at least significant annotation of the code.

Both [1] and [13] report that this is a difficult and error-prone process, which is consistent with our own experience in trying to do so. Chapter 5 of [10] shows how such optimizations can be formalized using predicates and partial evaluation of code, using a 4-layer protocol stack as an example. In this section, we describe how we have developed this into push-button technology using Nuprl which can be applied to any protocol stack.

4.1 Formal optimization in Nuprl

To understand the way Nuprl interprets a protocol, it is useful to think of a protocol as a function. Such a function takes the state of the protocol (the collected variables maintained by the protocol) and an input event (a user operation, an arriving message, an expiring timer, ...), and produces an updated state and a list of output events. This function can be optimized if something is known about an input event and the state of the protocol. We express this knowledge by a so-called *Common Case Predicate* (CCP): a Boolean function on the state of a protocol and an input event. CCPs are specified by the programmer of a protocol, and are typically determined from run-time statistics. We call the result of optimizing a protocol a *bypass*.

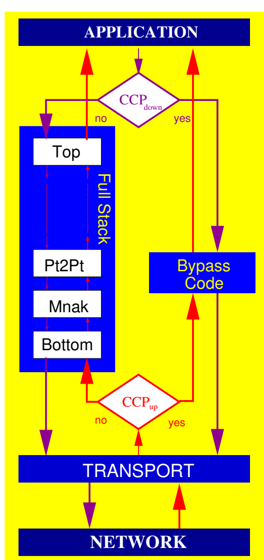


Figure 4. Optimized code architecture

For example, a CCP may be true if the event is a Deliver event, and the low end of the receiver's sliding window is equal to the sequence number in the event (in other words, this is the next expected packet to arrive, and it was not lost or reordered). If a message is received for which the CCP predicate is satisfied, that message may be delivered and the low end of the window moved up, without a need for buffering.

Bypass code fragments from a stack of layers can be composed to obtain a single bypass function guarded by a single common-case predicate. The same CCPs that are used at compile-time to generate the bypass code are also used at run-time to decide whether to deliver an event to the bypass or to the original stack, as illustrated in Figure 4. (The Transport module below the protocol stack provides marshaling of messages.) The CCP for a composed bypass is the logical conjunction of the individual CCPs for its components. It is necessary to generate efficient code for this CCP, since it will be executed for every event.

The partial evaluation of a layer removes all the unreachable code for the corresponding CCP condition. Often, as a result, much of the header for that particular layer becomes a constant, for example, because the CCP condition prescribes that the packet is an unfragmented DATA packet. All these constant header fields in the combined headers of all layers can be combined into a single, short, identifier, thus compressing the header and reducing handling overhead.

4.1.1 *Methodology*

There are two levels of formal optimizations. The first, or *static* level, depends solely on the code of the individual modules of the Ensemble toolkit. It is performed semi-automatically under the guidance of the programmer who developed a particular Ensemble protocol layer, and an expert on the Nuprl system. This optimization may take anything from 1/2 hour to an entire afternoon to develop. Its result is part of the optimization tool that is made available to the application developer.

The second, or *dynamic* level, depends on the particular communication stack that an application developer builds with the Ensemble toolkit and thus cannot be provided a priori. This level is completely automated and requires only the names of the protocol layers that occur in the application stack. Its result, which is typically obtained in less than 1/2 minute, is the complete code for an optimized protocol stack, the bypass.

It should be noted that optimization is orthogonal to verification. Our formal tools prove that the resulting code is semantically equal to the original protocol stack but do not make any assumptions about properties of the stack. Our optimization tool is built within the interactive Nuprl proof development system [7], which — due to its expressive logic — supports formal reasoning not only about mathematics but also about program properties, program evaluation, and program transformations. In the rest of this section we will explain the technical details of the optimization tool.

4.1.2 Static optimization of protocol layers

In order to be able to formally reason about Ensemble and to optimize its protocol stacks, we first have to import its code into the Nuprl proof development system. For this purpose we have developed a tool that parses the OCaml code of Ensemble's modules and converts it into terms of Nuprl's logical language that represent its formal semantics (see [14] for details). Nuprl's display mechanism allows us to give these terms the same syntactical appearance as the OCaml code they represent and thus to include "code fragments" in formal mathematical statements. This makes it easier to understand Nuprl's formal text and makes sure that all code transformations are semantically correct.

The static optimizations of the Ensemble toolkit are based on an a priori (before compilation) analysis of *possible* bypass paths through each individual protocol layer. A bypass path through a layer is a branch in the code of the layer. The CCP of this branch is the conjunction of the predicates in the conditions of the corresponding if-statements.

The optimization of a protocol layer proceeds by a series of code transformations that are based on the following basic mechanisms:

- *Function inlining and symbolic evaluation* simplifies the layer's code in the presence of constants or function calls. Logically, this means *rewriting* the code by definition unfolding and controlled partial evaluation.

- *Directed equality substitutions* such as the application of distributive laws lead to further simplifications of the code. Technically, we apply lemmata from Nuprl's logical library. Adding a direction to each lemma (*e.g.*, in the case of equality, whether the lemma should be applied from left-to-right or vice versa) guarantees termination of this process.

- *Context-dependent simplifications* help in extracting the relevant code from a layer. They trace the code path of messages that satisfy the CCPs and isolate the corresponding code fragments. Technically, we consult a CCP in order to substitute a piece of code by a value and then rewrite the result with the above two mechanisms.

All these mechanisms preserve the semantics of a layer's code under the assumption of the CCPs. To control their automated application, we restrict inlining to functions from a few specific Ensemble modules. Symbolic evaluations and distributive laws can be applied to any expression in the code, but the automatic strategy chooses the outermost possible. The fact that the code of all protocol layers has a common outer structure allows us to start the optimization of a layer with a predetermined sequence of controlled simplifications.

Optimizations for each layer are initiated for four fundamental cases: down or up going events (sending or receiving messages) for both point-to-point sending and broadcasting. The CCPs for these cases are created automatically and the Ensemble programmer may add additional CCPs to describe the common case. After applying the initial fixed series of simplifications we reach a choice point (*e.g.*, a conditional) in the code. The system then applies context-dependent simplifications for each of the CCPs until the corresponding code fragment has been isolated. All these steps are completely automated and usually the optimization can stop at this point. However, we give the Ensemble programmer an opportunity to invoke additional simplifications explicitly before committing the result to Nuprl's logical library.

Performing these steps in a logical proof environment guarantees that the generated bypass code is equivalent to the original code of the layer if the CCP holds. In most cases this means that about 100-300 lines of code have been reduced to a single update of the layer's state and a single event to be passed to the next layer.

4.1.3 Dynamic optimization of application stacks

In contrast to individual layers, application protocol stacks cannot be optimized a priori, as thousands of possible configurations can be generated with the Ensemble toolkit. But the application developer has little or no knowledge about Ensemble's code. Therefore, an optimization of an application stack has to be completely automatic. We have developed a tool for optimizing arbitrary Ensemble protocol stacks which proceeds as follows (see Figure 5).

Given the names of the layers in the protocol stack, the system consults the a priori optimizations of these layers and composes them into a bypass. The individual CCPs and state updates are instantiated and composed by conjunction. This is done separately for each of the four fundamental cases (point-to-point send and receive, as well as broadcast send and receive events). *Header compression*, described below, is integrated as well. Afterwards the four cases are joined into a single program that uses the CCP as switch between the bypass code and the normal stack. This program is then exported from Nuprl to the OCaml programming environment to be compiled and executed.

To ensure that the generated optimized code is semantically equal to the original protocol stack in *all* cases, these steps are performed in the framework of formal optimization and composition theorems.

An *optimization theorem* proves that, under a given CCP, a piece of bypass code is semantically equal to the Ensemble protocol stack from which it was generated. For individual protocol layers (*i.e.*, 1-layer stacks) these theorems are created and proven automatically from the a priori optimizations stored in Nuprl's library. A bypass path through Ensemble's *Bottom* layer, for instance, is described by the following optimization theorem.

```
OPTIMIZING LAYER Bottom
 FOR EVENT DnM(ev, hdr)
 AND STATE s_bottom
 ASSUMING getType ev = ESend ∧
            s_bottom.enabled
YIELDS EVENTS [:DnM(ev, Full_nohdr(hdr)):]
 AND STATE s_bottom
```

This theorem formally states that, under the assumption that the layer is *enabled*, a down-going *send*-event

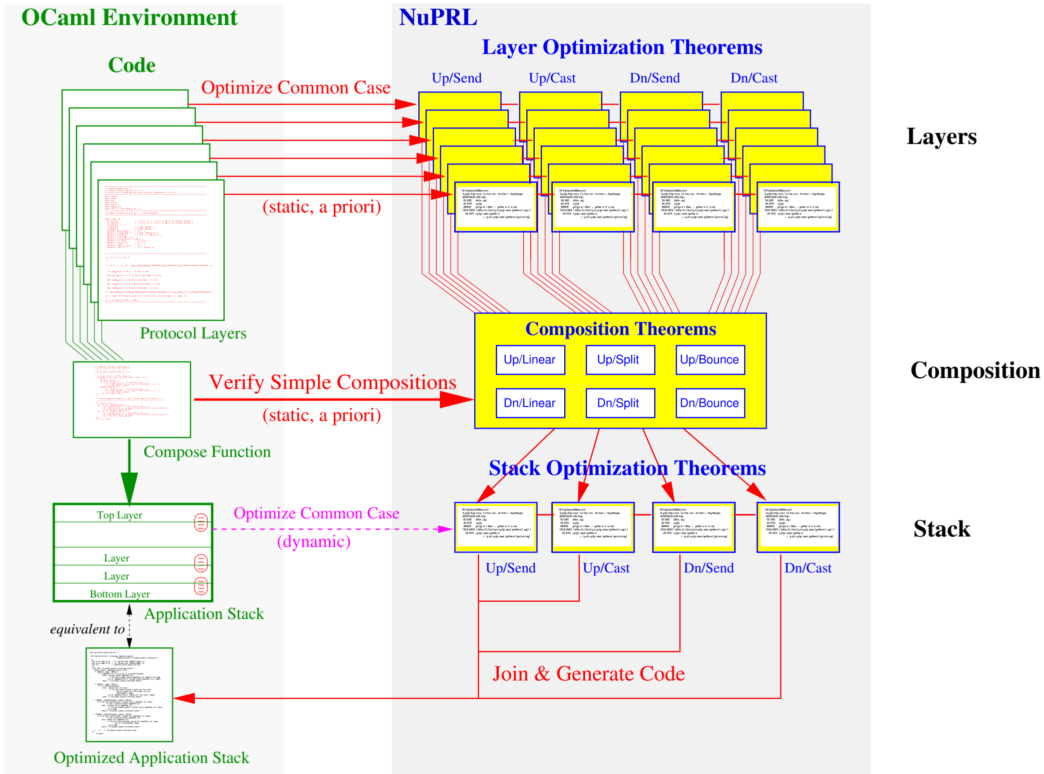


Figure 5. Optimization methodology: composing optimization theorems.

does not change the state `s_bottom` and is passed down to the next layer, with its header `hdr` extended to `Full_nohdr(hdr)`.

Optimization theorems for larger stacks are created from those for single layers by applying formal *composition theorems*. These theorems describe abstractly the effect of applying Ensemble's composition mechanism to common combinations of bypass paths, such as *linear traces* (events passes straight through a layer), *bouncing events* (events that generate callback events), and *trace splitting* (message events that cause several events to be emitted from a layer) — both for up- and down-going events ([10], Chapter 5).

```
     OPTIMIZING LAYER Upper
       FOR EVENT DnM(ev, hdr)
       AND STATE s_up
     YIELDS EVENTS [:DnM(ev, hdr1):]
        AND STATE s1_up
∧    OPTIMIZING LAYER Lower
       FOR EVENT DnM(ev, hdr1)
       AND STATE s_low
     YIELDS EVENTS [:DnM(ev, hdr2):]
       AND STATE s1_low
⇒    OPTIMIZING LAYER Upper ||| Lower
       FOR EVENT DnM(ev, hdr)
       AND STATE (s_up, s_low)
     YIELDS EVENTS [:DnM(ev, hdr2):]
       AND STATE (s1_up, s1_low)
```

The formal theorem above, for instance, expresses the obvious effect of composing down-going linear bypass paths: if an event passes straight down through the upper layer and then through the lower one, then it passes straight through the composed layers (`Upper ||| Lower`) as well. The state of the combined layer is updated according to the individual updates.

While the statement of such a composition theorem is almost trivial, its formal proof is quite complex, because the code of Ensemble's layer composition function uses a general recursion. By analyzing this code abstractly, and providing the result of the analysis as a formal theorem, we have lifted the optimization process to a higher conceptual level: we can now reason about composition as such instead of having to go into the details of the code of each layer and of the composition mechanism. It also speeds up the optimization process significantly: optimizing composed protocol layers is now a *single* reasoning step, while thousands of simplification steps would have to be applied to the code of the entire stack to achieve the same result.

Like the optimization theorems for individual protocol layers, all composition theorems are provided a priori as part of the optimization tool, as they do not depend on a particular application stack. As a result, the optimization theorems for a protocol stack can be generated automatically. To create the statement of such a theorem, we consult the theorems about layer optimizations for the corresponding events and compose them as described by the composition theorems. To prove the theorem, we first instantiate the optimization theorems of the layers in the stack with the actual event that will enter them. We then apply, step-by-step, the appropriate composition theorems to compose the bypass through the stack.

The optimization theorems do not only describe a bypass through a protocol stack but also provide the means for an additional optimization that cannot be achieved by partial evaluation or related techniques. They tell us exactly which headers are added to a typical data message by the sender's stack and how the receiver's stack processes these headers in the respective layers. As most of the header fields are fixed (constant) now, we only have to transmit the header fields that may vary.

For this purpose, we generate code for *compressing* and *uncompressing* headers, and wrap the protocol stack with these two functions. Both functions are generated automatically by considering the free variables of the events in the optimization theorems. We then optimize the code of the wrapped protocol stack using the same methodology as above. We have provided generic *compression* and *uncompression* theorems, which describe the outcome of optimizing a wrapped stack relatively to results of optimizing a regular stack, and use them to convert the optimization theorems for the regular stack into optimization theorems for the wrapped stack. All these steps are completely automated.

It should be noted that integrating compression into the optimization process will always lead to an improvement in the common case, because the optimized code will directly generate or analyze events with compressed headers, instead of first creating a full header and then compressing it. Only for the non-common case there will be a slight overhead due to compression.

All the above steps describe logical operations within the Nuprl system. In a final step, their results are converted into OCaml code that can be compiled and linked to the rest of the communication system. To generate this code, we compose the code fragments from the four optimization theorems into a single program, using the CCPs as conditionals that select either one of the bypass paths or the normal stack. This program is proven to be equivalent to the original stack in all cases, but generally more efficient in common cases (introducing a slight overhead in the exceptional cases).

There may be multiple bypass paths for each layer in a stack, resulting in many possible bypasses for the entire stack. We hope to elaborate this technique into one that would allow us to detect common combinations at run-time, and generate the optimized code dynamically, using layer optimization theorems for all possible bypass paths. We can then make use of Ensemble's support for dynamically loading layers and switching protocol stacks on the fly [25].

4.2 Performance results

We will present and compare performance results for four different versions of Ensemble stacks:

1. (*Imperative* or IMP) This is the normal version. In this case, Ensemble has a central event scheduler. It instantiates each protocol layer individually, and hands events to the layers as they come out of the scheduler. In the common case that no other events are queued in the scheduler, an output event is directly passed to the appropriate layer. Otherwise, output events are enqueued back into the scheduler.

2. (*Functional* or FUNC) In this case no centralized event scheduler is used. Composition is done based on the following observation: When two protocols are stacked on top of each other, the result is a new protocol. When stacking p on top of q, one applies events going down to p, and up events going up to q. The *down events* that come out of p are applied to q, and the *up events* that come out of q are applied to p, recursively. The up events that come out of p and the down events that come out of q are merged together to form the output events. The state of the composition of p and q is the combined states of p and q. An entire stack can be composed one layer at a time this way.

3. (*Hand-Optimized* or HAND) For particular common protocol stacks, Ensemble provides carefully optimized *bypass code* for common paths through the protocol stack. These paths were created manually. The stacks are built in the same way as in the functional version, but just before events are given to the stack a condition is checked to see if the event can be handled by the bypass. The bypass can access the state of the various layers in the stack.

4. (*Machine-Optimized* or MACH) Code generated from the functional code using the techniques of the previous section. The resulting code is used as a bypass much like in the hand-optimized case. Generating code from the imperative version is much harder for Nuprl, because the imperative code is harder to formalize than the functional code, and should not result in any benefit.

(None of these versions leverage concurrency. Ensemble does supports running each layer in its own thread, but in practice the context switch and synchronization overheads are very large.)

It is important to note that we are optimizing already heavily optimized code. That is, the first two optimization steps, avoiding garbage collection and marshaling, have been applied to all four versions.

When an application sends a message, it first travels down the protocol stack to the Ensemble *Transport* module (see Figure 4 in Section 4.1). The Transport module marshals the message into a sequence of bytes before it is sent out onto the network. At the receiver, the message is passed through the Transport module, which unmarshals the message. The message then travels up the protocol stack, and is finally delivered to the application.

In case of the hand-optimized version (HAND), the message goes through the bypass code instead of the protocol stack and the Transport module. In case of the machine-optimized code (MACH), there is only a bypass for the protocol stack, not for Transport. In both cases, the CCPs need to be checked first.

In our experiments, the CCPs specify that messages are delivered in FIFO order, are not fragmented, and no failure

	MACH	IMP	FUNC
Down Stack	9	20	42
Down Transport	8	27	30
Up Transport	7	20	22
Up Stack	8	14	38
Total	32	81	132

(a)

	HAND	MACH	IMP	FUNC
Down Stack	2	2	13	14
Down Transport	4	6	4	6
Up Transport	6	7	8	9
Up Stack	2	4	10	13
Total	14	19	35	42

(b)

Table 1. (a) 10-layer stack code latency in μs, for 3 different configurations using 4 byte messages. (b) 4-layer stack code latency in μs, for 4 different configurations using 4 byte messages.

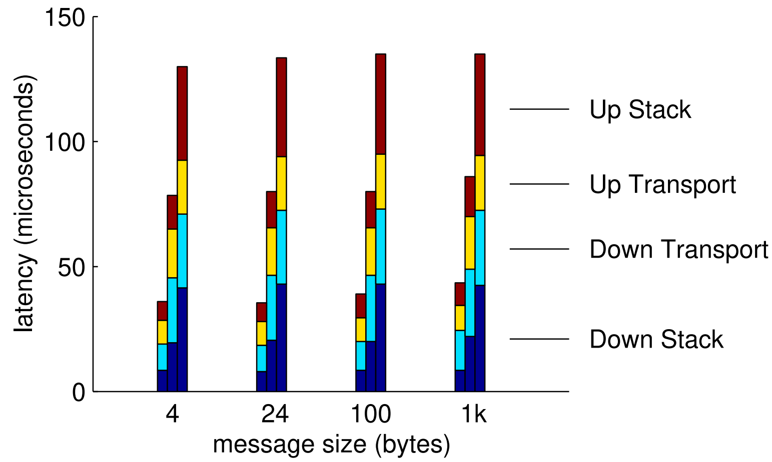


Figure 6. Code latency (10-layer protocol stack) comparison with different sized messages for the MACH, IMP, and FUNC configurations.

or other membership events occur. In practice, these circumstances are very common, and in the measurements below the outcome of the CCP checks is always the choice to run the bypass code. In our case, checking the CCPs takes only about 3 μs. Thus, even with CCPs for situations that are less common, a significant performance improvement may be expected.

We ran our measurements on two UltraSparc stations (300MHz) running Solaris 2.6, connected by a 100 Mbit Ethernet. For the latency measurements, we used Solaris' `gettimeofday()`. We ran each test 10,000 times and calculated the average. Since our experiments only measure code latencies, and do not require system calls, thread switches, or network communication, the variance in the reported numbers is negligible. We used the OCaml 2.0 native code compiler that produces good quality optimized machine code. We chose two different Ensemble stacks for our experiments. Both stacks provide reliable virtually synchronous delivery of multicast and point-to-point messages. The first stack uses 10 protocol layers and provides, additionally, total order, flow control, and fragmentation/reassembly. For comparison with the hand-optimized code we used a much simpler 4-layer stack, due to the difficulty of hand-optimization.

The hand-optimized code contains an optimization for the case when a process that is delivered a message immediately sends another message (*e.g.*, a response). If the first message is delivered through the bypass code, it assumes that the next message can be sent through the bypass as well, without checking the CCPs. For many cases this is true, and leads to a significant performance improvement. However, it may not be a correct assumption (*e.g.*, the response may need to be fragmented), and therefore the optimization cannot be generally substituted for the original code). In order to compare fairly between the hand-optimized code and the machine-generated code for the 4-layer case, we have added the same optimization procedure to the machine-generated code. (This could have been done by Nuprl itself, but we did it by hand.)

In Table 1(a), we show the code latencies involved in both the 10-layer protocol stack and the Transport module, for both the up and down event handlers. The latencies are for 4 byte messages. The savings in the Transport module are mostly due to header compression, resulting in less overhead in marshaling of headers. Most of the savings in the Stack processing is due to inlining of the common code path. In Table 1(b), we show the code latencies for the 4-layer stack. The performance improvements are naturally more substantial for larger stacks, but even for the small 4-layer stack a better than two-fold improvement is obtained.

Events	*Original Stack*	*Optimized Stack*
data_mem_refs	86293122	50905331
ifu_ifetch	172272565	100082695
ifu_ifetch_miss	3335271	1631051
itlb_miss	587083	361307
l2_ifetch	11075483	5525973
inst_decoder	182715118	98031212
ifu_mem_stall	143921523	76086051
cpu_clk_unhalted	348157540	199632585

(a)

10-layer stack		
Layer	*Down*	*Up*
partial_appl	684	392
total	820	1348
local	1388	1420
collect	44	2685
frag	900	144
pt2ptw	932	4899
mflow	596	572
pt2pt	268	1040
mnak	152	428
bottom	300	5052
total size	6084	17980
MACH (from 10 layers)	3592	2108

(b)

Table 2. (a) Data collected from Pentium performance monitoring counters for 10,000 send/recv rounds. (b) Object code sizes (in bytes) of the protocol layers and the generated code. The sizes have been separated for code dealing with down going events (such as Send), and upgoing events (such as Deliver).

These numbers do *not* include the network latency, which is about 80 μs in this case (using Ethernet). For the 10-layer stack, this means a total improvement of protocol processing in end-to-end communication from 50% to 29%, while for the 4-layer stack the improvement is from 30% down to 19%.

The performance improvements are more substantial for low latency networks than for high latency ones. For our Ethernet, using the 10-layer stack, the end-to-end latency improvement due to our optimization is 30%. On VIA (with 10 μs link latency), this would be 54%. For the 4-layer stack, the improvement is 14% on Ethernet, while 36% on VIA.

The hand-optimized code performs 25% better than the machine generated code. We believe that this difference can be attributed to having integrated the Transport module into the hand-optimized code.

In Figure 6, we show the same processing overheads of the 10-layer stack for different message sizes, 4, 24, 100, and 1024 bytes. For each size, there are three bars: from left to right MACH, IMP, and FUNC. In each bar, from top to bottom, we show the overheads of going down the protocol stack, down the transport, up the transport, and up the protocol stack. As may be observed, these processing overheads are mostly independent of message size. This is because we avoid copying by making use of the scatter-gather interfaces to socket communication.

To get more insight of micro-performance characteristics, we did some measurements using the performance monitoring counters on a Pentium II machine [12] running Linux 2.2.12. The machine we used has 32KB internal cache (16/16KB instructional/data cache), 512KB L2 cache and 128MB RAM.

In Table 2(a), we show some output from the performance monitoring counters. The result is for 10,000 rounds of message exchanges. In particular, we found that in a send/receive loop, the number of CPU cycles was reduced from 34816 to 19963, and the number of TLB misses from an average of 59 down to 36.

We believe most of this is due to partial evaluation and inlining of code. To show what effect this had on code size, we show, in Table 2(b), the object code sizes of both the up event handler and the down event handler for each protocol layer, as well as the size of the generated bypass handlers for a 10-layer stack.

5 Related work

The CMU Fox project [3] also uses an ML dialect, an extension of Standard ML, for building protocol stacks. In contrast to our project, Fox has built standard TCP/IP protocol software and a web server on top of that. The broader goal of the Fox project is to investigate the extent to which high level languages like ML, and particularly their support for modularity, are suitable for systems programming.

Integrated Layer Processing (ILP) [6] is an approach to reduce the overhead of layered protocols [1, 4]. By combining the packet manipulation of each layer, the total amount of memory accesses are reduced, and data locality is improved. The Filter Fusion Compiler (FFC) [22] implements ILP using partial evaluation. FFC has only been applied to very simple protocols, and the code generated by FFC has to be hand-modified to get good performance. ILP is most appropriate for protocols that do many "data-touching" operations (checksum computation, encryption, etc.) on large data packets. In our setting, many packets are quite small, and a large amount of protocol latency is introduced by protocol abstraction boundaries.

Our work presents a technique for optimizing mainly non-data touching operations, similar to *path-inlining* [21]. Path-inlining turns out to be difficult because of message ordering constraints, and it is out of the reach of traditional compiler optimization techniques because of the need for path constraints that cross component boundaries. Formal tools *are* able to analyze global properties, and we use them for synthesizing common code paths.

Other work has applied formal tools. Esterel [5] is an imperative, synchronous protocol specification language. The Esterel compiler is used to convert the protocol specification into a sequential finite automaton, from which efficient C code is generated. Esterel was used to specify a large subset of the TCP protocol, and to generate an implementation, but a general problem with this approach is that it does not scale easily to arbitrary protocol stacks.

Finding common execution paths is not always trivial. *Packet classifiers* help analyzing common paths in communication systems [2, 9, 24]. In operating system research there is related work on locating and optimizing common paths. Synthesis [19], which influenced systems as Scout, Aegis, SPIN and Synthetix, uses a run-time code generator to optimize the most frequently used kernel routines. [23] describes work on optimizing Synthetix kernel functions by reducing the length of common paths.

6 Conclusion

This paper described how correct protocol stacks can be configured from Ensemble's micro-protocol components, and how the performance of the resulting system can be improved significantly using the formal tool Nuprl. Although we chose a limited domain — all components are protocol layers, and all configurations are stacks of these — we believe that our approach could generalize and scale to more general configuration and component types.

Even for the limited domain, we feel that the following ingredients were necessary to make our project a success:

1. Using small and simple components. Smaller components are easier to reason about. On the other hand, components should be large enough so that their configurations do not become overly complex.

2. Using an event-driven, mostly functional implementation of components. Imperative implementations are much less suitable to formal analysis and transformation.

3. Using a well-defined configuration operation on components. The semantics of a configuration operation on components has to have a formal meaning. (In our case, a configuration of protocol components creates a new protocol component.)

4. Using a language with a formal semantics. Languages which do not have a formal semantics (*e.g.*, C) do not lend themselves to formal manipulation. (There is a rich subset of Java that has a formal semantics.)

5. Using I/O automata as a specification language. IOA have the look and feel of ordinary computer programs, and can be easily understood by programmers who have no background in computer science theory.

6. Using a formal tool. Hand-checking and hand-optimizing even only a handful of components is an arduous task. Our approach will only scale with appropriate tools such as Nuprl.

7. Using in-house expertise from both systems and formal groups. Neither the Ensemble nor the Nuprl group alone could have accomplished what we have.

It should be noted that Ensemble was created as a reference implementation of Horus [26], which was written in C. That is, Ensemble was designed to be verified, and there was initially no plan to retire Horus. We consider it infeasible to verify a complex system such as Horus which was not designed for verification. Because of the successful optimization approach in Ensemble (Ensemble is now generally as fast or even faster than Horus), the development of Horus was stopped. We have not yet been able to generate a machine proof of a non-trivial group communication protocol, but believe that we will complete a proof of one of Ensemble's total ordering protocols shortly.

We believe that it may be possible to use our approach in other complex systems such as file systems, atomic transaction protocols, and memory paging hierarchies, and perhaps eventually an entire operating system kernel. Because of its logical foundation and the high level of abstraction, our optimization techniques are relatively independent from the Ensemble toolkit. The methodology described here can be applied to other modular systems whose modules and composition mechanisms can be described semantically. As for correctness proofs, the THE operating system kernel [8] was built from layered components and proven correct, by hand, in 1968. (THE was also designed with verification in mind.) We believe it is crucial that the system be built from components with well-specified behaviors, and that the components are programmed in a language with a formal semantics.

In our experience, getting programmers to write specifications is a hard task. (Note that this does not hinder optimization, which is now a push-button technology that does not require specifications other than the code itself.) We still have a long way to go with writing all the abstract specifications for the Ensemble layers. The prospect of writing over 60 such specifications is daunting. Fortunately, many layers have similar specifications. Compare, for example, Figures 2(a) and (b). We are developing an object-oriented technique of extending existing specifications. For example, FifoNetwork may be generated from LossyNetwork by adding some state and a few conditions and actions.

See http://www.cs.cornell.edu/Info/Projects/NuPrl and http://www.cs.cornell.edu/Info/Projects/Ensemble for more information.

Acknowledgments

We would like to thank Mark Bickford, Alan Fekete, and Nancy Lynch for very helpful discussions. We also want to thank the anonymous SOSP reviewers, and our "shepherd" Peter Lee for numerous improvements made to our paper.

References

[1] ABBOTT, M., AND PETERSON, L. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610 (Oct. 1993).

[2] BAILEY, M., GOPAL, B., PAGELS, M., PETERSON, L., AND SARKAR, P. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementations* (Apr. 1994), pp. 115–123.

[3] BIAGIONI, E. A structured TCP in Standard ML. In *Proceedings of the '94 ACM SIGCOMM Conference* (London, UK, Aug. 1994), pp. 36–45.

[4] BRAUN, T., AND DIOT, C. Protocol implementation using integrated layer processing. In *Proceedings of the ACM SIGCOMM Conference* (1995), pp. 151–161.

[5] CASTELLUCCIA, C., AND DABBOUS, W. Generating efficient protocol code from an abstract specification. In *Proceedings of the ACM SIGCOMM Conference* (New York, 1996), pp. 60–72.

[6] CLARK, D., AND TENNENHOUSE, D. Architectural consideration for a new generation of protocols. In *Proceedings of the ACM SIGCOMM Conference* (Sept. 1990), pp. 200–208.

[7] CONSTABLE, R., ALLEN, S., BROMLEY, H., CLEAVELAND, W., CREMER, J. F., HARPER, R., HOWE, D., KNOBLOCK, T., MENDLER, N., PANANGADEN, P., SASAKI, J., AND SMITH, S. *Implementing Mathematics with the NuPRL proof development system*. Prentice Hall, 1986.

[8] DIJKSTRA, E. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346 (May 1968).

[9] ENGLER, D., AND KAASHOEK, M. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the ACM SIGCOMM Conference* (1996), pp. 53–59.

[10] HAYDEN, M. *The Ensemble System*. PhD thesis, Cornell University, Jan. 1998.

[11] HICKEY, J., LYNCH, N., AND VAN RENESSE, R. Specifications and proofs for Ensemble layers. In *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (1999), R. Cleaveland, Ed., no. 1579 in Lecture Notes on Computer Science, Springer-Verlag, pp. 119–133.

[12] INTEL CORPORATION, SANTA CLARA, CA, USA. *Intel Architecture Software Developer's Manual*, 1999. http://developers.intel.com/design/pentiumii/manuals, order numbers 243190, 243191, 243192.

[13] KAY, J., AND PASQUALE, J. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM Conference* (Sept. 1993), pp. 259–268.

[14] KREITZ, C. Formal reasoning about communication systems I: Embedding ML into type theory. Tech. Rep. TR97-1637, Cornell University, June 1997.

[15] KREITZ, C. Automated fast-track reconfiguration of group communication systems. In *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (1999), R. Cleaveland, Ed., no. 1579 in Lecture Notes on Computer Science, Springer-Verlag, pp. 104–118.

[16] KREITZ, C., HAYDEN, M., AND HICKEY, J. A proof environment for the development of group communication systems. In *15^{th} International Conference on Automated Deduction* (1998), C. . H. Kirchner, Ed., no. 1421 in Lecture Notes in Artificial Intelligence, Springer-Verlag, pp. 317–332.

[17] LEROY, X. *The OCAML System*. INRIA, Rocquencourt, France, 1998. http://pauillac.inria.fr/ocaml.

[18] LYNCH, N. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1987.

[19] MASSALIN, H. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Computer Science Department, Columbia University, 1992.

[20] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[21] MOSBERGER, D., PETERSON, L., BRIDGES, P., AND O'MALLEY, S. Analysis of techniques to improve protocol processing latency. In *Proceedings of the ACM SIGCOMM Conference* (New York, 1996), pp. 73–84.

[22] PROEBSTING, T., AND WATTERSON, S. Filter fusion. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, Jan. 1996), pp. 119–130.

[23] PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. Optimistic incremental specialization. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Dec. 1995), pp. 314–321.

[24] VAN RENESSE, R. Masking the overhead of protocol layering. In *Proceedings of the ACM SIGCOMM Conference* (Aug. 1996), pp. 96–104.

[25] VAN RENESSE, R., BIRMAN, K., HAYDEN, M., VAYSBURD, A., AND KARR, D. Building adaptive systems using Ensemble. *Software—Practice and Experience*, 28(9):963–979 (July 1998).

[26] VAN RENESSE, R., BIRMAN, K., AND MAFFEIS, S. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83 (Apr. 1996).

[27] VON EICKEN, T., AND VOGELS, W. Evolution of the Virtual Interface Architecture. *IEEE Computer*, 31(11):61–68 (Nov. 1998).