Generating Efficient Protocol Code from an Abstract Specification

Claude Castelluccia, Walid Dabbous
INRIA, 2004, route des Lucioles - BP 93, 06902 Sophia Antipolis, France
{claude.castelluccia, walid.dabbous}@inria.fr
Sean O'Malley
Network Appliance, 319 N. Bernardo, Moutain View, CA 94043, USA
sean@netapp.com

Abstract

A protocol compiler takes as input an abstract specification of a protocol and generates an implementation of that protocol. Protocol compilers usually produce inefficient code both in terms of code speed and code size. In this paper, we show that the combination of two techniques makes it possible to build protocol compilers that generate efficient code. These techniques are i) the use of a compiler that generates from the specification a unique tree-shaped automaton (rather than multiple independent automata), and ii) the use of optimization techniques applied at the automaton level, i.e. on the branches of the trees.

We have developed a protocol compiler that uses both these techniques. The compiler takes as input a protocol specification written in the synchronous language Esterel. The specification is compiled into a unique automaton by the Esterel front end compiler. The automaton is then optimized and converted into `C` code by our protocol optimizer called HIPPCO. HIPPCO improves code performance[1] and reduces code size by simultaneously optimizing the performance of the common path and optimizing the size of the uncommon path. We evaluate the gain expected with our approach on a real-life example, namely a working subset of the TCP protocol generated from an Esterel specification. We compare the protocol code generated with our approach to that derived from the standard BSD TCP implementation. The results are very encouraging. HIPPCO-generated code executes up to 25 % fewer instructions than the BSD code for input packet processing while maintaining comparable code size.

1 Introduction

The development of a compiler that takes as input an abstract (and hence theoretically tractable) protocol definition and produces an implementation of that protocol has been the subject of much research in the networking community [24, 28, 20, 32]. Unfortunately the compilers to date have produced code too slow to be used in production protocols. The goal of the work presented in this paper is to develop a protocol compiler which takes as input an abstract protocol definition and produces code that is good enough to be used in production applications. More specifically the goal of our protocol compiler is to produce protocol code that:

- is as fast or faster than the manually optimized implementations of that protocol
- is as small or smaller than the manually optimized implementations of that protocol
- inter-operates with existing versions of standard protocols

The reasons for these goals should be obvious: until the code produced by a protocol compiler equals or surpasses existing code, protocol compilers are unlikely to be used.

Our approach to achieving these goals is to take an existing protocol compiler and to develop a new code generator. This code generator uses a variety of standard and new optimization techniques and applies them systematically to the `C` code generated by the protocol compiler. Our work, which is presented in this paper, has concentrated on the generation of small and efficient control flow code. The results are quite promising. Using the above techniques, we have produced a protocol compiler which compiles a subset of the TCP protocol into code which is:

- 25 % faster than a similarly restricted version of the BSD TCP implementation.
- only 25 % larger than a similarly restricted version of the BSD TCP implementation.

Our basic conclusion is that there is no intrinsic performance penalty incurred when compiling from a high level protocol description. The poor performance of existing compilers can be explained by the lack of adequate code generator back ends or the use of poorly designed protocol execution environments. The ability to apply optimizations in a systematic fashion yields results superior to those obtained with manual methods both in terms of the performance and the quality of the resulting code. We hope that this work generates renewed interest in and raises the performance standards of protocol compilers, since there is no good reason why generated protocol code should not be efficient.

[1] Throughout this paper, we use code performance to mean code execution speed performance.

2 Related Work

The work presented in this paper relates to the fields of automatic protocol code generation, protocol optimization and compiler optimization. Our primary claim to novelty is combining these existing techniques in a protocol compiler that generates efficient code.

A significant amount of work has been done on the generation of protocol implementation from standard specification languages such as SDL, LOTOS or ESTELLE [28, 20, 32, 24]. However, the performance of the code generated by these tools is usually quite poor. The primary reason for this is that the generated code is derived directly from the specification in the sense that each module of the specification is implemented by a process. This type of protocol implementation incurs the cost of a context switch and a data copy at each module interface. In general, several module interfaces might be traversed to process a single packet. This introduces significant overhead. This type of protocol implementation also inhibits compiler optimization by reducing the size of the basic blocks. Most of these protocol compilers simply do not perform any form of optimization. In contrast, we are using a synchronous approach, the specification is compiled into a single integrated automaton, and we optimize the generated code.

Most of the protocol optimizations implemented in our compiler are well known. For example, the use of inlining, outlining and code cloning to improve the performance of protocol code is thoroughly described in [27]. One of our contributions is to apply these optimizations in the context of a protocol compiler and to automate the application of these techniques.

In the field of compiler optimizations, the problem of restructuring programs to improve program performance has been studied by numerous researchers. In [25, 11, 29], profiling data is used to guide the positioning of basic blocks and functions in order to increase the performance of the instruction cache. The optimizations in this paper are directly inspired by this work. However our approach differs in several aspects. First, the tree structure of our code, without loop and jump, considerably simplifies the analysis of the code and the design of the optimizations. As shown in [25, 11, 29], the graph structure of traditional programs requires manipulation of the program basic blocks to optimize the performance of the code. With our tree structure, the same results can be achieved at a higher level just by moving around branches and subtrees. Second, our optimizer is specialized to protocol code and to the automaton structure of our particular environment. For example, the clear separation that exists between the common path and the uncommon path in protocol code simplifies both the analysis phase and the application of the optimizations.

The rest of our paper is structured as follows. Our protocol development environment is described in section 3. Section 4 and 5 detail the performance and code size optimizations applied by HIPPCO. To validate our approach, we performed a set of experiments. These experiments are described in section 6. Section 7 analyzes the results. Section 8 concludes the paper.

3 An Automated Protocol Development Environment

In this section we describe the general architecture of our protocol compiler development environment. This environment has three parts, namely the specification language, the compiler for that language and the run time system. The source language in our development environment is the Esterel language [4, 2]. Esterel is a synchronous language that supports the abstract specification of protocol behaviors [6, 3]. The compiler for that language is composed of the combination of the Esterel front end compiler and of the HIPPCO tool. The Esterel front end compiles the modular specifications into sequential and minimal automata. These automata are then optimized and converted into efficient `C` implementations by HIPPCO[2] [9], the protocol code optimizer we have developed. `C` is used as the target language for HIPPCO for portability reasons. The overall architecture of this environment is presented in figure 1 and detailed in the remainder of this section.

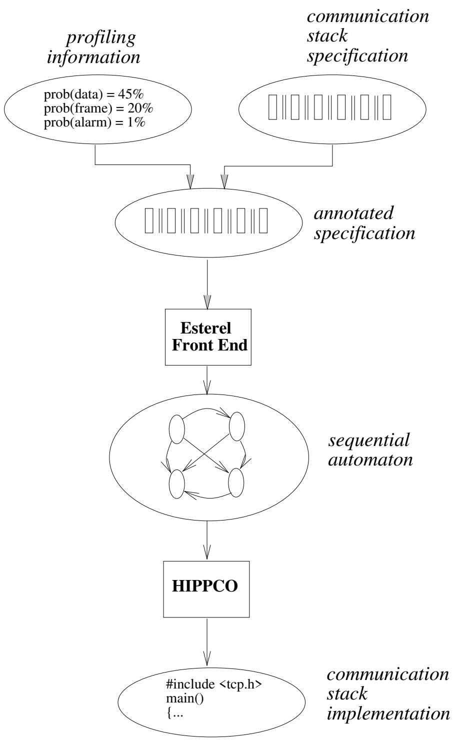


Figure 1: The Protocol Development Environment

3.1 The Esterel Tool

We have used Esterel as the specification language and automaton generator for our development environment. Esterel is an imperative *synchronous* language developed at INRIA [4, 1]. In this subsection, we justify this choice and present the main characteristics of this language.

Most languages and code generators currently used for protocol specification and design, such as SDL or Estelle, are asynchronous [20, 32, 23, 24], meaning that modules are specified and implemented as separate processes communicating via asynchronous queues. We believe that the synchronous approach is more attractive for the generation of high performance implementations. Synchronous languages compile a set of parallel or sequential specification modules into a single automaton using the data dependencies, the control dependencies and the language semantics. This allows the automaton to be implemented in a single process. This approach leads to more efficient implementa-

[2] HIPPCO stands for HIgh Performance Protocol Code Optimizer.

tions because it minimizes the number of context switches when compared to the naive "one process per layer" architecture. Furthermore, this architecture increases the potential of compiler optimizations by eliminating all traces of module boundaries and hence enabling nearly unlimited inter-procedural optimization [26, 15].

Synchronous languages are used to implement the control portion of a program. The computational and data manipulation parts are performed by functions implemented in another language (`C` for example). Data declarations are encapsulated, so that only the visible interface declarations are provided in Esterel (i.e. type, constant, function and procedure names). These declarations can then be freely implemented independently of the Esterel program design. They will be linked with the automaton generated in the last phase when executable code is produced.

Esterel makes an assumption of perfect synchrony meaning that program reactions can not overlap. There is no possibility of activating a system while it is still reacting to the current activation. This assumption makes Esterel programs deterministic, since their behaviors are reproducible; the generated automaton can then be tested for correctness using validation tools [30].

The Esterel Specification Language

Esterel programs are composed of parallel modules, which communicate and synchronize using signals. The output signal of a module is broadcast within the whole program and can be tested for presence and value by any other module. This communication mechanism provides a lot of design flexibility, because modules can be added, removed or exchanged without perturbing the overall system. A module is defined by its inputs (the signals that activate it, they can potentially be modified by the module), sensors (input signals used only for consultation, they can not be modified) and outputs (signals emitted). The inputs of a module can either be the outputs of another one (modules executed sequentially) or external signals (such as incoming packets). The design of an Esterel program is then performed by combining and synchronizing the different elementary modules using their input, sensor and output signals.

The Esterel Compiler

At compile time, the Esterel program is translated into a sequential finite automaton (in a standard format called Oc) by an exhaustive exploration of the set of control states of the program; the code of the different modules is sequentialized according to the program concurrency and synchronization specifications. The correspondence between the specification code and the generated automaton is far from being obvious. A small change in the specification can involve a complete modification of the automaton.

Automata are often used to implement the control kernel of complex systems. Given a set of input values, the automaton selects a transition from its current state, calls the corresponding sequential tasks, and changes its state for its next reaction. A reaction is a *linear* piece of code (no loops, no recursion, no interrupts, and no overhead due to process management).

The generated automaton is usually composed of several *states*. Each of these states is described by a state-tree, where each node in the tree corresponds to some action or test in the protocol being implemented. A state-tree is composed of initialization actions followed by independent subtrees processing the different types of input events. A test at the root of each subtree determines whether the incoming event should be processed by this subtree. If that is the case, the backward branch (**then** branch) is executed, otherwise the subtree exits. The state-tree is generated by cascading the roots of those different subtrees as shown in figures 2, 3 and 5. An incoming event will then be processed by a sequence of tests until the appropriate subtree is found.

Trees facilitate dependency analysis and simplify optimization design and implementation. In fact the tree representation corresponds to a protocol implementation that has been fully inlined and cloned [27]. However, the tree representation generally leads to very large code size because many infrequently executed actions are duplicated in the different branches. A DAG (Directed Acyclic Graph) would lead to a smaller code size and certain HIPPCO code size optimizations to convert this tree representation to a DAG.

An example of an Esterel specification and of the derived automaton is presented in appendix A. This module originates from a prototype specification of a restricted version of the TCP protocol in Esterel [8]. It handles the management of the TCP retransmission timer.

Notations

We define in this subsection the notations that are used in figures 2, 3 and 5. A state-tree is composed of a collection of cascaded nodes. There are three types of nodes namely `decision_nodes`, `state_nodes` and `action_nodes`.

- A `decision_node` is composed of three elements: **`test`**, **`then`** and `else`. The **`test`** is the predicate to test, **`then`** is a pointer on the node to execute if the test is correct and `else` is a pointer on the node to execute if the test is uncorrect. In the figures of this paper, the backward branch of a decision node (the **`then`** branch) is its left branch. A number below a `decision_node` indicates the probability that its predicate is true. here are two types of `decision_nodes` namely `input_nodes` and `test_nodes`. The `input_nodes` test the presence of the different inputs that can activate the automaton. For a protocol, they might test whether the incoming event is a packet from the network, a data from the application or a signal from the clock. They are drawn as diamonds. The `test_nodes` represent the tests of the specification. They might test, for example, the checksum of a packet or the reception window size. The are drawn as circles.

- A `state_node` is composed of one element `nstate`, an integer that indicates the number of the next state the automaton is going to. The `state_nodes` are the leaves of the trees. State nodes are drawn as two brackets containing the value of the `nstate` variable (<`nstate`>).

- An `action_node` is a node that performs an action or a set of actions, such as variable assignments and/or function calls. They are drawn as rectangles. For simplification, some action nodes are omitted in figure 3.

In the figures, an arrow represents a pointer to a node (implemented as a function call) while a line represents a direct access (the node is inlined).

3.2 The HIPPCO Code Generator

HIPPCO is a new optimizing back end for Esterel. It takes as input the automaton generated by the Esterel front end

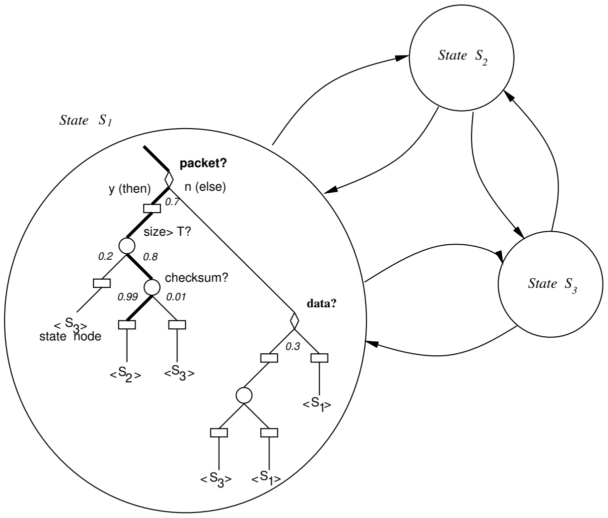


Figure 2: State tree

and generates an efficient C implementation of that automaton. The motivation for HIPPCO is two-fold. First, the quality of the code currently generated by the Esterel compiler is inappropriate to protocols. It is either interpreted, and therefore slow, or inlined, and therefore large (megabytes in the case of TCP). Second, the optimizations performed by most C compilers are somewhat limited. While C compilers perform many optimizations, these optimizations are usually local and based on static analysis of the code. This type of non-local and dynamically analyzed optimization is the key to improving protocol performance.

Finding the Common Path

Many protocol optimizations depend upon the identification of the most frequently executed sequence of basic blocks (the common path) in the protocol code. In fact, many optimizations which increase code speed also increase code size. Thus we do not want to perform these optimizations on code which is not used frequently. The common path is usually very small. For example, in the BSD version of TCP, the size of the common path of the input processing is 640 bytes long (on the DEC Alpha). This represents 8.1 % of the total size of the input processing code. The clean automaton representation allows a much more systematic approach to the identification of the common path through protocol code. HIPPCO divides the generated automata into two parts: the *common* path, and the *uncommon* path. In the terminology of automata, the common path of an automaton is defined by the x % transitions that use y % of the protocol execution time (where x is typically equal to 10 and y to 90). Therefore, the common path can be identified from the probabilities of the transitions [31]. In HIPPCO, these probabilities are computed by a Markov analysis [10] of the automaton.

The transitions of a state are defined by the different paths that an input can follow through the tree of this state. They are therefore defined by an input and the sequence of tests (and their results) that lead to the leaf node. In figure 2, the set of transitions of the state S_1 leading to the state S_2 is defined by the single element set $T^{1,2}$: $(packet, \neg(size > T), checksum)$, where $packet$ is the input firing the transition and the following elements are the results of the tests that compose this transition. The probability of this transition is then equal to the product of the following terms: the visit probability of state S_1, the occurrence probability of the firing input, and the probabilities of the visited branches. Thus:

$$P(T^{1,2}) = P(S_1) \times P(pack) \times P(\neg(size > T)) \times P(check))$$

In [10], we have shown that the visit counts V_j of the state S_j can be computed by solving the following system of n linear equations (where n is the number of states):

$$V_j = \delta_{1j} + \sum_{k=1}^{n} V_k \times P(T^{k,j})$$

where $\delta_{ij} = 1$ for $i = j$, and 0 otherwise. The visit probability of a state S_j is given by:

$$P(S_j) = \frac{V_j}{\sum_{k=1}^{N} V_k}$$

Therefore the transition probabilities can be computed from the probabilities of the occurrence of each external input and from the probabilities of the protocol test outcomes. To make it possible for the programmer to enter the necessary probabilities we implemented an extension to the Esterel compiler that allows specifying numerical probabilities for the input events of a specification, and for the two outcomes of an *if* statement in an Esterel specification.

In the example of the figure 2, the probability to receive an input of type *packet* (coming from the network) was set to 70 %, whereas the probability to receive an input of type *data* (coming from the application) was set to 30 %. Also the probability, that the test *(size>T)* is correct, was set to 20 % and the probability that the checksum is correct, was set to 99 %. This information is preserved through the compilation phase of Esterel and appear on the automaton description. It is then used by the optimizer HIPPCO to identify the different paths and to apply the optimizations selectively.

The main problem in code analysis is to determine these probabilities. They usually cannot be derived by low-level compilers because they strongly depend on the semantics of the protocols. They should either be derived from profiling data or be specified by the protocol designer, who should know the normal path of the protocol he is designing. In section 7, we will show the performance implications of these two approaches.

3.3 The HIPPCO Runtime System

All compilers require runtime systems to provide an interface to services provided by the host computer. For protocol compilers, the runtime system must provide standard interfaces to such operating system entities as buffers, timers, and hardware devices. The performance of the overall system is critically dependent upon the efficiency of the runtime system. Since the design of efficient protocol runtime systems is a well understood problem [22], we decided not to spend any effort improving the existing runtime system. Instead, we built a minimal (and not fully functional) runtime system into the test harness we used to test our code. This system is described in section 6.

4 HIPPCO Performance Optimizations

The primary goal of HIPPCO is to generate time efficient protocol codes. Increasing the execution speed of a protocol code is equivalent to decreasing the number of cycles required for its execution. This is a complex task because these cycles comes from different sources. The number of cycles required to execute the program is given by the following formula [19, 7]:

$$Cycles_{total} = IC \times CPI + m\_access \times miss\_rate \times miss\_pen$$

where IC is the number of instructions executed, CPI the average number of cycles per instruction, m_access the total number of memory accesses within the program, $miss\_rate$ the rate of memory references which are not in the cache and $miss\_pen$ the penalty, expressed in cycles, encountered when a memory access with a cache miss occurs. The total number of cycles is composed of the sum of the number of the cycles spent executing the instructions of the program and the cycles spent waiting for the memory system (memory stalls). In this evaluation, it is assumed that all memory stalls are due to the cache. Although this is not true for all machines, the stalls due to the cache always dominate the effect of other stall sources. We also consider, in this formula, that the cycles for cache accesses are part of the CPU-execution cycles and are therefore included in CPI.

Thus according to this formula, the optimization of a program execution time involves three distinct kinds of optimizations: those that reduce the instruction count, those that decrease the CPI, and those that reduce the miss rate. As the performance bottleneck has been shifted to the memory system [13, 18, 17], the second two forms of optimization have gained in importance.

HIPPCO uses those three types of optimizations. The effects of HIPPCO's optimizations on the tree structure and the memory layout are illustrated in figures 3 and 4 (the darker lines in figure 3 and the shaded parts in figure 4 represent the common path). As shown on those figures, HIPPCO's optimizations reduce the instruction count (the length of the common path is reduced) and increase the code locality.

The remainder of this section describes the optimizations used by HIPPCO to improve protocol performance.

4.1 Input Rescheduling

The number of instructions to process an input event in a state is the sum of two components: the cost to reach the subtree corresponding to the input and the cost to actually process the input event. This cost is minimal when the subtrees corresponding to the most frequent events are scheduled first in the tree, i.e. the subtrees are sorted in decreasing event probabilities.

The *Input Rescheduling* optimization (designated by the letter R) is based on this observation. It consists of scheduling the subtrees handling the most frequent input-events before the subtrees handling the less frequent input-events. Input events belonging to the common path would then be detected faster (IC is reduced). By grouping the most frequently executed subtrees, this optimization also increases the common path code locality. The input probabilities are obtained from the information provided by the protocol or the application designer in the specification. This optimization is illustrated in figure 3. The $subtree_D$ corresponding to the input with the highest occurrence probability (60 %) is scheduled at the top of the tree, followed by the subtrees processing $input_B$ (20 %), $input_C$ (10 %) and $input_A$ (10 %).

4.2 Extension of the Common Branches

In order to reduce the size of the generated code, the HIPPCO code size optimizations (described in section 5) transform the Esterel tree representation into a graph representation by sharing similar subtrees. These optimizations introduce indirections and therefore increase the number of the executed instructions. It is therefore not desirable to perform these code size optimizations on the common path.

The *Common Branch Extension* optimization (designated by E) undoes the effect of the code size optimizations on the common path. It transforms the graph representation into a tree representation along the common path. The generated common path is then streamlined. This optimization reduces the number of instructions on the common path (IC) by removing the function calls overhead. It also improves the instruction-cache behavior (i.e. reduces $miss\_rate$) by removing the indirection along the common path. This optimization is illustrated in figure 3. The $subtree_D$ which belongs to the common path (the dark line) is not shared.

The drawback of this optimization is that it increases the code size. However, since the common path constitutes only a small part of the total code, we expect this code size increase to be negligible, or at least small, compared to the size of the uncommon path.

4.3 Branch Pruning

In some protocol configurations, some inputs never occur and/or some test outcomes are constant. In this situation, the processing relative to these inputs and tests is unnecessary.

The *Branch Pruning* optimization (designated by P) "specializes" a general protocol specification to a particular application. This optimization is performed in two steps. First, all subtrees handling impossible events for a given protocol configuration are removed. Then, within a subtree all tests with predictable results are detected and replaced by their constant outcomes.

By removing the unnecessary tests, this optimization reduces the number of instructions to execute (IC). This optimization is performed on the initial tree representation, before any code size optimization is performed. It is performed everywhere (on the common and uncommon paths). This optimization is illustrated in figure 3. The test t_1 of $subtree_D$ is always correct. The test t_1 and its `else` branch have then been removed in the optimized tree.

4.4 Inlining

The *Inlining* optimization (designated by I) consists in replacing a function call by a copy of the function body. Function inlining decreases execution time for two reasons: first, the overhead of the function call disappears (IC decreases). Second, by removing the function boundary, more efficient optimizations can be performed by the low-level compiler. The penalty for inlining is a code size increase which may adversely effect the instruction cache hit rate and slow down program execution speed [14, 16, 27, 21]. If code size is not a direct constraint, inlining a function call is beneficial if the function is only called once, the size of the function is smaller or equal to the number of instructions required to call this function, and/or this function call is executed very often.

Traditional compilers generally generally functions satisfying the two first properties. Since they don't have enough information on the function call frequencies, they rarely take the third property into account. In HIPPCO, the fine analysis of the automaton allows a more efficient and simple application of this optimization: all function calls along the common path are automatically inlined.

4.5 Outlining

Reducing the cache footprint of the common path code is another common optimization. Outlining [27, 29] compacts frequently executed code and moves infrequently executed code out of the mainline of execution. The resulting code features a very spatially compact common path which is more likely to fit into the cache. As explained in [27], this optimization has its limits when it is performed manually. HIPPCO automatically identifies the common path and compacts it by coding all uncommon subtrees and branches along the common path as functions calls. The resulting code is then very compact.

This optimization is illustrated in the figure 3. In the optimized tree, the branch of $subtree_D$ leading to state S_1 along the common path is coded by a reference (function call).

4.6 Rescheduling of Test Outcomes

To increase program execution speeds, current processors frequently use an implementation technique called *pipelining*. This technique consists of executing micro-instructions concurrently. The low level compiler schedules the instructions to take as much advantage as possible of the pipelining. However, there are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated cycle. For example, when a branch instruction is executed, the following instruction can not be loaded before its address has been computed[3]. Therefore to reduce this penalty, most processors make predictions: they make the assumption for conditional branches that the forward branch (`else` branch) is not taken and continue to fetch instructions as if the backward branch (`then` branch) were a normal instruction. If the prediction turns out to be invalid, the pipeline is stopped and restarted with the fetch of the new instruction. The branch penalty is then suppressed for each valid prediction.

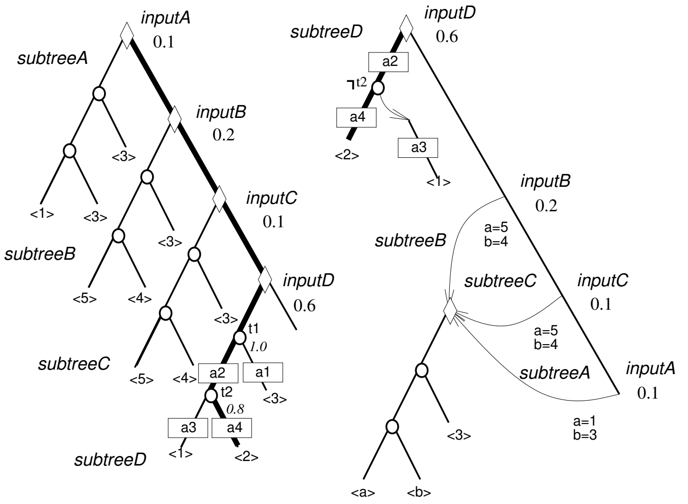
The Rescheduling of Test Outcomes optimization (designated by T) is based on this observation. It restructures the test nodes in the decision tree such that the most frequent outcomes are on the left branch of the tree[4] in conformity with the low-level compiler predictions. This transformation is performed by considering all the program trees nodes (on the common and uncommon path) and by reversing all the tests whose most frequent outcomes are on the forward (else) branch. Because we are using a tree representation, this optimization also increases the common path locality: the code is restructured such that the common path is completely separated from the uncommon path. This optimization is very similar to the one presented in [29], however the tree representation simplifies considerably its application and makes it practical.

This optimization is illustrated in figures 3 and 4. The test t_2 has a 20 % probability of being correct. It is then inverted (t_2 is replaced by $\neg t_2$ and its branches are inverted).

[3]It is interesting to note that a side effect of the *inlining* optimization presented in 4.4 is to improve pipelining.

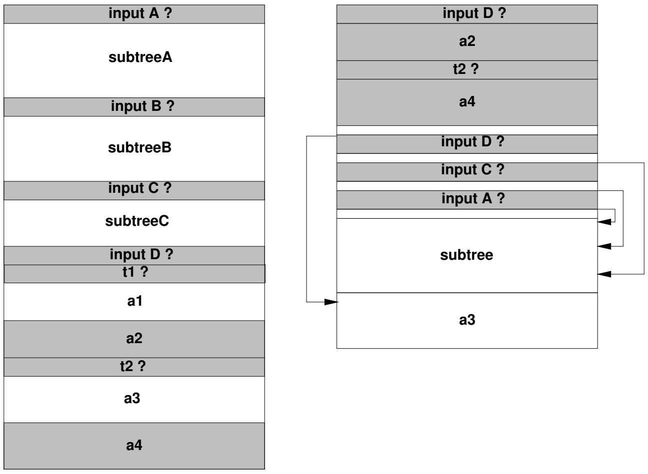
[4]If the processor predicts the forward branch, it is sufficient to reverse the effect of this optimization i.e., to schedule the most frequently outcomes on the forward branch.

This optimization has a double effect: it increase pipelining and the common path code locality.



(a) before optimization (b) after optimization

Figure 3: Code speed optimizations (tree representation)



(a) before optimization (b) after optimization

Figure 4: Code speed optimizations (cache layout)

5 HIPPCO Code Size Optimizations

The tree data structure used to represent the output of the front end leads to a significant increase in code size. In some respect, HIPPCO starts with code that is fully inlined. Thus, to reduce code size we must eliminate identical sub-trees instances in the automaton.

5.1 State Pruning

As detailed in 3.2, HIPPCO computes the visit probabilities of each state for the common path identification. In some configurations of the protocol, some states of the corresponding automaton are unreachable: they have a visit probability of zero. This is usually the result of external inputs being impossible (their occurence probability is null). In this case, HIPPCO removes the code of these unreachable states through the use of the *State Pruning* optimization (designed by G). This specialization results in a reduction of the protocol code size. The branch pruning optimization (described in 4.3) also reduces the size of the generated code through specialization of the code.

5.2 Sharing of Similar Subtrees

The most important code size optimization in Esterel transforms the tree representation in a directed acyclic graph (DAG) representation by sharing identical subtrees of the automaton. HIPPCO's first code size optimization (designated by S) is an extension of the Esterel optimization cited above. It consists in sharing *similar* subtrees. Two subtrees are said to be *similar* if, by eliminating some leaves, they become identical. This optimization is motivated by the observation that many subtrees differ only by their leaves. The leaves in the shared subtrees are then replaced by references to a so-called signature tree which gives the value of the shared leaves based on the results of previous tests identifying those leaves. Figure 5 illustrates this optimization. The three subtrees rooted at t_2 with the `then` branch corresponding to a_7 and the `else` branch to a_8 are shared in the optimized tree. The differing leaf (the one of the `then` branch) is identified by a signature which is based on test t_0. Note that the Esterel (O) optimization would have resulted in sharing only the two identical subtrees on the right bottom of the unoptimized tree as indicated by the dotted arrow.

5.3 Sharing of Intermediate Subtrees

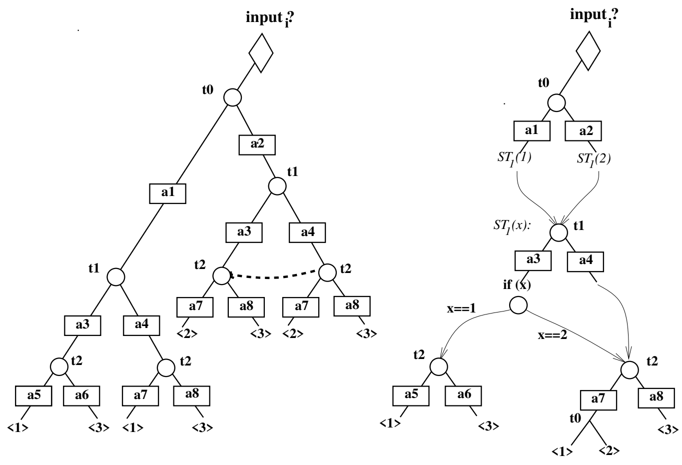
The previous optimization generates *intermediate* subtrees, whose leaves are pointers to others subtrees. Many of these subtrees only differ by their leaves. The optimization called *sharing of intermediate subtrees* (designated by N) codes these similar subtrees by an unique function with parameters indicating the differing leaves. This is illustrated in figure 5, where the intermediate subtree rooted at t_1 is shared. The resulting size reduction may be important, as we will show in section 7.

6 The Design of the Experiment

In order to evaluate the impact of our optimizations, HIPPCO was used to generate implementations of a restricted version of TCP from a high level specification. The TCP data transfer was completely specified in Esterel but TCP connection establishment was ignored. The specification is presented in appendix B. We then hand-coded a user-level version of the same restricted TCP. This implementation was directly derived from the BSD version of TCP.

In this experiment we were attempting to compare the performance of our code to that of BSD TCP with header prediction (the most common TCP optimization). We used a file transfer application as our test case. A client, already connected to a server, sends a 4 megabyte file. The data transfer is unidirectional: the server receives pure data and sends pure acknowledgment packets.

We are concerned with the quality of the generated code rather than the performance of underlying system utilities



(a) before optimization

(b) after optimization

Figure 5: Code size optimizations

like bcopy. Therefore we constructed a special purpose loop back test harness. This test harness eliminates all data copies and context switches. Thus the results below test only the control path of the protocol and ignore any performance effects of data handling.

HIPPCO automatically optimizes the protocol to meet the application requirements. These requirements are expressed to HIPPCO as the probabilities of external events and test outcomes. The real question in this experiment is how to set these values. For the purposes of this experiment, we used a variety of methods. First, in *raw profiling* the external events are considered equiprobable and the test outcomes probabilities are set to 50 %. Second, in *default profiling* these frequency variables have default values set by the module designer. These default values correspond to the common case as seen by the module designer, i.e. without information on the specific application that will use this module. Third, in *application profiling* the values are derived from the specific application behavior. The data used in application profiling will generally be determined by running an instrumented version of the protocol. Finally, if any of the application profiling probabilities are set to zero, we say that the code has been *specialized.* In this case, all branches with zero probability will simply be cut from the automaton. The resulting code is not fully functional TCP. It has been specialized for the particular application.

We focused our analysis on the input processing of the server side (receives pure data and send acknowledgments) of our test application. To insure fairness we compared our two TCPs instruction by instruction in order to verify that they implemented the same functionality. To isolate the cost of the protocol processing from the cost of the execution environment, we used the four following metrics for the performance evaluation: the average number of instructions executed (IC), the instruction cache miss rate ($miss\_rate$)[5], the average number of cycles per instruction (CPI) and the code size (the text and data segments size). This is similar to the approach adopted in [12]. A specific detail concerning IC should be taken into account: let I_c be the average number of instructions executed to process an input packet,

[5] We measure the miss rate as if the protocol was running alone. This gives a better insight on the performance effects.

and O_c be the average number of instructions to process an output packet (in our case to send an acknowledgment). Since in TCP, an acknowledgment is usually sent after the reception of 2 input packets, the average number of instructions executed on the reception of a packet, IC, is equal to $\frac{2\times I_c+O_c}{2}$. In our experiments, we used the ATOM tools to get these instruction counts. ATOM gives the number of instructions executed per function, thus IC. In some cases, it was easy to isolate in the code the part processing the input from the part sending the acknowledgment. In other cases, that was more difficult and we could not measure I_c and O_c. This explains why some elements of the table 2 are missing.

The experiments were performed on an Alpha 200 workstation. This workstation uses the 41466 Alpha CPU running at 166 MHz. The memory is composed of a primary instruction-cache and data-cache of 8Kbytes each, an unified 2MB second-level cache and 64MB of main memory. All caches are direct-mapped and use 32B cache-blocks. ATOM provides tools such as tracing, instructions profiling and instruction and data address tracing.

7 Analysis of the Results

The Esterel compiler generates a 21 state automaton from the specification. HIPPCO further reduces the number of states to 11. In fact, the Esterel compiler sometimes generates intermediary states that are used to solve causality problems. Those states are sometimes identical. For example in our TCP specification the intermediary states are just buffering states before exiting the automaton. HIPPCO detects these identical states and shares them.

In the following subsection, we present and analyze the gain achieved by the different optimizations. We then compare the implementations fully optimized by HIPPCO with the BSD code.

HIPPCO Optimization Performance Analysis

Table 1 presents the performance results for the different implementations. BSD results are shown for reference. The Esterel generated code results follow. A high gain factor is obtained by inlining the control of the protocol so that we obtain a compiled code. This optimization reduces the number of instructions by a factor of five. This result was expected. The original Esterel back end generates interpreted code. Each Esterel action is implemented by a function. A state is then defined by the set of functions that are called by the interpreter at the execution. Interpreted coding is thus very slow, as it requires one function call per elementary action. Compiling the code reduces the number of function calls and therefore the number of instructions required. It also improves pipelining and consequently decreases the CPI. However, applying this optimization (already partially supported in Esterel) resulted in a size explosion in our example (1772 Kbytes). We used the Esterel (O) optimization which reduces the code size to 45.7 Kbytes by sharing identical subtrees in the automaton.

HIPPCO optimizations are shown sequentially (application profiling was used for these measurements). We first apply the identical intermediate tree sharing code size optimization (S). This optimization increases the instruction count by 44 instructions (10 %) due to the introduction of some indirections, but decreases the code size to 30.48 Kbytes. Note that this version has bad cache and pipelining behavior compared to BSD. This result is explained by the poor code locality of the common path.

The second HIPPCO optimization (I), which inlines all function calls along the common path, reduces the number of instructions by 9 % (42 instructions), but resulted in a slight size increase to 30.816 Kbytes. This optimization also reduced the cache miss rate by compacting the code.

The third optimization (E), which extends the branches on the common path in order to generate straight-lined code (i.e. without function calls), reduces the number of instructions by 51 % (217 instructions) and increases the size to 32.24 Kbytes. Note a drastic improvement in cache behavior. The common path is now smaller than the cache size (8 Kbytes in these experiments) and the cache miss rate dropped to an insignificant value (0.0364).

The fourth optimization (T) has an positive impact on both cache and pipeline behaviors. However, this optimization resulted in a code size increase probably due to the test negations that were introduced.

The fifth optimization (U) outlines uncommon branches. It therefore does not affect the number of instructions executed on the common path. It enhances the cache behavior as one could expect.

The specialization optimizations (P and G), which respectively prune the branches of the tree with an execution probability of zero, and the states of the automaton that are never visited reduce the number of instructions by 5.4 % (11 instructions). These optimizations considerably reduce the size of the code (from 34.1 Kbytes to 6.672 Kbytes). This code reduction leads to a better cache utilization.

The last optimization (R), which reschedules the input processing subtrees according to the probability of each input event, reduces the number of instructions by 1 % (2 instructions). This corresponds to the rescheduling of one test: instead of testing whether a *End_ of_ connection* is received and then whether a packet is received, the reception of a packet (which has a higher probability) is first tested before the *End_ Of_ Connection* presence.

The numbers in brackets under text and data code sizes are the size of the HIPPCO generated code when the (N) code size optimization is applied. This optimization which shares similar trees in the automaton results in an important code size reduction. However, due to the lack of time, we did not execute experiments to evaluate the corresponding code speed. However, as this code size optimization mainly affects the uncommon path, the code speed of the common path is likely to stay the same. Note that the branch and state pruning optimization is much more efficient when this code size optimization is not performed. This is due to the fact that the (N) optimization shares similar trees by adding indirections in their original places. If the pruning of a tree occurs before applying the similar tree sharing optimization, an indirection could be saved. Therefore, if code specialization is to be performed, the branches and state pruning optimizations should be applied before the code size optimizations.

HIPPCO and BSD Performance Analysis

Table 2 presents a summary of the performance comparison between BSD TCP and the HIPPCO generated implementations. Two interesting observations arise from these results. The first is that the preconceived idea that classical Formal Description Techniques (FDT) tools generate poor quality code, seems to be also true for the original Esterel compiler (approximatively seven times slower than the BSD

	Code Speed			*Code Size (Kbytes)*	
	IC	*miss_rate*	*CPI*	*text*	*data*
BSD	397	1.0	1.28	16.864	1.760
Esterel (interpreted code)	2133	3.5	1.52	8.2	155.0
Esterel (compiled code)	-	-	-	1772.0	1.5
Esterel (compiled code + O)	421	3.86	1.30	45.7	1.2
HIPPCO: intermediate trees sharing (S)	465	4.07	1.32	30.480 (16.300)	3.52 (2.100)
+ Inlining (I)	423	3.7	1.33	30.816 (16.400)	3.60 (2.100)
+ Common Branch Extension (E)	206	0.0364	1.24	32.240 (19.648)	3.60 (2.100)
+ Rescheduling of Test Outcomes (T)	-	0.0345	1.23	33.136 (19.680)	3. 60 (2.128)
+ Outlining (U)	206	0.0289	1.23	34.160 (21.072)	3.80 (2.384)
+ Branch and state pruning (P,G)	195	0.0217	1.23	6.672 (10.624)	1.00 (1.456)
+ Input Rescheduling (R)	193	-	-	-	-

Table 1: Performance Results

	I_c	O_c	*IC*	*miss_rate*	*CPI*	*sizeKbytes(text)*
BSD	186	422	397	1.0	1.28	16.864
Esterel (interpreted)	-	-	2113	3.5	1.52	8.2
HIPPCO application profiling	143	147	204	0.0289	1.23	21.072
HIPPCO specialized	139	142	193	0.0217	1.23	10.624
HIPPCO raw profiling	-	-	320	4.7	1.31	32
HIPPCO default profiling	166	168	220	0.0298	1.25	24.84

Table 2: Summary of Performance Results

implementation). The second observation is that with the proper optimizations, automatically generated protocol implementations can be at least as fast and even faster (in terms of instruction count) than hand-coded protocol implementations. In fact, the code generated by HIPPCO is faster than the BSD code whether specialization is performed or not. For the input processing, HIPPCO code with default profiling requires 20 less instructions (11 %) than the BSD implementation. This is explained by the structure of the HIPPCO generated code: each state is implemented by a different function and some operations that are necessary for the BSD implementation are not for the HIPPCO implementation. For example, locating the current state is not necessary in HIPPCO because this is implicit. When the protocol automaton is activated, the function of the current state is actually executed.

When application profiling is performed, the HIPPCO implementation executes 43 (23 %) fewer instructions on the input path than the BSD implementation. This reduction is due to optimizations which transform the protocol automaton structure in order to better fit that particular application. HIPPCO code with specialization is only slightly faster than the application profiled code. In this case, the input processing requires 47 fewer instructions (25 %) than the BSD implementation. However, the biggest gain achieved by code specialization concerns the code size. It reduces the code size from 21.072 Kbytes to 10.624 Kbytes. With specialization, all the unnecessary parts of the code are removed. For example, in our test, the server only receives pure data and no acknowledgment, therefore branches which process the acknowledgment can be removed from the server common path. The resulting program is then smaller and faster.

With default profiling, the performance results are comparable to those obtained with application profiling. This result is explained by the predictable behaviors of the TCP protocol and its symmetry (the predictions at the server and client side do not differ too much).

With raw profiling, the performance is much worse. This result confirms the weakness of pure static analysis. The analysis of the structure of the automaton is not enough to identify precisely the common path and therefore to apply the optimizations in an efficient manner. This result motivates our approach of guiding the automaton analysis with prediction information.

The results on the output path show a reduction of the executed instructions of 70 % compared to the BSD implementation. Even though the measures on the output path were not performed as carefully as on the input path, they show that our approach also performs better than the BSD implementation on the output path. This large improvement is explained by the poor quality of the BSD output path implementation.

8 Conclusions and Future Work

The most important result of this work is that the systematic and automatic application of a known optimization can produce code which is faster than code manually optimized with that same optimization. For example the version of BSD we used did not do header prediction on the sending side. In HIPPCO, once implemented, header prediction is automatically applied on both the input and furthermore, automatic optimization appears to be a less error prone process. TCP-Lite was shipped with a minor bug in its header prediction code. The code as shipped includes one case in its fast path which requires re-calculation of the congestion window which is not done in the fast path [5]. HIPPCO optimizations are general tree to tree translations which can easily be shown not to change the semantics of the overall program.

It also needs to be driven home that a process intensive specification does not require a process intensive implementation. The Esterel front end takes a highly modular specification and produces a fully integrated automaton.

Another clear advantage of the automatic approach is that it is also easy to back out an optimization and re-optimize the protocol if the situation or traffic load changes. Once an optimization has been manually inserted into a complex protocol implementation it is basically there for the life of the protocol. There is no clean specification of the original protocol to work from as there is with HIPPCO.

However, actually eliminating unused code has a much less significant effect upon performance. We believe that this diminished effect is a result of the extensive systematic optimizations done before the unused code was eliminated. Clearly the basic goal of these earlier optimizations was to achieve the effect of removing the code without actually having to remove the code. Thus doing systematic optimization would seem to reduce the need for specialization at least of the limited form HIPPCO supports. This says nothing about the usefulness of doing specialization on pre-existing (and probably not particularly well optimized) code.

The greatest weakness of this research is the lack of a large scale test case. One would really like to see TCP, IP and ethernet all implemented in Esterel and optimized into a single automaton. This would test if the Esterel compiler front end can handle large enough protocols (and hence state machines) to make it a realistic option. It would also allow us to test the performance impact of integrating automatons from separately defined protocols and hence achieving automatic integrated layer processing.

This work has lead us to the conclusions that a protocol compiler is a practical tool for producing real protocol implementations rather than some form of academic exercise. Certainly more work is needed to produce such a tool but no truly significant roadblocks appear to be blocking the path to the creation of such a tool.

References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantic, implementation. Technical Report 842, INRIA, Sophia-Antipolis, France, 1988.

[2] Gérard Berry. Real-time programming: Special purpose or general purpose languages. In *Information Processing IFIP Conference, Elsevier Science Publishers, B.V. North Holland*, September 1989.

[3] Gérard Berry and Georges Gonthier. Incremental development of an HDLC protocol in esterel. Technical Report 1031, INRIA, Sophia-Antipolis, France, May 1989.

[4] Frédéric Boussinot and Robert de Simone. The ESTEREL language. Technical Report 1487, INRIA, Sophia-Antipolis, France, July 1991.

[5] L. Brakmo and L. Peterson. Performance problems in bsd4.4 tcp. *Computer Communication Review*, (5):69–86, October 1995.

[6] C. Castelluccia, I. Chrisment, W. Dabbous, C. Diot, C. Huitema, E. Siegel, and R. De Simone. Tailored protocol development using esterel. Technical Report 2374, INRIA, Sophia-Antipolis, France, Octobre 1994.

[7] Claude Castelluccia. Automating header prediction. In *Proceedings of the 1st Annual Workshop on Compiler Support For System Software*, Tucson, Arizona, February 1996.

[8] Claude Castelluccia and Walid Dabbous. Modular communication subsystem implementation using a synchronous approach. In *Proceedings of the Usenix High-Speed Networking Symposium*, pages 1–11, Oakland, California, August 1994.

[9] Claude Castelluccia and Walid Dabbous. HIPPCO: an HIgh performance protocol code optimizer. Technical Report 2748, INRIA, Sophia-Antipolis, France, December 1995.

[10] Claude Castelluccia and Phillip Hoschka. A compiler-based approach to protocol optimization. In S.-P. Chang, editor, *Proceeding of the High Performance Communication Subsystem Workshop*, Mistic, Connecticut, USA, August 1995.

[11] P. Chang, S. Mahlke, and W. Hwu. Using profile information to assist classic code optimizations. *Software-Practice and Experience*, 21(12):1301–1321, 1991.

[12] D. Clark, V. Jacobson, J. Romkey, and M. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[13] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, Pennsylvania, September 1990. IEEE. Computer Communications Review, Vol. 20(4), Sept. 1990.

[14] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software-Practice and Experience*, pages 581–601, June 1991.

[15] Jon Crowcroft, Ian Wakeman, Zheng Wang, and Dejan Sirovica. Layering considered harmful. *IEEE Network Magazine*, January 1991.

[16] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of a c function inliner. *Software-Practice and Experience*, 18:775–790, 1988.

[17] David C. Feldmeier. A survey of high performance protocol implementation techniques. Technical report, Bellcore, February 1993.

[18] Per Gunningberg, Craig Partridge, Teet Sirotkin, and Bjorn Victor. Delayed evaluation of gigabit protocols. In *Proceedings of the Second MultiG Workshop*, Stockholm, Sweden, June 1991.

[19] J. L. Hennesy and D. D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, 1990.

[20] B. Hoffmann and W. Effelsberg. Efficient implementation of Estelle specification. Technical Report 3/93, Universitat Mannheim, Mannheim, Germany, March 1993.

[21] Philipp Hoschka. *Optimisation Automatique dans un Compilateur de Talon de Communication*. PhD thesis, INRIA, Sophia Antipolis, France, July 1995.

[22] Norman C. Hutchinson and Larry L. Peterson. Design of the x-kernel. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 65–75, Stanford, California, August 1988. ACM. also in *Computer Communication Review* 18 (4), Aug. 1988.

[23] S. Leue and Ph. Oechslin. Optimization techniques for parallel protocol implementation. In *Proceedings of the Fourth IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 387–393, 1992.

[24] J.A. Manas and T. de Miguel. From LOTOS to C. In *Proceedings of the International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols*, pages 79–84, 1988.

[25] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[26] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Paterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report 94-20, Computer Sciences Department - University of Arizona, Tucson, Arizona, USA, June 1994.

[27] David Mosberger, Larry L. Peterson, and Sean O'Malley. Protocol latency: MIPS and reality. Technical Report TR 95-02, Department of Computer Science, University of Arizona, Tucson, Arizona, January 1995.

[28] Philippe Oechslin. *Implémentation Optimisée de Protocoles a Haut Débits*. PhD thesis, EPFL Lausanne, Lausanne, Switzerland, 1995.

[29] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[30] V. Roy and R. de Simone. Auto and autograph. In *Proceedings of the Workshop on Computer Aided Verification*, June 1990.

[31] K. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science.* Englewood Cliffs: Prentice Hall, 1982.

[32] Son T. Vuong, Allen C. Lau, and R. Isaac Chan. Semi-automatic implementation of protocols using an Estelle-C compiler. *IEEE Transactions on Software Engineering*, 14(3):384–393, March 1988.

A Esterel Specification of the TCP Timer Management Module

The specification in figure 6 shows the Esterel specification of the management of the TCP retransmission timer used as example in the paper. For simplification, variable declarations are not shown.

All statements resulting in branches in the flow graph corresponding to this module are annotated by the probability to be taken. For a *wait* or a *case* statement, the value gives the probability that the input signal of the case statement will be observed. For an *if* statement, the value gives the probability that the condition of the statement is true.

The figure 7 shows the corresponding automaton generated by the Esterel front end compiler.

B Specification of TCP in Esterel

In this appendix, we address briefly how the Esterel modules were designed and combined to generate the required protocol. A more detailed description of this specification is presented in [8].

The building blocks have been designed to be meaningful to the designers and so that changes in the protocol specification only induce local changes in the architecture and the code. An overview of the whole system is shown in figure 8. For clarity purposes, only the main modules have been described and displayed.

The specification is structured in three main concurrent modules: the Send Module, the Receive Module and the Connection Module. The Send Module is composed of two concurrent submodules:

- The *User_Input_Handler* module which is a four state automaton. The first is the initialization state and the last the absorbing state. In the second state, the module is expecting data from the application. Whenever it receives data, it processes and copies it into an internal buffer. If the buffer is full, it goes to state 3, otherwise it broadcasts a *Try-to-Snd* signal and goes back to state 2. In state 3, this module is waiting for an acknowledgment packet. If this packet frees some space in the buffer, the module goes back to state 2, otherwise it stays in state 3. The module returns to state 1 when a *End_of_Connection* signal is received.
- The *Emission_Handler* module transmits packets on the network. It is a three state automaton. The first is the initialization state and the last the absorbing state. In the second state, the module can be activated by several signals, among them the *Try_to_Snd* signal emitted by the previous module. The module then tries to

```
module TIMER_HANDLER:
/* inputs declaration */
input N_S     (integer), /* current sequence number */
      N_A     (integer), /* ack. sequence number */
      ALARM   (integer), /* retransmission timer */
      Rxt_Cur(integer); /* current rtt estimate */
...

loop
 trap NOTIMER in
  await  N_S do  P(1.0)
   /* set retrans. timer to current rtt estimate */
   emit SET_ALARM(?Rxt_Cur);
   /* store seq. numb. of ack for next rtt estimate */
   N_Clock := ?N_S;

  loop
   trap RETRANSMIT in
    await
     case N_A do P(0.95)
      if (?N_A >= N_Clock) P(0.9) then
       /* there are still unacknowledged packets */
       if (?N_S > ?N_A) P(Pkt_N-1/Pkt_N) then
        /* set retransmission timer to current rtt estimate */
        emit SET_ALARM(?Rxt_Cur);
        /* store sequence number of ack for next rtt estimate */
        N_Clock := ?N_S;
       else
        /* all packets acknowledged -> reset timer */
        emit SET_ALARM(-1);
        exit NOTIMER;
       end
      end
     case ALARM do P(0.05)
      trap CLOSE_TCP in
      /* activate retransmission phase */
      emit Retrans_Phase(1);
      /* activate slow start module */
      emit Slow_Start;
      t_rxtshift := t_rxtshift+1;
      /* bound on max number of retransmissions exceeded */
      if (t_rxtshift >= TCP_MAXRXTSHIFT) P(0.01) then
       t_rxtshift := 0;
       /* close the connection */
       exit CLOSE_TCP;
      end;
      await N_S do
       /* recalculate retrans. timer using exp. back-off */
       Rexmt := (SHIFT_RIGHT(?T_Srtt, TCP_RTT_SHIFT)
              + ?T_Rttvar) * TCP_BACKOFF(t_rxtshift);
       Rexmt := TCPT_RANGESET(Rexmt, TCPTV_MIN , TCPTV_MAX);
       /* set timer */
       emit SET_ALARM(Rexmt);
      end;
      handle CLOSE_TCP do nothing;
    end; /* loop */
  end /* await */
end. /* module TIME_HANDLER */
```

Figure 6: Esterel Specification of TCP Timer Management

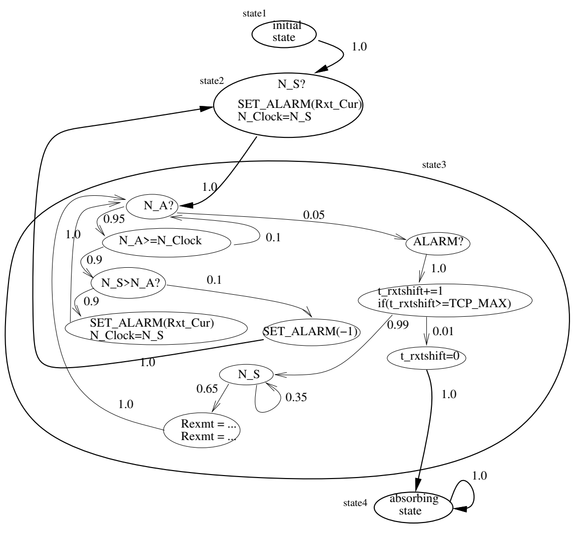


Figure 7: TCP Timer Management Automaton

send packets by evaluating the congestion window size, the silly window avoidance algorithm and the number of bytes waiting to be sent. It may then send one or several packets. The *Send_ Now* input forces the sending of a packet even though the regular sending criteria are not satisfied (this is used to acknowledge data for example). If the module decides to send packets, the checksum function is called and the header is completed.

The Receive Module processes the incoming packets. It is itself composed of several three state modules that scan all incoming packets, check their validity (checksum, length, ...) and broadcast all the header fields within the specification. Then it delivers the packet to the application.

The Connection Module is composed of the following submodules that are executed concurrently:

- The *RTT_ Handler* is a three state module that computes the round trip time of the connection. When a packet is emitted, and if no other packet belonging to this connection is in transit and we are not in a retransmission phase then a timer is started. When this packet is acknowledged, the timer is stopped and a new RTT value is computed.

- The *Window_ Handler* is a three state module which concurrently updates the congestion window and the send window (corresponding to an evaluation of the space left in the receiving buffer of the remote host). When an acknowledgment signal is received, the *Window_ Handler* increases the congestion window either linearly or exponentially according to the slow-start algorithm, and the sending window is either updated to the value of the *Win* field (emitted by the *Scan_ Handler* module) or reduced by the number of bytes acknowledged. If a signal corresponding to a window shrink request (from the *Timer_ Handler* module) is received, the congestion window is set to 512 bytes (value of the Maximum Segment Size).

- The *Acknowledgment_ Handler*: is a three state module which handles the acknowledgment information received in the incoming packet. If the received acknowledgment sequence number (which corresponds to the value of the next sequence number the remote host is expecting to receive) is greater than the latest byte sent or less than the last already acknowledged byte then, it is ignored. Otherwise, the acknowledged value is updated.

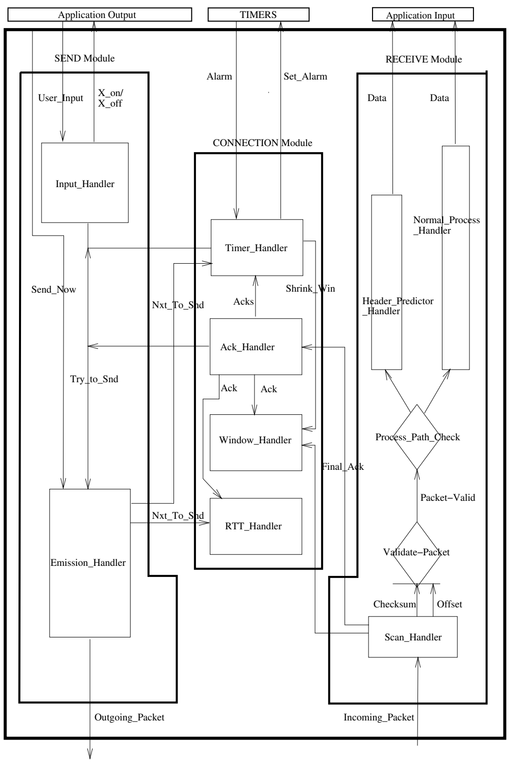


Figure 8: Protocol Specification Overview

- The *Timer_ Handler* module manages the different timers. It is a five state automaton. In state 2, no packet is in transit and the retransmission timer is not set. When a packet is sent on the network, the timer is set and the module goes into state 3. In this state, two types of events may occur: an acknowledgment signal or an *Alarm* signal. If an acknowledgment signal, which acknowledges all packets in transit, is received the retransmission timer is reset and the module goes into state 2. If an *Alarm* signal is received, the module goes to state 4, where it is waiting for the retransmission packet before resetting the timer with a larger value.