HIPPCO: A High Performance Protocol Code Optimizer

Claude Castelluccia, Walid Dabbous

N° 2748

Décembre 1995

PROGRAMME 1

ISSN 0249-6399

HIPPCO: A High Performance Protocol Code Optimizer

Claude Castelluccia, Walid Dabbous

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet RODEO

Rapport de recherche n ° 2748 — Décembre 1995 — 76 pages

Abstract: This report presents HIPPCO, an High Performance Protocol Code Optimizer. HIPPCO belongs to the HIPPARCH compiler. HIPPARCH is a tool which proposes to generate automatically from the application communication requirements and the network characteristics an efficient implementation of a customized protocol.
HIPPCO is the last stage of this protocol compiler. It takes as input a description of the protocol automaton, optimizes it and generates an implementation in *C*.
HIPPCO decomposes the protocol automaton in two parts: the common and uncommon path. It then uses this decomposition to apply a set of optimizations toward a good code speed/code size tradeoff.
In the first part of this report, the code speed optimizations are described. Those optimizations reduces the number of executed instructions and improves the instruction cache and pipeline behaviors. In the second part, a comparaison of HIPPCO automatically generated implementations of TCP are compared with the BSD implementation. We show that the HIPPCO generated codes requires up to 70% less instructions than its BSD counterpart.

Key-words: Code optimization, Automated Protocol Generation, ALF

(Résumé : tsvp)

HIPPCO: Un optimiseur haute performance de code de protocole

Résumé : Ce rapport décrit l'architecture de HIPPCO, un optimiseur et générateur d'implémentation de protocoles de communication. HIPPCO fait parti du compilateur de protocoles HIPPARCH. Ce compilateur propose de génèrer automatiquement, à partir des besoins de communication d'une application distribuée et des caractéristiques du réseau sous-jacent, le protocole spécialisé correspondant.
HIPPCO est la dernière passe du compilateur HIPPARCH. Il génère à partir de la description de l'automate du protocole, une implémentation optimisée en *C*.
HIPPCO décompose l'automate de communication en deux parties distinctes: le chemin fréquent et le chemin rare. Utilisant cette décomposition, un ensemble d'optimisations est appliqué pour obtenir le meilleur compromis entre la vitesse d'exécution et la taille du code.
Ce rapport est structuré en 2 parties: La première détaille les différentes optimisations utilisées pour l'amélioration de la vitesse d'exécution du code. Ces optimisations sont basées sur une réduction du nombre d'instructions à exécuter, sur une meilleure utilisation du cache d'instruction et du pipelining.
Dans le seconde partie, une implémentation automatique du protocole TCP est detaillée. Les performances de cette implémentation sont comparées avec celles de la version BSD de TCP.

Mots-clé : Optimisation de code, Génération automatique de protocoles, ALF

1 Introduction

The emergence of new distributed computing applications and of new networking technologies is driving the need of more specialized communication protocols. Generating and implementing those new protocols manually in an efficient way is very often a complex and time-consuming task.

One option, that has been extensively studied in the last few years, is to automate or semi-automate the protocol design and implementation phases. This is usually performed by selecting a set of predefined modules, which generally implement the basic protocol mechanisms, and combining them into the final communication system using configuration tools like ADAPTIVE or x-kernel [BSS92, NL88, OP91]. This approach has the advantage over the manual approach to generate a modular and configurable system.[1]

However, this approach has not encountered yet the success that was expected. The main reason is that the modularity of those systems introduces a performance penalty on the generated protocol implementation. In fact, those configuration tools very often derive the implementation directly and naively from the specification. As a result, the generated protocol implementation is composed of several modules that communicate and interact via some kind of interfaces. These module interaction mechanisms are usually very costly and are the source of performance penalties that overtake the gain achieved by the configuration.

Another limitation of these configuration tools is that they generally use a bottom-up approach: i.e. they configure a communication system based on the mechanisms provided by current "transport protocol". The granularity of those tools is then very often coarse.

In the HIPPARCH project, we propose to build a protocol compiler which automatically generates the communication system of a distributed application. This approach is interesting only if the generated implementations are performant. The efficiency of the implementations was therefore one of our main goals. To reach this objective, the HIPPARCH compiler has been built around the following properties:

- **Synchrony:** The HIPPARCH compiler is built around a synchronous language, Esterel. Esterel generates from a set of concurrent modules an integrated automaton. In this automaton, all the specification building blocks are merged together and communicate directly without interfaces.

[1] A system is said to be *modular* if it is composed by a number of functional entities called modules which interact together. A system is said to be *configurable* or *flexible* if it is relatively easy to add or remove a functionality without modifying the whole system.

- **Configurability:** The HIPPARCH compiler selects the protocol mechanisms according to the application requirements and networks characteristics. The HIPPARCH approach is application-led (top-down): the communication requirements of several applications (such as JPEG photo server [C. 95], secure login, multicast video and audio, WWW transactions, multimedia multicast mail delivery, large scale multicast image dissemination [A. 95]) have been studied and analyzed. The compiler should generate the complete communication system for such applications.
- **HIPPCO:** The HIPPARCH compiler features a *code optimizer*, called HIPPCO, which optimizes the structure of the automaton and generates highly optimized code in *C*. HIPPCO uses profiling information provided by the protocol designer to dynamically identify the *common path* and apply a set of optimizations. Those optimizations are specific to the automaton structure and are not redundant with optimizations that may be performed by existing low-level compilers. The goal of HIPPCO is to generate protocol code that can efficiently be compiled and optimized by current *C* compilers. It optimizes a program execution time by reducing the number of instructions on its common path and generating code that exhibits good cache and pipeline behaviors.

In this report, we describe the design of HIPPCO and present some performance results of TCP automatically generated implementations. We show that automatically generated protocol code can be faster than the best optimized hand-coded implementations. This report is composed of 6 main sections. Section 2 introduces the HIPPARCH project and gives some insights on the Esterel language. Section 3 presents HIPPCO concepts and describes its optimization principles. Section 4 details the design of HIPPCO. In this section, we present the various optimizations that are performed by HIPPCO. Section 5 evaluates the performance of HIPPCO automatically generated codes. We compare the instruction counts, i-cache and pipeline performance of some HIPPCO generated TCP implementations with the BSD implementation. We also evaluate the individual effects of each proposed optimization. We, finally, conclude in section 6.

2 Context: The HIPPARCH Protocol Compiler

2.1 General Presentation

The work presented in this report was done in the context of the HIPPARCH project [CCD$^+$94]: an European-Australian collaboration action which proposes to study a novel architecture for communication protocols based on the Application Level Framing (ALF) and Integrated Layer Processing (ILP) concepts [Cla90]. Its final objective is to build a compiler which generates the communication system of a distributed application. This tool uses application-specific knowledge to configure this communication system for improved performance [Cas94a].

The HIPPARCH compiler is composed of 2 tools : the ALF and ILP compiler. The ALF compiler generates from the application requirements and the networks characteristics an automaton representing the control part. This control part contains the application and protocol control parts. The idea of HIPPARCH is to integrate the application and its communication subsystem within a single automaton. This integration removes a lot of interfaces and leads to more performant systems. In this report, for simplicity reason, we separate the control part of the application from the control part of the communication subsystem. This allows us to generate existing protocols and compare their performance to their manually implemented counterparts. The ILP compiler is a stub compiler, which combines the data manipulations functions in an ILP manner.

The control automaton and the data manipulation functions are combined together to form the complete protocol implementation as illustrated in figure 1. Whenever the application sends a piece of data to a remote correspondant, it calls the protocol output procedures which process the control part of the protocol. The protocol control variables (sequence number, timers, transmission window,...) are then updated and the data manipulation functions are called with the data and some control information as arguments. This control information depends on the generated protocol. It may be composed of some fields of the protocol control block that are used to build the protocol header or to process some of the manipulation functions. After the data manipulation operations (checksum, encryption, marshalling, ...) are performed, packets are built and sent on the network. At the other side, when a packet is received, it is first marshalled, decrypted ans checksummed. The processed information (data and control information) is then handed to the control automaton. The protocol control variables will then be updated, a packet possibly sent, and data is delivered to the application.

Separating the data manipulation functions from the control flow processing has the advantage of isolating the problems and simplifying the overall compiler design. It also allows to consider each part independently and apply more specific optimizations. In this report, we do not consider the ILP compiler (i.e. how the data manipulation functions are implemented). We describe the HIPPCO tool which optimizes the control automaton, and therefore is a part of the ALF compiler.

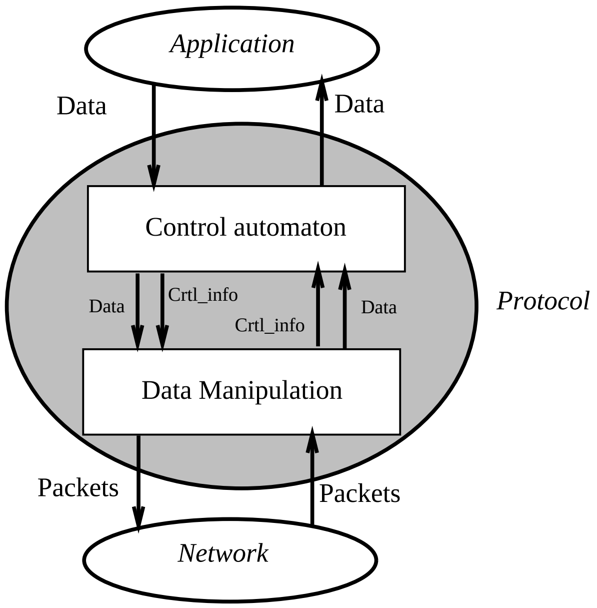

Figure 1: Protocol Structure

The ALF compiler is designed around a synchronous language, Esterel ([Ber89], [Ber92]). It is composed of 4 main parts (figure 2-a):

- *a library* of predefined modules in Esterel. It is a collection of various protocol building blocks.
- *a parser* which combines the application specification written in Esterel with selected library modules, to generate an Esterel specification of the communication stack and some profiling information.
- *an Esterel Front-End*: compiles the Esterel specification into an integrated automaton.
- HIPPCO: optimizes the generated automaton and generates an efficient implementation in a target language (C in HIPPARCH).

Although HIPPCO is developped in the context of the HIPPARCH project, it was designed independently of the other HIPPARCH compiler tools. HIPPCO takes as input any automaton description in a standard language called `Oc` and generates an implementation in the `C` language. Therefore HIPPCO can in principle be used to optimize the output of any tool generating `Oc` automata, independently of the high-level specification language. To illustrate this, we describe in figure 2-b, an alternative way to use HIPPCO. The proposed architecture optimizes existing communication system implementations provided that an integrated automaton generator exists. This automata generator takes as input the different *C* protocol implementations of a communication stack, and generate an integrated automaton in the `Oc` language, which is feeded into HIPPCO to produce an integrated and optimized implementation of the communication system.

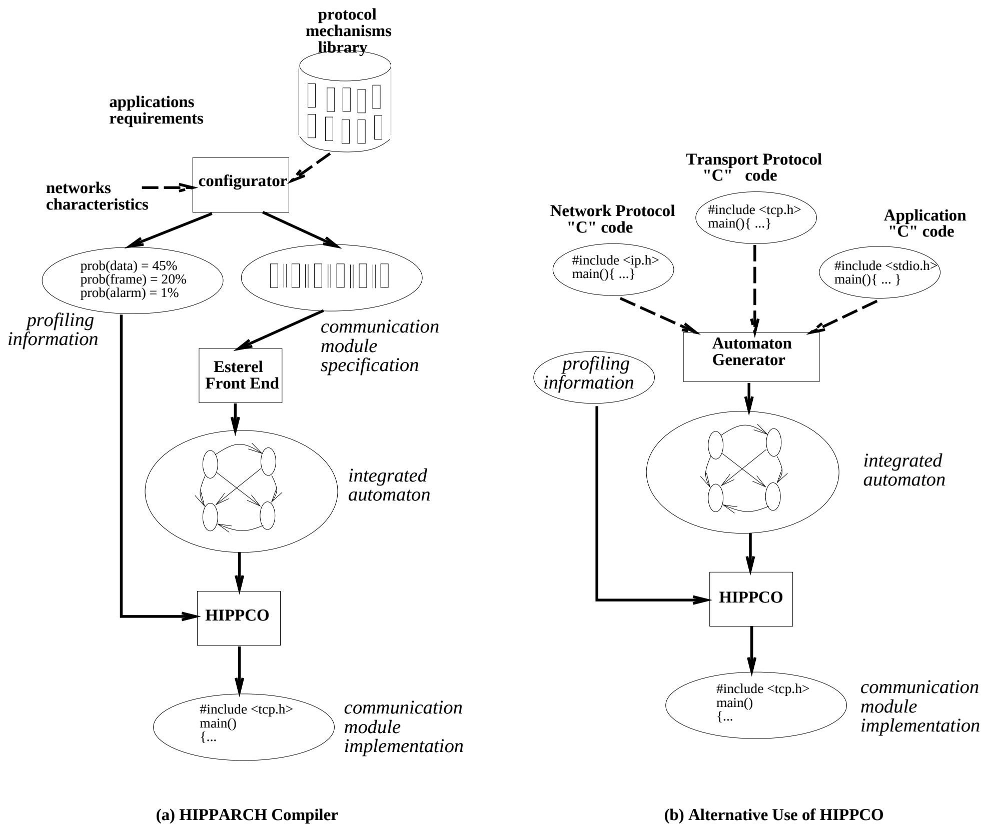
protocol mechanisms library
applications requirements
networks characteristics
configurator
prob(data) = 45%
prob(frame) = 20%
prob(alarm) = 1%
profiling information
communication module specification
Esterel Front End
integrated automaton
HIPPCO
#include <tcp.h> main() {...
communication module implementation
(a) HIPPARCH Compiler
Transport Protocol "C" code
#include <tcp.h> main(){ ...}
Network Protocol "C" code
#include <ip.h> main(){ ...}
Application "C" code
#include <stdio.h> main(){ ... }
Automaton Generator
profiling information
integrated automaton
HIPPCO
#include <tcp.h> main() {...
communication module implementation
(b) Alternative Use of HIPPCO

Figure 2: Protocol Compiler Architectures

In the HIPPARCH compiler, HIPPCO takes as input an automaton description generated by the Esterel compiler. In the following, we introduce the Esterel language and describe the generated automaton format.

2.2 Esterel

2.2.1 The Esterel Language

Programs can basically be divided into three classes : (1)transformational programs that compute results from a given set of inputs, (2) interactive programs which interact at their own speed with users or other programs and (3) reactive programs that interact with the environment, at a speed determined by the environment, not by the program itself.

Synchronous languages were specifically designed to implement reactive systems. The Esterel language is one example [BdS91, BG89]; others include languages such as Lustre [], Signal [], Sml [] and Statecharts [].

Protocols are good examples of reactive systems; they can be seen as *black boxes*, activated by input events (such as incoming packets) and reacting by producing output events (such as outgoing packets). Other systems e.g. window servers such as X, NeWs and MS Windows, can be considered as reactive systems and may be specified using synchronous languages.

The Esterel language was chosen as the specification language within the HIPPARCH project [].

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.

Synchronous languages are used to implement the control part 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 the assumption of perfect synchrony: 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 [RdS90].

At compile time, the Esterel program is translated into a sequential finite automaton; the code of the different modules is sequentialized according the program concurrency and synchronization specifications.

However the synchronous approach cannot be considered as a stand-alone solution, principally because the synchronous assumption is not a valid one in the real implementation world. A so-called *Execution Machine* is required [AMP91]. This machine is aimed at interfacing the asynchronous environment to the synchronous automaton. It collects the inputs and outputs and activates the automaton only when it is not executing; the "synchronous assumption" is then respected.

2.2.2 The Esterel Tree Representation

The Esterel front-end generates from the specification an automaton in a standard language called `Oc`. The generated automaton is usually composed of several **states**. Each of those states are described by a state-tree (figure 3). Trees, unlike graphs, do not contain any loop. This property facilitates dependency analysis and simplifies optimizations design and implementation. As we will show later in this report, trees can be easily modified for better performance. However the tree representation generally leads to large code size. In fact in a tree representation, lots of actions are duplicated in the different branches. A graph representation usually leads to a smaller code size.

Each 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 those different subtree's roots as shown in figure 3. An incoming event will then be processed by a sequence of tests until the appropriate subtree is found.

We define a set of notations that will be used throughout the whole report. They are illustrated by the figure 3. A state-tree is composed of a collection of cascaded nodes. There are three types of nodes: `decision_nodes`, `state_nodes` and `action_nodes`:

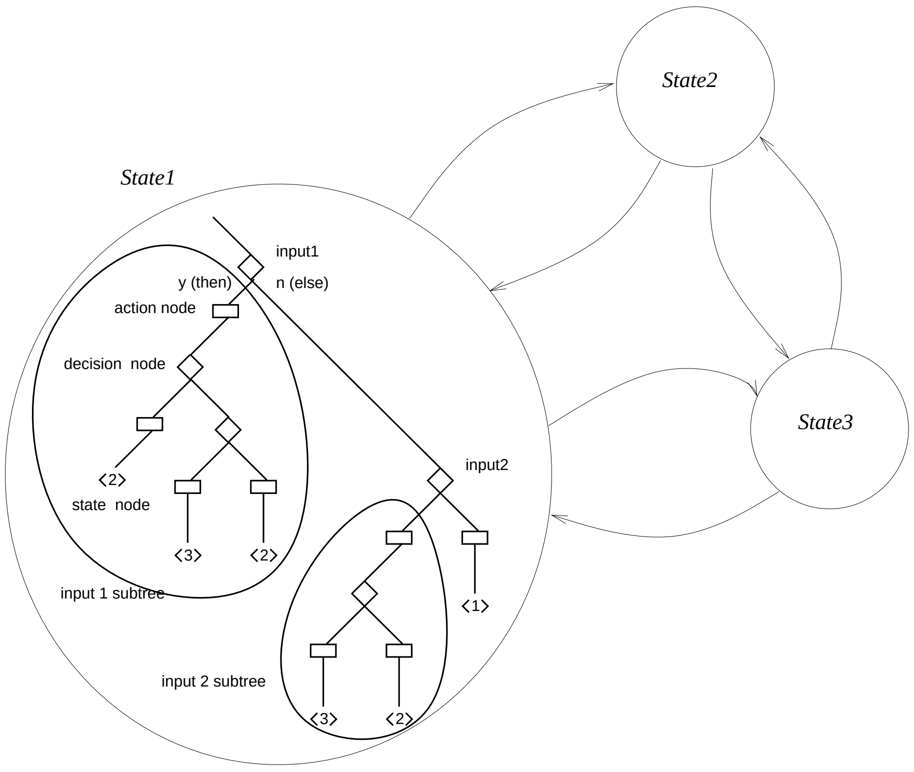

Figure 3: The Tree Structure

- In Esterel, 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. `else` is a pointer on the node to execute if the test is uncorrect. We added two more elements: `per` and `CP`. These two elements will be used by HIPPCO. `Per` is an integer which indicates the execution probability of the `then` branch. `CP` is a flag, used by the common path identification algorithm.

- 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 tree, but as we will see later in this report, every leaves are not `state_nodes`. In fact, some leaves may be references to other trees.

- An `action_node` is a node that performs an action or a set of actions, such as variable assignements and/or function calls. It is composed of two elements: the action itself (`elt_action`) and a pointer on the following node to execute (`nnext`).

We adopt the following conventions in the tree figures. Decision nodes will be drawn as diamonds. The backward branch of a decision node (the `then` branch) is its left branch. Action nodes will be drawn as rectangles. State nodes will be drawn as two brackets containing the value of the `nstate` variable (<>). After code size optimisations, some trees are ponting on others trees. A reference to a tree labeled `n` is coded as [n]. In this report, some of these details will be omitted if this does not affect the figure clarity.

3 HIPPCO: High Performance Protocol Code Optimizer

3.1 Presentation and General Concepts

The optimizer operates in two phases (see figure 4): the first stage, *the structural optimizer*, modifies the structure of the internal automaton, reschedules the different actions according to the optimization constraints and generates an optimized automaton customized to the application that is being designed. The second stage, *the code optimizer* generates, from this automaton representation, the final code.

The structural or automaton optimizer is independent of the final target language. It takes as input the automaton representation in `Oc` and generates an other automaton in `Oc`. The code optimizer is independent of the automaton optimizer, it takes as input an automaton description (optimized or not) and generates code in a target language (such as C). The optimization performed by those two stages are quite different. The first stage generates optimizations that modify the software structure: optimizations like header prediction [Jac88] should be taken into account at this stage. The code optimizer applies code optimization techniques that are more language-specific such as inlining. Separating these two kinds of optimizations into two different stages has the advantage of isolating them and therefore leads to more efficient algorithms.

HIPPCO design follows four basic concepts:

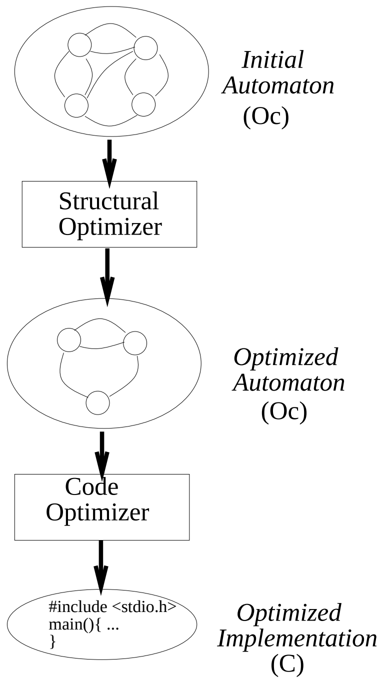

Figure 4: HIPPCO Architecture

- **Complementary to low-level compiler optimizations:** Existing low-level compilers [2] perform a lot of optimization techniques. Those optimizations transform the code to generate faster code without information on the semantic of the application. They are applied either as a result of a static analysis of the program or systematically, such as inlining very small functions.

  The HIPPCO optimizations uses profiling information provided by the application designer. It can therefore perform more aggressive and specific optimizations. However some optimizations are more efficient when they are performed at the low-level, such as *Basic Block Rescheduling* ([McF89]) for better cache utilization. As a consequence, the approach that we followed in the design of HIPPCO was to implement optimizations that are not redundant with the ones present in the low-level compiler but that are complementary. HIPPCO was not designed to replace the existing low-level optimizers, but to help them. In

[2] We call low level compiler, a compiler that generates executable code from a high level language such as the C compiler.

HIPPCO, we transform the code in order to make the low-level optimizations more efficient. For example, when the *Common Branches Extension* algorithm is applied (see section 4.2.3), it generates code on the common path that is straightline. This code is faster because of the suppression of the function call overheads but also because the streamlined structure of the common path gives to the low-level compiler more optimization opportunities.

- **Synchrony:** Most of languages and code generators currently used for protocol specification and design, such as SDL or Estelle, are asynchronous ([Hof93], [Leu94]): 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 implementation: synchronous languages compile parallel or sequential specification modules into a single automaton using the language semantic, the data and control dependencies. This automaton can be implemented by a single process, removing inter-module communication overhead. This approach has in addition the advantage of facilitating the verification of the generated code [RdS90].

- **Configurability:** A possible way to enhance a communication system performance is to reduce the protocol functionalities (i.e. the number of protocol operations) to the minimum required by the application. This leads to a communication system configured to meet the application needs. In HIPPCO, profiling information provided by application designer are used to generate a configured version of this system.

- **Common/Uncommon path separation:** Another way to enhance the performance is to reduce the amount of processing per operation. Improving the performance of the protocol operations is classically achieved by optimizing the operations that are on the common path [HH93, CH95]. The common path, also called the critical path or the fast path, is the part of the code that is the most frequently executed. By optimizing this path, the execution time of the overall program under normal conditions will be optimized. This optimization technique has been widely utilized to increase the performance of protocol code. The TCP BSD header prediction is probably the best known example of an optimized common path protocol implementation [CJRS89]. The concept of separating the common and uncommon path for protocol code optimization is motivated by three reasons:

1. Optimizing the execution speed a program is a difficult process, it is then important to focus the optimization effort on the most profitable parts of the code. The common path is only a small part of the whole program but it is executed more frequently than the rest. Therefore, optimizing the common path generates relatively higher performance gain. In fact, the average number of instructions executed by a program can be evaluated by the following formula:

   $$IC_{Av} = IC_{CPath} * freq_{CPath} + IC_{UCPath} * freq_{UCPath}$$

   where IC_{CPath} is the number of instructions that are executed when the incoming event follows the common path and IC_{UCPath} is the number of instructions that are executed when the incomming event follows the uncommon path. The variables $freq_{CPath}$ and $freq_{UCPath}$ define the execution frequencies of each path. $freq_{CPath}$ is expected to be close to 1.0, whereas $freq_{UCPath}$ close to 0.0.

   The gain achieved by reducing the number of instructions on the common path is then $freq_{CPath}/freq_{UCPath}$ times larger than the gain achieved by the reducing the number of instructions by the same amount on the uncommon path [3].

2. Code optimizations involve somehow tradeoffs. A classic example is the inlining optimization, which reduces the number of instructions by replacing each function call by the body of the corresponding routines. This optimization, if performed systematically, can generate very large codes which lead to very bad instruction cache behavior and very poor performance (see section 5.5.6). A solution to this problem is to perform those optimizations selectively on the common path.

3. In protocol implementation, the distinction between the common and uncommon path is clear (this is not the case for all types of programs). There are many paths through the protocol codes, however there is only one common path which is usually only composed of few tens of instructions ([CJRS89]). The code to be optimized corresponds to the normal case. In this case, the protocol behavior is generally predictable. In addition, there is no need to optimize the path followed when a error occurs,

[3] In this analyse, we neglected the performance penalty added to the uncommon path by the optimization the common path. We expect the product of this penalty with the execution frequency of the uncommon path to be very small.

such as a packet loss. In fact, in those cases, the bottleneck is not the protocol code but the network or the interface. Optimizing that path will have a minimal effect on the protocol performance. HIPPCO makes intensive usage of this common/uncommon path distinction.

3.2 Optimization Principles

HIPPCOs' primary goal is to generate protocol code that has very fast execution time and small code size. This optimization goal has been divided into two parts:

- code execution speed optimization
- code size optimization

These two optimizations can not be treated individually because they are not completely independent. As we show in this report, some code speed optimizations increase the code size and similarily some code size optimization imply some speed penalties. This report only presents the code speed optimization algorithms. The results of the work on code size optimization algorithms will also be cited, but the details will be presented in another paper [Cas95b]. The goal of the code speed optimization is to reduce the number of cycles that a program takes to execute. This is a complex task, because the execution speed of a code depends on several components that interact together. The number of cycles required to execute the program is given by the following formula [HP90, Cas95a]:

$$Cycles_{total} = IC * CPI_{Execution} + Memory\_accesses * Miss\_rate * Miss\_penalty$$

where IC is the number of instructions executed, $CPI_{Execuction}$ the average number of cycles per instruction, $Memory\_accesses$ the total number of memory accesses within the program, $Miss\_rate$ the rate of memory references which are not in the cache and $Miss\_penalty$ the penalty, expressed in cycles, encountered when a memory access is performed.

The number of cycles is composed of the sum 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 the memory stalls are all due to the cache. Although this is not true for all machines, the stall due to the cache always dominate the effect of other stall sources.

We also consider, in this formula, that the cycles for caches accesses are part of the CPU-execution cycles and are therefore included in $CPI_{execution}$.

According to this formula, a program execution time depends on three components:

- The instruction count (IC)
- The memory stall cost $(Memory\_accesses * Miss\_rate * Miss\_penalty)$
- The number of cycles per instructions (CPI)

Optimizing the global execution speed involves optimizing each of its components. We will present in the three following subsections the general program optimization principles for each of those components. In section 4, we describe how these principles are applied to protocol code optimizations in HIPPCO.

3.2.1 Instruction Counts Optimizations

The optimizations that improve program performance by eliminating instructions are widely used in today's compilers. However most of those optimizations are limited, because they are only based on static analysis (control flow, dependency analysis). Compiler optimizers that use profiling data are not very commonly used yet. In this subsection, we described the approach that is commonly used to optimize programs through high-level transformations. We will discuss machine level optimizations (such as register allocation,...), or optimizations that can be performed by lower-level compilers such as loop unrolling, redundancy elimination. We present optimization techniques that uses additional informations about the application behavior to reduce the instruction count.

The optimizations that tend to reduce the number of instructions to perform are usually performed on the most frequently executed parts of the code. Those optimizations can be classified in two categories: first, those modifying the structure of the program and which are independent of the programming language, we will call them the *structural optimizations*. Second, the optimizations that reduce the number of instructions by some programming techniques. We call them *code optimizations*.

The structural optimizations consist of restructuring a program such that the most frequently path is scheduled at the beginning of the program. Events that follow the common path are then detected earlier. All processings that belong to the uncommon path are scheduled out of the common path stream. Such an optimization is very desirable in the case of communication protocols is the cyclic usage behavior of these protocols (e.g. bulk transfer TCP has same size, in order packets 95 % of the time on most networks). A classic structural optimization is the

TCP header prediction [CJRS89] implemented in TCP BSD. It predicts the values of the fields of the next incoming packet, such that the reception of in order packets can be processed in few instructions. Some experiments that we performed, and which are presented in section 5.5.2, showed that the TCP header prediction reduces the average number of instructions, necessary to process an incoming packet, from 508 to 186.

The code optimization that is the most usually used in programming language such as C is the inlining technique. Inlining involves a space/speed tradeoff, therefore it should be applied selectively. The effect of inlining on program performance has been extensively studied [CHT91, DH92, Hos95]. The issue of how function inlining can be automated has been addressed several times in previous research [DH92, Hos95]. Current low-level compilers feature inlining optimizations. Usually those compilers only inline simple and small functions. The heuristics used can not perform very aggressive inlining because the frequencies of execution of each functions is difficult to evaluate at the low-level.

In section 4.1, we show how the automatic common path identification and the intermediate program representation facilitates the application of structural optimizations in HIPPCO. We also show how the prediction information provided by the application designer provides more opportunities to perform efficient code optimizations. Heuristics based on the predictions of the different code part execution frequency are used to apply inlining in the most optimal way.

3.2.2 Memory Stall Optimizations

Traditional optimization techniques improve program performance by reducing the number of instructions. Those optimizations have been designed at the time where most of the cycles were used to execute the program instructions.

However as the processor speeds increase at a much faster rate than the memory speeds, the bootleneck has been shifted to the memory system [CT90, GPSV91, Fel93].

The emergence of the RISC technology, which features very simple and uniform instructions, amplified the effect of the costly memory access for at least 2 reasons: first, programs running on a RISC processor are longer than the one running on CISC type processor and therefore perform more instruction memory references. Second, the uniform RISC instruction can be efficiently pipelined. However if one of this pipeline components, such as the *fetch* instruction, takes more time than the other, the pipeline efficiency is not optimal.

Optimizing the number of instructions is not enough anymore, optimizations of the memory bandwidth is also required.

As expressed in the previous section, the memory stall cost is estimated by : $Memory\_accesses * Miss\_rate * Miss\_penalty$. Optimizing this cost requires to:

1. **Reduce the total number of memory accesses** ($Memory\_ accesses$)
   The memory accesses are of two types: the data and instruction memory accesses.

   Reducing the number of data memory accesses is performed by a technique called Integrated Layer Processing (ILP) [CT90]. The motivation of ILP came from the observation that data manipulations (checksum, presentation formatting, encryption) are very costly. The reason is that manipulating each byte of the messages, requires a memory *load* and a memory *store*, which are relatively slow operations on modern RISC architectures. Communication systems make extensive uses of data manipulations. Data manipulations are even more frequent in naive implementations of layered systems which code protocol layers by different functions communicating asynchronously. A data processed by the communication system is then processed sequentially by each of its layers. This implementation architecture multiplies data manipulations. The ILP technique consists of integrating all the manipulation functions of the different layers into an unique function. The motivation of layer and function integration is double: first, the number of memory accesses is reduced (the bytes are only read once, then all the data manipulation functions are pipelined, and the resulting data is written back in memory). Second, integration increases the data locality of the program, which improves the data cache behaviour. ILP has been extendively studied in the past few years [AP92, BD95, GPSV91]. [AP92] proposes a a technique to automize ILP functions implementation using a concept called the word filter. [BD95] shows that the ILP is not very easy to implement and that the gain achieved is less important than expected for various reasons. In this report, we are not considering ILP techniques. As detailed in section 2.1, the HIPPARCH compiler is composed of two tools: the first generates the control part of the protocol and the second the data manipulation functions. ILP techniques are considered in the second tool, whereas HIPPCO belongs to the first one.

   The reduction of the the number of instruction memory accesses is principally performed in HIPPCO by the techniques proposed in section 4.2 (Instruction Count Optimizations). Those optimizations reduce the number of instructions to execute and consequently the number of *fetch* and *load* instructions.

2. **Reduce the cost of each memory access** (*Miss_ rate*)
   Reducing the cost of memory accesses is performed by reducing the miss-rate through a better cache memory utilization.

   Most of current architectures use a hierarchical memory system to reduce the memory bottleneck. Typically memory systems are now composed of a main memory and several memory caches. Cache memories are small, high-speed buffer memory used to hold parts of the code which is frequently refered [HP90, Smi92]. Information (data or instructions) located in cache memory can be acceded in much less time than that located in main memory. Thus, the CPU spends less time waiting for instructions and operands to be fetched and/or stored. There is of course a trade-off between small, high miss rate caches with fast access time and large caches that may increase processor cycle time. Caches work better when the program exhibit significant locality. Temporal locality exists when a program references the same memory location several times in a short period. Caches exploit temporal locality by retaining recently referenced data. Spatial locality occurs when the program access memory locations close to those it has recently accessed. Caches exploit spatial locality by fetching multiple contiguous words (a cache block) whenever a miss occurs. Optimizing the memory access cost is then achieved by increasing those localities and reducing cache misses.

   Cache misses results from essentially two sources [GC90]:

   - interference between instructions competing for the same block in the cache
   - lack of room for some instructions. This lack of room is often caused by cache pollution. A cache can be polluted by instructions that are loaded and never executed. There are two causes that generates this pollution: First, if cache prefetching is used, instructions are loaded in the cache before they are referenced. Second, as cache blocks are usually larger than one instruction, some instructions are loaded and not executed.

   If the cache is large enough to hold the entire program, cache pollution will not deteriorate the program performance. However, if the cache is smaller than the program size, cache pollution should be reduced to achieve high performance.

Many researchers worked on the problem of restructuring programs to improve their memory performances. Most of those works use some profiling information to restructure the object code of the programs to achieve better code locality.

McFarling [McF89] proposed algorithms and heuristics that use profile data to guide in excluding some instructions from the cache to increase the performance of a direct mapped cache.

Pettis and Hansen [PH90] described techniques that pack the most frequently executed instructions and move the unfrequently executed instructions at the end of each function. Global analysis arranges functions to reduce inter-function cache conflicts.

In [GC90], the authors proposed to reduce instruction cache pollution by code repositioning. Contrary to the other works, no profiling data is used. The optimizations uses control flow and dependency analysis.

Profile guided code optimization is implemented in some of today's compilers. The C compiler for the Alpha CPU (distributed with OSF-1) features some of this profile guided optimizations. A study of the effect of profile guided optimization performed by commercial compiler (HP C compiler) on protocol codes (TCP and UDP) is presented in [SKP94]. This compiler implements the code restructuring algorithms described in [PH90]. Although the protocol throughput increased from 300 to 500KBit/s, the effect of those optimizations is limited. One reason is that the optimizations performed in this experiment only work at the procedure level and does not perform any cross-module optimizations.

The optimizations that we propose in section 4.4 are based on the techniques presented in [McF89, PH90, SKP94]. They differ however on several aspects:

- The analysis is performed using *prediction information* specified by the module designer and not from profiling data collected from a first implementation. This approach, which reduces the profiling collection process overhead, makes the compiling phase more efficient.

- *global (cross-module) optimizations* are performed. The optimizations are performed directly on the intermediate tree generated by the Esterel compiler. This compiler generates from a set of modules an integrated tree. The module abstraction does not exist anymore at this level of representation.

- the optimizations are performed at *a higher level* than the object code level . The goal of our optimizations is to generate good quality (language) code that can be optimally processed by the low-level compiler (C-compiler).

3.2.3 Cycle Per Instruction (CPI) Optimizations

Processors' speeds increased drastically over the last decade. The RISC architecture, which aims for both simplicity in hardware and synergy with compilers, is probably one of the most important factors of this improvement. In the RISC systems, each instruction is composed of micro-instructions which execute in a single cycle and are pipelined for better performance [HP90]. Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This is a technique used to make fast processors. A pipeline is often compared to an assembly line: each step in the pipeline completes a part of the instructions. An instruction can be implemented with five basic execution steps:

- IF- Instruction Fetch
- ID- Instruction Decode
- EX- Execution
- MEM- Memory access
- WB- Write back

Five instructions can be processed concurrently in the pipeline. One instruction is fed into the pipeline at the start of each clock cycle and moved one stage further with each clock cycle.

The low level compiler schedules the instructions to take advantage as much as possible of the pipelining. However there are situations, called *hazards*, that prevent the next instruction in instruction stream from executing during its designated cycle. For example, if an instruction to be fetched in the first stage is not in the cache, the processor stalls (waits) until the missing instruction is retrieved from the main memory. It is then important to have the code in the cache to fully take advantage of the pipelining behavior.

There are three classes of hazards:

- *Structural hazards* arise from resource conflicts
- *Data hazards* arise when a instruction depends on the result of a previous instruction
- *Control hazards* arise with instructions which change the Program Counter

The low-level compilers try to minimize those hazards with an optimum scheduling of the basic instructions. However low-level compiler do not have usually enough information on the program behavior to obtain the optimal performance. For example, when a branch instruction is executed the address of the next instruction depends on the test result and therefore can not be defined before the third step (instruction execution step). Three cycles are wasted for each branch of the program. Therefore to reduce this penalty, most of the processors make predictions: they make the assumption 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.

Low-level compilers usually keep the branch order of the original code, by lack of information on the actual program behavior. Some researchers ([SKP94, PH90]) proposes to use profile information to reorder the basic blocks such that the most heavily used basic blocks are placed in an order which favors the hardware branch prediction logic. Those optimization techniques are usually cumbersome to use. They require to generate a first version of the code, then to instrument it, to run it in order to get some profiling information and finally to recompile the program.

In section 4.4.2, we show how HIPPCO uses the prediction information to reorder the tests' outcome in order to improve pipelining.

4 HIPPCO Design

In this section, we describe the details of HIPPCO design. We firstable describe the common path identification phase, which is based on a Markov analysis. We then present the various instruction count and the memory stall optimizations applied to the common path.

4.1 Common Path Identification by Markov Analysis

4.1.1 General Presentation

For determining the execution frequency of different sections in a program, two alternative approaches are known from general compiler construction: code profiling [Cha91] and static analysis of the program code ([Bal93], [Wu94], [Wag94]). With code profiling, the program is executed using input data that is representative of the program's actual use. This has the advantage that a very accurate calculation of the probability of executing different program sections can be achieved. However, profiling is a time consuming activity since it requires selecting representative input data, and then modifying the program code according to the calculated execution probabilities. An additional difficulty with protocol code is that profiling requires the execution of two working program modules over a network connection.

Thus, for protocol code, static analysis of the program control flow graph seems to be the more practical solution. Again, static analysis may be realized in several ways: by simple loop detection ([Aho86]) or by a full Markov analysis of the control flow graph ([Ram65], [Tri82], [Wag94]). While simple loop detection allows to determine a superset of program sections that will be executed with high frequency, the optimizations we consider (such as test outcomes rescheduling or function inlining) require more exact information on the execution frequency. We therefore use a full Markov analysis of the control flow graph of an Esterel program [CH95].

The general idea of Markov analysis is to regard the control flow graph of a program as a finite Markov chain. The control flow graph is composed of nodes connected by arcs. It is constructed as follows: the nodes in the graph correspond to sequential sequences of assignement statements or function calls. In accordance with standard terminology of compiler construction [Aho86], these nodes are referred to as *basic blocks*. The last statement in each basic block is either an `if` statement or an `await` statement.

The arcs in the graph correspond to possible transfers of control between basic blocks. In an Esterel program, a control transfer can occur in two different ways: first, each `if` statement introduces two arcs, one leading to the basic block that is executed when the `if` test is true, and one leading to the basic block that is executed when the `if` test is false. Second, each state in the Esterel program has one arc for each input event in the automaton. Again following standard compiler terminology, the arcs will be referred to as *branches* in the following.

For the purpose of Markov analysis, the basic blocks of the control flow graph are referred to as *mstates* (for Markov states, not to be confused, at this point, with the states of an Esterel automaton), and the branches as *transitions*. Markov analysis requires that the probability of traversing each transition is known. In other words, the probability of traversing each branch in the control flow graph must be determined. This is called *branch prediction*.

In our design, branch prediction for protocol code can be performed at two different points in time. First, the writer of the building blocks defines the probabilities for branches leading code sections for error handling and similar infrequent events. However, for some branches the execution frequency depends on conditions that are only known when the building block is used in a particular protocol configuration. An example is the TCP protocol, where the authors of [Cla89] found that the branch probabilities vary when the protocol is used by a client or by a server module in a distributed application. In this case, the designer of the protocol library defines several different branch predictions, one for each case. Then, a *configuration variable* is introduced with one value for each case. This variable can be set by the user of the protocol library when configuring a particular protocol. In the example of the TCP code, the variable can be set to the values "Server" or "Client". Such variable is easily introduced by the means of keywords in the interface specification language.

The current version of the Esterel language does not allow to specify branch probabilities. We therefore implemented an extension to the Esterel compiler that allows specifying numerical probabilities for the input events of a module, and for the two outcomes of an `if` statement in an Esterel action.

Using this model, the execution frequency of the different parts in a configuration of protocol building blocks written in Esterel could work as follows. First, the `OC` code for the configuration is generated. Then the `Oc` code is split into basic blocks and branches in the way described above, and the branch probabilities are written into a transition matrix.

Let n be the number of basic blocks in control the flow graph, and let $P(i,j)$ be the probability that control passes from basic block i to basic block j. It can then be shown [Tri82] that the number of times, V_i, that each of the basic blocks is visited can be calculated by solving the following system of n linear equations:

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

where $\delta_{ij} = 1$ for $i = j$, and 0 otherwise.

4.1.2 HIPPCO Implementation

Applying this method directly on protocol code is not practical. In fact, the resulting matrix can become rather large. The amount of computation can be significantly reduced by making use of the Esterel code structure.

As described earlier, Esterel generates from a specification an automaton. This automaton is composed of a set of Esterel states, which are internally coded as trees. Therefore, there is no loop within a state. The execution probability of each state basic block can be determined by the hierarchical approach described hereafter:

1. Compute the state execution probabilities by a Markov analysis using the Esterel states as unit instead of the basic block (*mstates* = Esterel states)

2. Compute the probabilities of the basic blocks within each Esterel state, by multiplying the state probability with the probability of traversing each of the branches going from the root to the corresponding basic block.

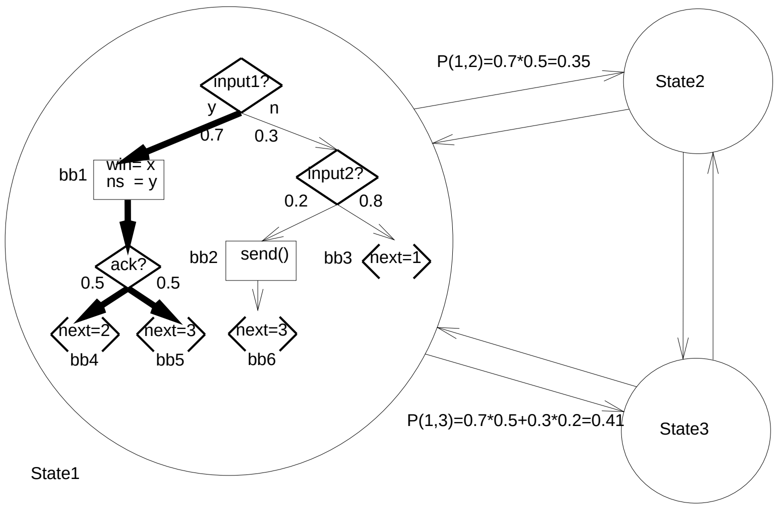

Figure 5: HIPPCO Markov Analysis

Once all the basic block probabilities have been computed, a heuristic should be used to identify those belonging to the common path. This approach has the major drawback that it requires the probability computations of all basic blocks, which is quite computing-intensive.

Generally, we expect the common path to be composed of a set of contiguous basic blocks. In fact, with the tree representation of the generated automaton, the execution control flows are contiguous, there are no loops. Based on this observation, we devise an algorithm that identifies the common path with a minimal amount of computations. The contiguity constraint makes the algorithm more stable and less sensitive to the branch prediction values. This algorithm has been implemented within HIPPCO for common path identification and it is composed of four steps:

1. *Computation of the Transition matrix*: (with $mstates =$ Esterel states) In this step, the transition probilities $P(i,j)$, which define the probabilities to go from $mstate_i$ to $mstate_j$ are defined. The probability $P(i,j)$ is computed by considering the tree representation of $mstate_i$ and selecting all the paths B_k^{ij} ($k = 1$ to M^{ij}, where M^{ij} is the number of paths in $mstate_k$ leading to

$mstate_j$). Each path B_k^{ij} is a sequence of L_k^{ij} branches. Each of these branches has a probability $q_{k,l}^{ij}$ ($l = 1$ to L_k^{ij}). $P(i,j)$ is defined by:

$$P(i,j) = \sum_{k=1}^{M^{ij}} (\prod_{l=1}^{L_k^{ij}} q_{k,l}^{ij})$$

for i and j going from 1 to N, where N is the number of states in the automaton. We call S this set of states.

In the example of figure 5, $P(1,3)$ is computed as follows: there are two paths leading to $mstate_3$ ($M^{13} = 2$). The first path has an execution probability of 0.35 (0.7*0.5) and the second of 0.06 (0.3*0.2). $P(1,3)$ is then equal to the sum of those two probabilities, thus 0.41.

2. *Computation of the state visit probabilities*:
   Once the transition Matrix, P, has been computed, the state visit probabilities are computed, by solving the following system of linear equations:

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

   for $j = 1$ to N, where N is the number of states in the automaton. In HIPPCO, this is performed using a Gaussian algorithm. The result vector V defines the visit count of each state.
   The probability PS_i of visiting $mstate_i$ ($i = 1$ to N), is then obtained by computing:

   $$PS_i = V_i / \sum_{k=1}^{N} V_k$$

3. *Identification of the common states*:
   The state visit probabilities, PS_i, are then used to identify the common states set CSS, which is the set of the states that are the "most frequently" visited of S. This identification is done with the following two-steps heuristic:

   (a) $mstate_i$ is automatically added to the CSS if: $PS_i > SPT$, where SPT is the state probability threshold above which a state is considered a "common" state. The value of this parameter has an impact on the size of the resulting code: if the value of SPT increases, less states will be automatically selected as "common" states. In the present HIPPCO implementation, SPT is set to 0.15 but this can dynamically be changed at compilation time. Let N_{as} be the total number of the those automatically selected states.

(b) Let MST be the minimum selection threshold i.e. the minimum fraction of selected states (if the value of MST increases, more states are added to the CSS).MST is set to 0.25 in the current HIPPCO implementation. The second step of the heuristic is as follows:

```
while $((\sum_{i \in CSS} PS_i) < MST)$
        add the $mstate_m$ in CSS s.t. $PS_m = max_{i \in (S-CSS)} PS_i$.
```

Consider the example of a four states-automaton with a state visit probability vector $V = \{0.1, 0.16, 0.03, 0.01\}$. The first step of the previous heuristic selects automatically $mstate_2$, which has a visit probability of 0.16. However as 0.16 is smaller than 0.25, the second step adds $mstate_1$ to the CSS. As $PS_1 + PS_2 \geq MST$ the algorithm stops. CSS is therefore composed of $mstate_1$ and $mstate_2$.

4. *Identification of the common branches*:
The common branches of each state of CCS are then selected. This is performed by going from the root to the leaves, for each state-tree of CCS, and cutting off all the branches with an execution probability smaller than a "common path" threshold CPT, commonly 0.5 in HIPPCO. The remaining branches constitute the common branches. In figure 5, the common branches of $mstate_1$ are shown by the thicker lines. The protocol common path is finally identified by the union of all the common branches.

In the following two subsections (4.2 and 4.3), we detail the Instruction Count optimizations that have been implemented in HIPPCO. Those optimizations that were applied to the common path are classified into two groups: the *Structural Optimizations* and the *Code Optimizations*.

4.2 Instruction Count Optimizations/ Structural Optimizations

In this subsection, we present the tree transformations performed by the *structural* optimizer. Some of those optimizations are These transformations improve the common path performance by decreasing the average number of instructions. These optimizations are applied to the control flow graph of the program. They are thus independent of the target language.

As described in section 2.2.2, an Esterel state-tree is composed of cascaded independent subtrees. Each of those subtrees processes one of the possible automaton inputs. If the automaton can receive I different inputs in state k, the tree of state k would be composed of I cascaded subtrees.

The number of instructions per input event i in state k is then the sum of two components: Cr_i^k, the cost to reach the subtree corresponding to the input, and Cp_i^k, the cost to actually process the input event.

The different inputs have different probabilities P_i^k. In fact, some events are more frequent than others. Typically *frames* from the network occur more often than the *alarm* signals. Therefore the average execution time of the state k can be modeled by the following formula:

$$C_{av}^k = \sum_{i=1}^{I} P_i^k * (Cr_i^k + Cp_i^k)$$

or

$$C_{av}^k = \sum_{i=1}^{I} P_i^k * Cr_i^k + \sum_{i=1}^{I} P_i^k * Cp_i^k$$

In the rest of this subsection, we propose a set of algorithms that reduce the average number of instructions by reducing each of those two components. The *Inputs Rescheduling* optimization reduces the first component. The Common Branches Extension optimization reduces the second component. The *Branches Pruning* optimization reduces both of them.

4.2.1 Inputs Rescheduling

In this paragraph, we propose a transformation that optimizes the first component $\sum_{i=1}^{I} P_i^k * Cr_i^k$. This component can be rewritten as

$$\sum_{i=1}^{I} P_i^k * N_i * C_{test}$$

where N_i is the rank of subtree i in the state tree. It can easily be demonstrated that this sum is minimal when the subtrees corresponding to the most frequent events are scheduling first in the tree, i.e. the subtrees are sorted in decreasing event frequencies.

The *Inputs Rescheduling* optimization is based on this observation. It consists in rescheduling 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. The input frequencies are defined from the information provided by the protocol designer. Figure 6 illustrates this optimization. Subtree A (i.e. handling the input event A) has an execution probability of 0.1, subtree B of 0.4 and subtree C of 0.5. The *Inputs Rescheduling* optimization restructures the initial tree, by rescheduling subtree C at the root, followed by subtree B and then subtree A.

A problem may appear when several inputs can activate the automaton simultaneously. The subtrees are not then completely independent. In this case, the partial order between those inputs should be preserved in the optimized tree otherwise the generated tree will not be equivalent to the initial one. However, the HIPPARCH execution environment was designed, such that only one input at a time can activated the protocol automaton. The subtrees are then all independent in our case.

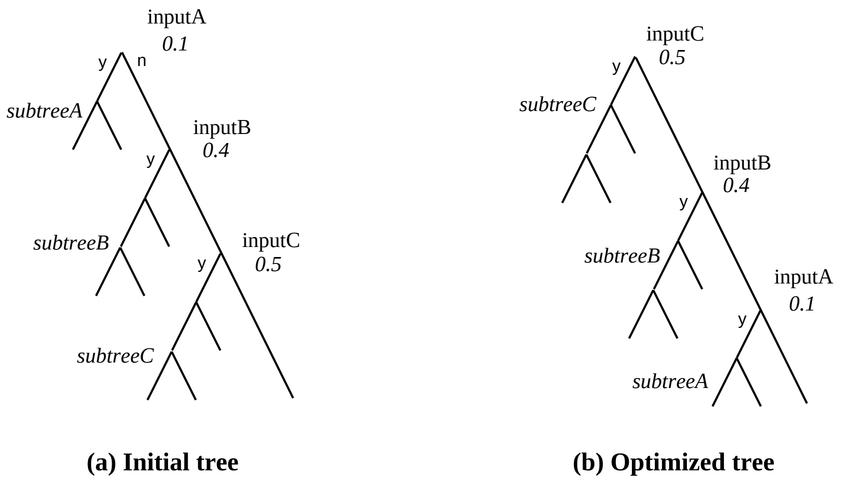

Figure 6: Inputs Rescheduling Optimization

4.2.2 Branches Pruning

In HIPPCO, the protocol designer can specify the probability of each input event and of each specification test's outcome. In some configurations, it may happen that some tests' outcomes are constant and that some inputs never occur. In this situation, the processing relative to those tests and inputs is unnecessary.

The *Branches Pruning* optimization configures a general protocol specification to a particular application. This optimization is performed in two steps:

1. all subtrees handling incoming events that are not possible for a given protocol configuration are removed,

2. within a subtree all tests with constant outcomes are detected and replaced by the predicted outcomes.

Figure 7 illustrates this optimization. Figure 7(a) displays the initial tree with the inputs and test outcomes' probabilities. Figure 7(b) shows the resulting tree when the subtrees handling impossible events have been removed. In this example, input A never occurs. Therefore the subtree handling input A is removed. The number of instructions to execute before reaching the subtree B is reduced. The presence test of input A is not performed anymore. Figure 7(c) shows the resulting tree when the tests within a subtree and with constant outcomes are removed. In this example, test1, is always correct. Test1 is then replaced by its backward outcome a_2. This reduces the proceesing cost of subtree B. In fact, by removing the unnecessary tests, the number of instructions to execute is reduced.

This optimization is performed on the initial tree representation, before any code size optimization is performed. It is performed everywhere, not only on the common path.

4.2.3 Common Branches Extension

In order to reduce the code size of the generated code, the tree representation of Esterel is transformed into a graph representation by HIPPCO [Cas95b]. However, we may want not to perform this code size optimization is some cases e.g. on the common path.

The *Common Branches Extension* optimization transforms the graph representation into a tree representation only on the common path. The generated common path is then straightlined. This optimization reduces the number of instructions on the common path (by removing the function calls overhead). It also improves i-cache (instruction-cache) behavior by improving the code locality of the common path.

The drawback of this optimization is that it increases the code size. However since the common path is only a small part of the total code, the code size increase is much smaller than the original Esterel code.

Figure 8 illustrates this optimization. In the initial tree, subtrees A, B, C and D are coded by a reference to an intermediary subtree with appropriate parameters. In the optimized tree, the subtree (or in general the parts of the subtrees) belonging to

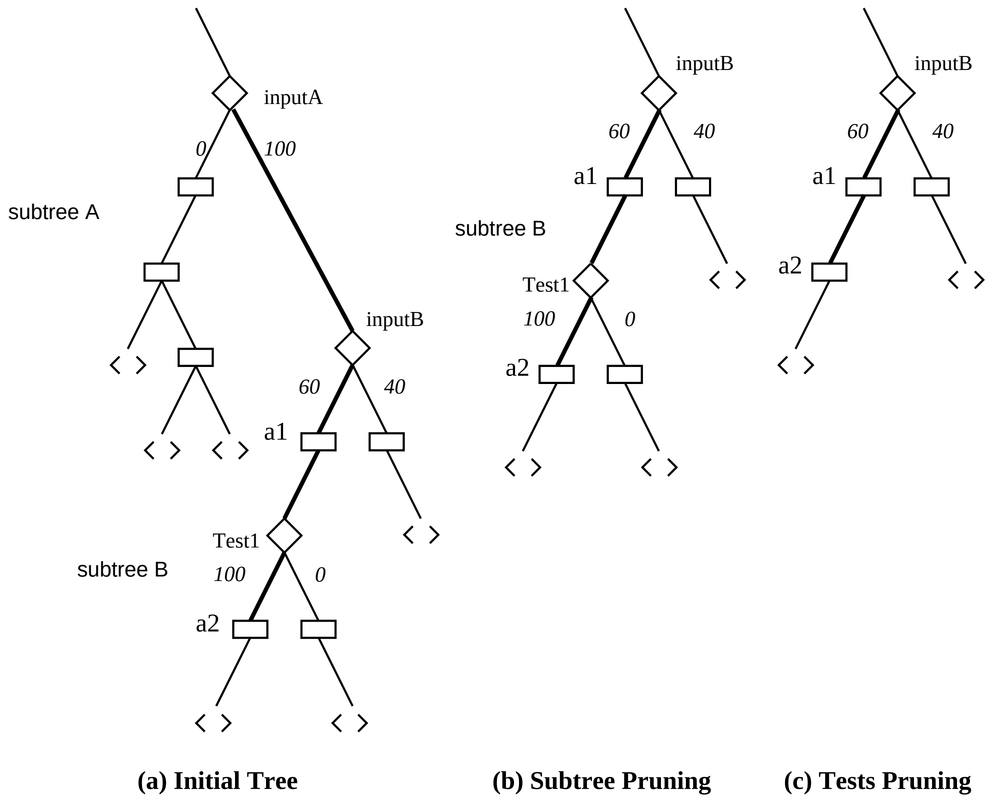

Figure 7: Branch Pruning Optimization

the common path (thicker line) is extended. As a result, subtree *A* can be acceded directly without indirection.

4.3 Instruction Counts Optimizations/ Code Optimizations

4.3.1 Inlining

Inlining consists of replacing a function call by a copy of the procedure body. Function inlining decreases execution time for two reasons: first, the overhead of the function call (instructions that save the status of the caller on the call stack) disappears. Second, by removing the function boundary, other optimizations can be performed by the low-level compiler (develop impact on i-cache). The penalty of inlining is a code size increase. Inlining is generally not a trivial optimization because

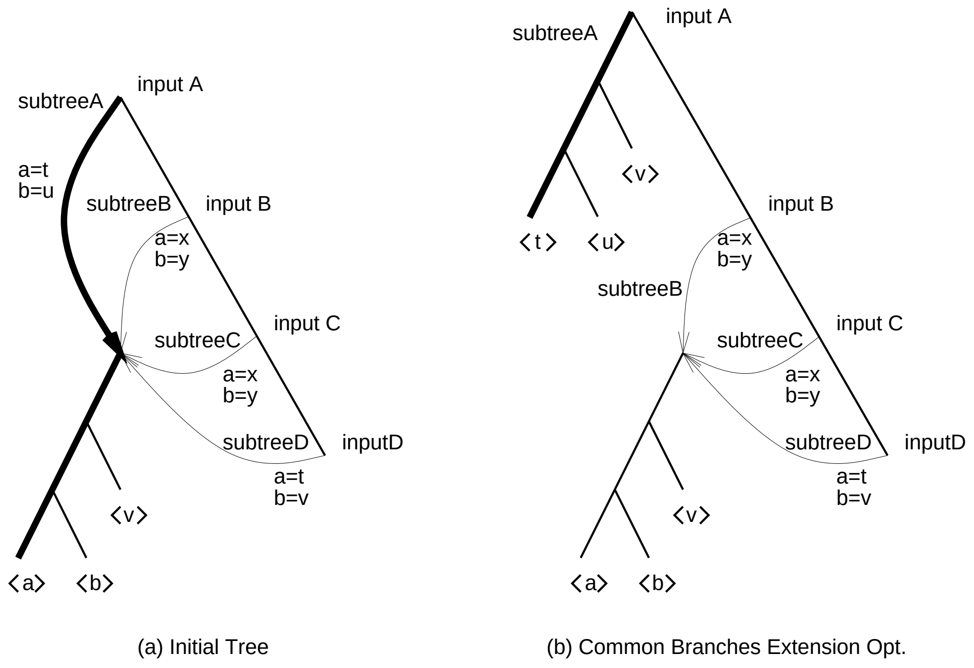

Figure 8: Common Branches Extension Optimization

it involves a code size/ code speed tradeoff. Inlining the wrong functions might have a bad impact on the i-cache performance and slow down program execution speed. The issue of how function inlining can be automated has been addressed several times in previous research []. If code size is not a direct constraint, inlining is beneficial in any of the following cases:

- 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,
- the function is executed very often.

Our common/uncommon path approach makes the choice of functions to inline easier. In HIPPCO, in addition to the functions that comply to one of the two first properties listed previously, all the functions that belong to the identified common path are automatically inlined whatever their size in conformity to the third property.

4.4 Memory Stall Optimizations

In this section, we present the tree transformations, performed by the *structural* optimizer, that reduce the memory stall cost. These transformations improve the common path performance by increasing code locality.

4.4.1 Outlining

Protocol common path code sizes are usually smaller than i-cache (typically 8KB). In this case, i-cache misses are not due the a large code size, but to a synchronization problem. Two instructions generate a cache conflict (and therefore a miss) if their addresses are spaced by a multiple of the cache size. Reducing the number of misses can then be performed by restructuring the code blocks such that the most frequent one will not conflict.

This scheduling problem is difficult to solve at the language level, because the unit the cache handles, the cache block, is not defined at the user level. It seems easier to solve this problem at the machine level by lower level compiler. However the problem of identifying the block frequencies is not trivial at this level.

Outlining can be applied as a simple solution to this scheduling problem. It compacts the frequently executed instructions and moves unfrequent code out of the mainline of execution. The resulting code features a very spatially compact common path which can entirely fit into the cache. Instruction blocks of the common path do not interact anymore.

As explained in [MPO95], this optimization has its limits when it is performed manually. Only conservative predictions can then be made. Performing this optimization automatically, on customized protocols with more aggressive predictions, will generate better results. HIPPCO identifies automatically the common path and compacts it by coding all uncommon subtrees or branches as functions (in fact, branches are outlined only if they are larger than the number of instructions that is required to perform a function call). The remaining code, the common path, is then very compact.

Figure 9 illustrates this optimization on a program tree. All uncommon branches such as act_1, act_4, act_{10} and act_9 , which can be composed of a sequence of instructions, are replaced by a reference (function call) to separated subtrees $ref_1()$, $ref_2()$, $ref_3()$ and $ref_4()$. The effect on the i-cache performance is shown in figure 10. The resulting tree is more compact and therefore has better icache behavior.

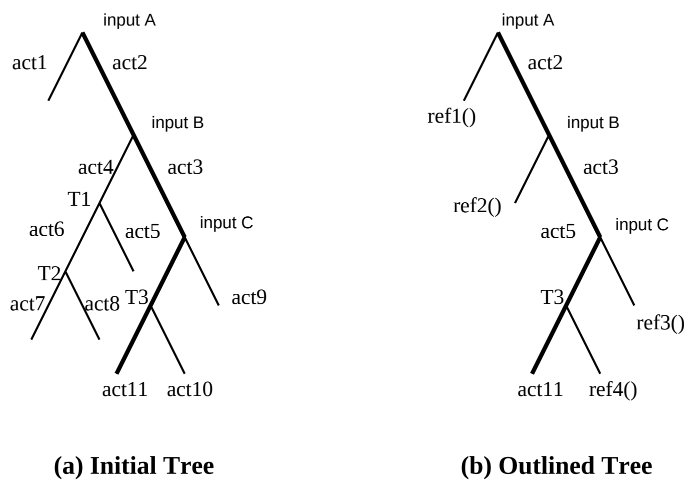

Figure 9: Outlining Optimization

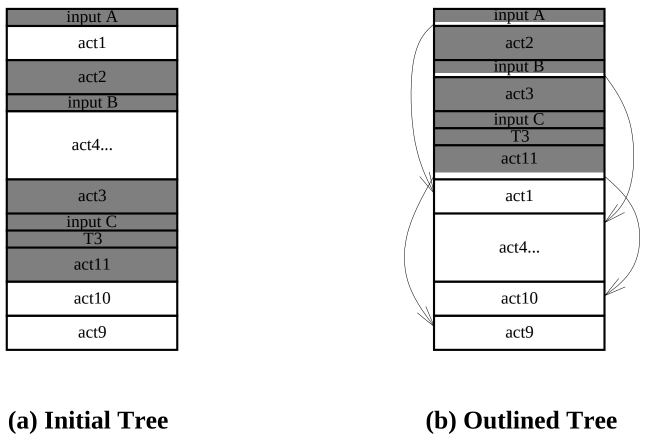

Figure 10: i-cache layout

4.4.2 Test Outcomes' Rescheduling

This optimization is based on two observations. The first is that basic blocks are very often generated in the order of the corresponding source code lines. The second is that the execution of the forward branch (else-branch) of a test requires a jump, whereas the code corresponding to the backward branch is streamlined. For example, the compiled code of the C program:

```
if (test)
 then action1;
 action2
```

will often be composed of the code of action1 with a conditional branch around it to handle the action2 code. The code of action2 is only reachable via a jump which break up the program pipeline. The execution of the forward branch leads to inefficient cache utilization and to wasted cycles due to processor stalls, a penalty which varies from a RISC processor to another.

The proposed optimization restructures the test nodes in the decision tree such that the most frequently outcomes are on the left branch of the tree [PH90]. 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. The test reverse operation is performed by negating the test and interchanging the backward and forward branch.

Because we are using a tree representation, this optimization also increases the i-cache utilization by structuring the code such that the common path is completely separated from the uncommon path. The i-cache utilization is then maximum, i-cache pollution is then minized. The benefit on the i-cache utilization is bigger if this optimization is conjunctly executed with the Common Branches Extension optimization, which ensures that the code on the common path is straighlined.

This optimization also increases program pipelining. In fact, as we described in section 3.2, processors usually makes predictions on the tests' outcome to increase the pipelining gain. They usually predict that the forward branch is not taken. This Rescheduling optimization increases the processor prediction correctness rate and consequently the pipelining gain.

Figure 11 illustrates this optimization on the outlined tree of the previous section. In the example, the most frequently outcome of input A is on its forward branch. The transformation converts then input A into $\neg$ input A and interchanges the backward and forward branches. The same transformation is performed on the test of input B. As a result of this transformation, all the tests along the common path have their most frequently used outcome on the backward branch (left branch).

Figure 12 shows the impact of this optimization on the i-cache layout. All the instructions of the common path (shaded) are contiguous in the memory layout. The uncommon path (included the reference to the uncommon subtrees) is mapped after it.

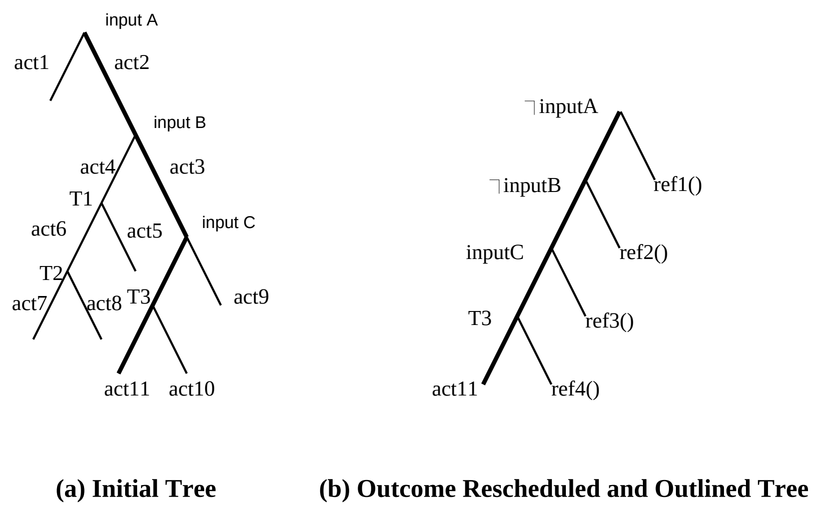

Figure 11: tree transformation

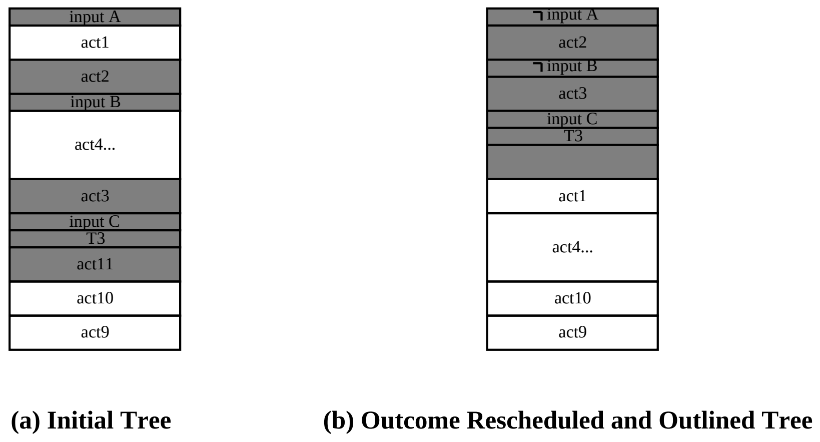

Figure 12: i-cache layout

The corresponding code transformation is shown by figure 13 (the shaded parts show the common path).

```
if(inputA)
 ref1();
else{
 act2;
 if(inputB){
   ref2();
 }
 else{
  act3;
  if(inputC){
   if(T3){
    act11;
   }
   else
    ref4();
 }
 else
  ref3();
 }
}
```

(a) Outlined Code

```
if(!inputA)
 act2;
 if(!inputB){
  act3;
  if(inputC){
   if(T3)
    act11;
   else
    ref4();
   }
   ref3();
  }
  else
  ref2();
 }
else
 ref1();
 }
}
```

(b) Outcome Rescheduled Code

Figure 13: Code Transformation

4.4.3 State Functions Rescheduling

This optimization is very similar to the outlining optimization, but works at the function level instead of the instruction level. HIPPCO generated code is composed of several functions. Each of them implements one of the automaton states. Those functions are scheduled in the same order in which they are defined by the automaton. The function implementing state 1 is just before the function coding state 2, which is just before the state 3 function and so on. This order is usually conserved by the low-level compilers.

The *State Functions Rescheduling* optimization increases the code locality, by rearranging, in the protocol code, those functions from the less to the most frequently executed. As a result the most executed states are grouped and the program code locality is improved. Better i-cache utilization is achieved.

The motivation for this optimization is then same than for the Outlining optimization. If all the most frequently executed functions are spatially grouped, they do not interact too much. The synchronization problem expressed in section 3.2 is then minimized. The benefit of the proposed optimization is bigger if it is associated with the Outlining optimization. In fact, in this case, the frequently executed state functions are smaller, because all the uncommon subtrees have been outlined, and have a larger probability to entirely fit in the instruction cache.

Also in general, the linker preserves the order in which the files are presented. For example, if two files are linked together , the basic blocks of the first file are ordered before the basic blocks of the second one in the final code. The order in which the files are linked has therefore a direct impact on the program memory layout and therefore on the i-cache performance.

To take advantage of this property, we schedule the most frequently visited state-functions at the bottom of the protocol code, and link it with the application code in the order: protocol-application. The memory gap between the protocol and the application, source of i-cache pollution, is then minimized.

Figure 14 illustrates this optimization. State 1 and state 3, which are the most frequently executed, are scheduled at the bottom of the program, increasing the code locality. Also the i-cache pollution between the protocol and the application is reduced by relocating state 4.

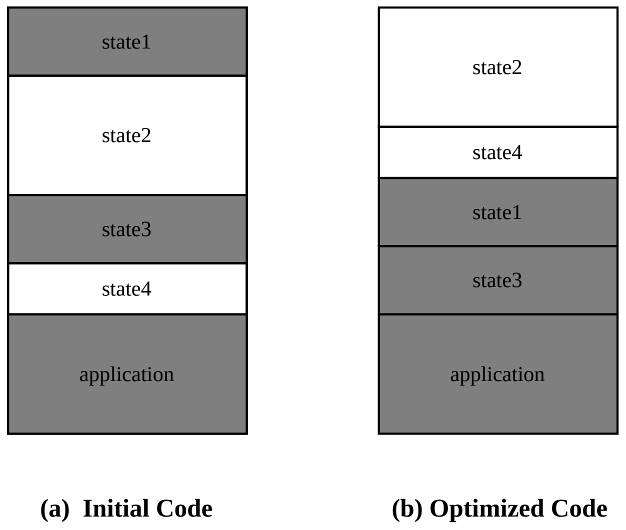

Figure 14: Memory Layout

5 Evaluation

In this section, we present an evaluation of the optimization techniques that has been described in the previous sections. The test case and the experimental environment are first described, the results are then carefully analysed.

5.1 Test Case: A File Transfer Application

The experiments were performed with a file transfer application. This application is composed of a client and a server. This application is very simple, it can be thought off as a backup application: the client connects itself to a server to transmit a 4MB file. The data-transfer is unidirectional: the server receives pure data and sends pure acknowledgment packets.

This application uses the TCP protocol. The protocol implementations used for our experiments were generated by HIPPCO with different levels of optimization. A brief description of the protocol specification is presented in the following section.

In order to evaluate the impact of our optimizations, we performed the same experiment with a user level implementation of the BSD-TCP protocol that we developed at INRIA. The comparison criteria that we adopted (instruction count, i-cache miss rate, number of cycles per instruction and the code size) will be detailed in the end of section 5.4.

In our experiments, no packet loss was generated. Thus, incoming packets follow the header prediction path. This ensures that we are really measuring the performance on the common paths of the different implementations.

Because it is always difficult to compare two different implementations of a same protocol, we focused our analysis (at least for the instruction count) on the input processing of the server side (receives pure data and send acknowledgements) of our application and compared instructions by instructions those implementations to make sure that the comparaisons were fair. This means that we took into account the cost of the control part of the protocol and not of the execution enviroment. This is similar to the approach adopted in [CJRS89].

5.2 The Protocol Specification

The protocol implementations used were automatically generated from an Esterel specification. In this section, we give a brief overview of the specification of the transport protocol and also of the customization choices that we made for the experiments.

The protocol that has been specified is the data transfer part of TCP (we omit the connection establishment and termination phases). In this section, we address very briefly how the building blocks were designed and combined to generate the required protocol. A more detailled description of this specification is presented in [CD94].

Reusability and flexibility were our main design goals here. The building blocks have been designed so that they are meaningful to the designers and so that changes in the protocol specification only induce local changes in the architecture and the code. We implemented this example incrementally in order to satisfy the modularity property that we are aiming for. We started from a simple protocol and added modules step by step until we accomplished all required functionalities. Following the precepts of Object Oriented Programming we followed the rule *one functionality-one module*. An overview of the whole system is shown in figure 15 . For clarity purposes, only the principal modules have been described and displayed.

Our data-transfer protocol is structured in three main concurrent modules :

The Send Module: composed of two concurrent submodules:

- The *User_ Input_ Handler* module is a four states 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 some data, it processes and copies them into an internal buffer. If the buffer is full, it goes to the 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 acknowledgement packet. If this packet frees up 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 states automaton. The first is the initialization state and the last the absorbing state. In the second state, the module can get activated by several signals, among them the *try_ to_ send* signal emitted by the previous module. The module then tries to 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

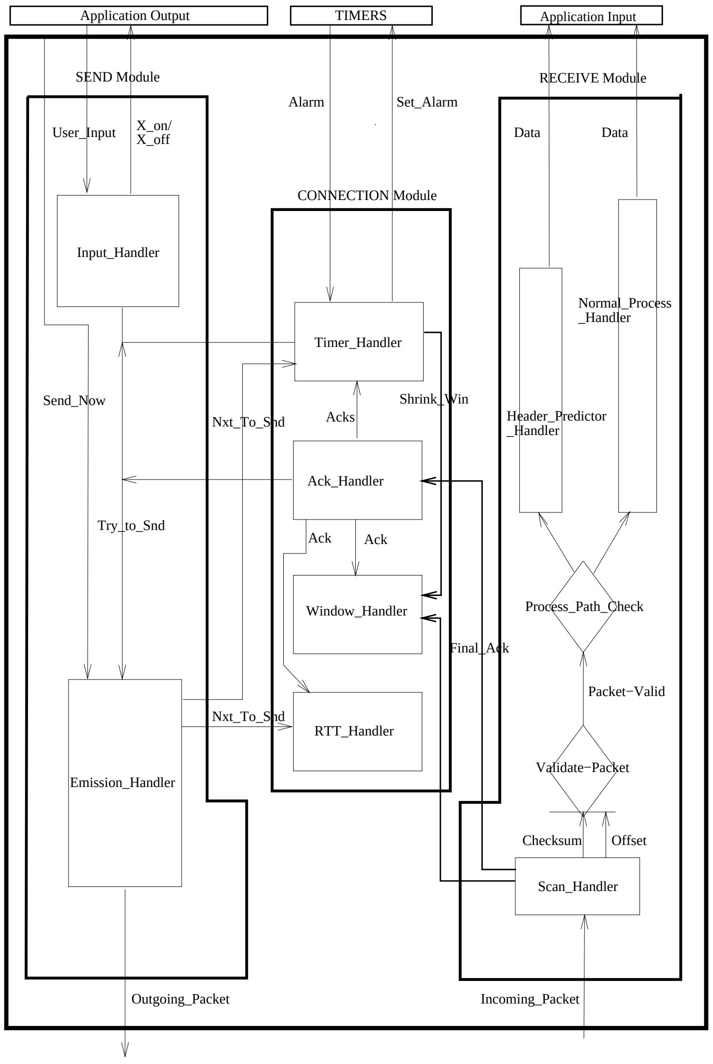

Figure 15: Protocol Implementation Overview

satisfied (this is used to acknowledge data for example). If the module decides to send packets, the checksum is performed[4] and the header is completed.

[4]In fact, the checksum calculation is a call to a void function, and the checksum verification returns 99% of times true without any computing as data manipulation functions are not performed as it will be detailed in the 5.3 subsection.

The Receive Module: this module processes the incoming packets. It is itself composed of several three states modules, which 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: this module is composed of the following submodules that are executed concurrently:

- The *RTT_Handler* is a three states 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 also a three states module which updates concurrently 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 (from the *Acknowledgment_Handler* module), 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 decreased 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 states 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.

- The *Timer_Handler* module manages the different timers. It is a five states automaton. In the state 2, no packet is on 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 acknowlegment signal or an *Alarm* signal. If an Acknowlegdment signal, which acknowledges all transiting packets, 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 reseting the timer with a larger value.

All the modules just described have been implemented in Esterel and compiled into an automaton.

5.3 The Customization Specification

As described in section 3.1, HIPPCO automatically configures the specification to meet the application requirements. This configuration requires some kind of information about the application behavior. Those informations are the:

- *External Input Frequencies*: an execution frequency is associated to each incoming event that can activate the protocol
- *Specification Test's Probabilities*: for all tests of the specification, a probability of occurence is associated to each test's outcome.

When *default profiling* is applied, those 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. An example is the default value for the result of the checksum operation: predicting that the checksum of the incoming packet is almost always valid is very reasonable (we set it to 99%).

These default values may be overwritten by the application designer if he has more information on the application behavior. When *application profiling* is performed, the values are derived from the specific application behavior. For example in our test-case, the server side only receives pure data packet: the probability to receive an acknowledgement packet is thus 0. Also since we want to optimize the processing of in-sequence data packet, which fit in the reception buffer, we set the probabilities of the tests' outcomes corresponding to this path to high values. For example the probability, in the reception module, that the sequence number is the expected one is set to 99%. The probability that the reception buffer is not full and has enough room to hold the incoming packet is also set to 99%.

More "aggressive" optimization may be performed in case of a *specialization* of an application profiled automaton. In this case, all branchs with zero probability will simply be cut from the automaton. The resulting code is not fully functional TCP. It has been specialized for the considered application.

Raw profiling corresponds to the case where precise values are very difficult to predict by the designer. The specification test probabilities are set to 50 % and external events are considered equiprobable. For example, in TCP, when a packet is sent (and no packets are on transit), a timer is set. When this packet is acknowledged the retransmission timer is reset. Predicting that each incoming acknowledgment packet will reset or not the pending timer is not easy.

In this section, we present the prediction paramaters that have been choosen for the profiling of the protocol at the server side. Those parameters have been used by HIPPCO to generate the customized implementations according to figure 16.

We remind that the server sends an initial packet with the file it requests. The client responds by sending the file in consecutive data packets. The server receives those packets and acknowledges them.

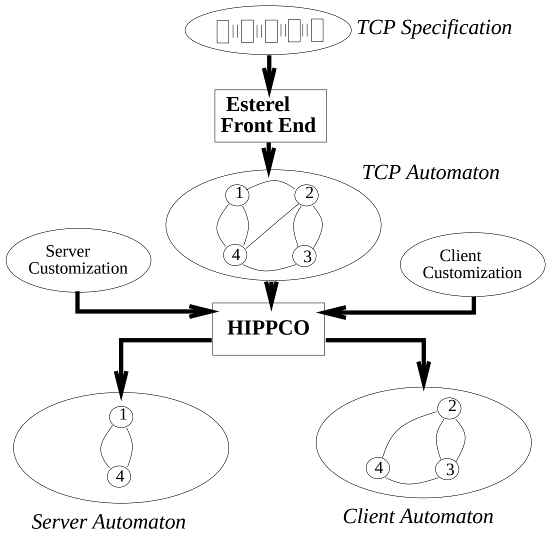

Figure 16: Protocol Customization

External Input Frequencies

Five input events can activated the protocol automaton:

- *Alarm*: is the input sent by an external timer to indicate that a timer has expired. In our tests, we consider that the application is running on top of an uncongested network, therefore the probability of packet loss is very low. The *Alarm* input probability was then set to 2%.

- *Input_ Frame*: is the input that contains incoming packets from the network. This is the type of input that is the most common in our test-case. Its input probability was set to 97%.
- *User_ Input*: is the input that contains data from the application. At the server side, no data is ever received from the application. This input probability was set to 0%.
- *End_ Of_ Input*: is the input that indicates that the application has no more data to send. Since we expect no data from the application, its probability was also set to 0%.
- *Close_ Connection*: is the input that is sent when the application decides to close the connection. This event only occurs once per connection. Its probability was set to 1%.

For comparaison purpose, at the client side, *Alarm* frequency was set to 2%, *Input_ Frame* to 45%, *User_ Input* to 45%, *End_ Of_ Input* to 1% and finally *Close_ - Connection* to 1%. The input execution frequency P_i of input i is specified by declaring the variable *input_ name_ Prob* and setting it to P_i. This has to be done for each possible input as illustrated in figure 17-a.

Specification Test's Frequencies

The identification of the common path also requires information on the outcomes' probability of the different specification's tests. Some of those are *default* probabilities (e.g checksum) and others are *application profiled* values. The specification of the *Checksum* module is presented in figure 17-b. The *default* assignment $P = 0.99$ indicates that the probability that the checksum is valid is 99%.

The implementation versions generated by HIPPCO are of four types:

- *raw profiling* versions: i.i.d probabilities are used for input events and test outcomes
- *default profiling* versions: the default predictions made by the module designer are used
- *application profiling* versions: the specific information on the application behavior is used
- *specialized* versions: the agressive optimizations described hereabove (e.g. branch pruning) are applied.

```
        /*    Server     */

int  User_Input_Prob            =  0.0
int  Input_Frame_Prob           =  0.97
int  Alarm_Prob                 =  0.02
int  End_Of_Input_Prob          =  0.0
int  Close_Connection_Prob      =  0.01

        /*    Client     */

int  User_Input_Prob            =  0.45
int  Input_Frame_Prob           =  0.45
int  Alarm_Prob                 =  0.02
int  End_Of_Input_Prob          =  0.01
int  Close_Connection_Prob      =  0.01
```

(a) Input Frequency

```
module CHECKSUM:

loop
 await Input_Frame
  if (TCPCksu(?Input_Frame) ==
       Get_Cksu(?Input_Frame))
  then
    P = 0.99;
     emit Frame_Valid;
  end_if
 end_await
end_loop
```

(b) Tests' Outcomes Frequency

Figure 17: Protocol Profiling Specification

5.4 Experimental Setup

To validate the proposed optimizations, a set of experiments has been performed. The primary goal of these experiments is to address the cost of running the generated protocols without the external overheads. To achieve this goal we removed all the data manipulations of the generated protocols: the packets are not actually checksummed, and they are never copied (pointers are used instead). In addition, we designed our own execution environment. This environment minimizes the number of context-switches, copies and memory allocations. In this environment, the protocol runs at the user-level and is directly linked with the application. The socket and network layers were removed: the network and socket interface were replaced by a buffer. A packet is sent by copying its pointers in this buffer. A packet is received by getting its pointer from this buffer. No data copy or manipulation is performed. In order to minimize the number of context-switches and the inter-process communication overheads, the server and the client were linked within a single Unix process and they communicate via the internal buffer previously described as illustrated in figure 18.

Beside the protocol implementation *Proto()*, HIPPCO generates for each automaton input an API. Those APIs are implemented by macros, with two arguments: the connection control block address and a pointer on the input value. They are used to activate the protocol automaton. For example, the application sends data *d* to the connection *cb* by calling the macro *User_ Input(pd, pcb)*, where *pcb* and *pd* are pointers on *cb* and *d*. For the specification described in section 5.2, five APIs have been defined: *User_ Input(), Input_ Frame(), Alarm(), End_ Of_ Input()* and *Close_ Connection()*.

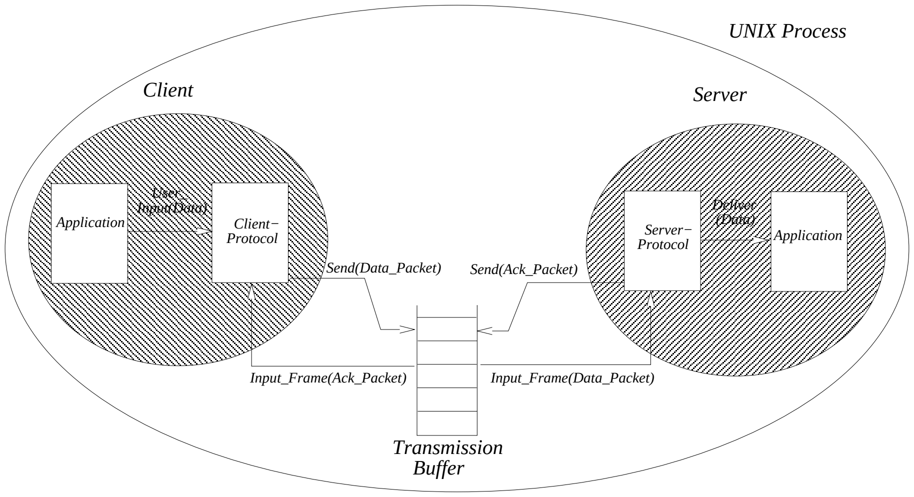

Figure 18: Execution Environment

This environment is executed in three steps:

1. At the client side, the application sends the data *d* by calling the protocol API *User_ Input(pd, pcb)*. The protocol processes the data and possibly builds some packets that are copied in the transmission buffer. The transmission buffer simulates the network.

2. The server side is then activated. It checks whether some packets are in the transmission buffer. If that is the case the corresponding packets are processed by the protocol and the data are handed to the server-application. Acknowledgement packets are possibly formed and copied in the transmission buffer.

3. The client then verified whether it has received some acknowlegment packets. If that is the case, there are processed and some more data packets are possibly formed.

Those three steps are repeated until no more data have to be send by the client-application. A signal *End_ of_ Input* is then sent to the protocol, which finishes to send the data stored in its internal buffer.

This environment has been used to evaluate the effect of the different optimization techniques described in this document, on the protocol performance. Four standard metrics have been used: the number of instructions executed, the i-cache miss-rate and the pipeline behavior (the number of cycles per instruction) and the code size. As described in section 3.2, the three first metrics are necessary to evaluate the execution time of the protocol. The fourth, the code size, is interesting to study the effect of the speed optimizations on the code size.

The experimentations 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 8KB each, an unified 2MB second-level cache and 64MB of main memory. All caches are direct-mapped and use 32B cache-blocks.

The experimental results were obtained using the ATOM tools of DEC. ATOM provides a rich set of tool building primitives that can be used to analyse a program behavior. Basic tools such as tracing, instructions profiling and instruction and data address tracing exist.

5.5 Analysis of the Results

In this section, we present and analyse the gain achieved by the different optimizations proposed in section 4.2. We first analyse the generated automaton and the results of the common path identification algorithms. We then analyse separatly the gain achieved by each optimization on the instruction counts, the i-cache miss-rate and the pipelining stalls. Although we do not present in this report the code size optimization techniques implemented in HIPPCO in detail, we present some of these these optimizations resutls in subsection 5.5.6.

5.5.1 The Generated Automaton

The Esterel compiler generates from the parallel specification a sequential automaton. The maximum size of the automaton is the product of the size of all parallel modules that composes it. The Esterel compiler reduces it to a minimal automaton by removing all unreachable states. In our example, the maximum size of the automaton (if all the modules were completely independent) would be 4723920 states. But because the modules are not completely independent, the Esterel compiler reduces it to 21 states.

HIPPCO reduces furthermore the number of states to 11. In fact, the Esterel compiler generates sometimes intermediary states that are used to solve causality problem. Those states are sometimes identical to others. For example in our TCP specification: they are just buffering states before exiting the automaton. HIPPCO detects those identical states and removes them. The HIPPCO generated automaton is presented in figure 19. In this figure, only the transitions that cause the automaton to change state are shown.

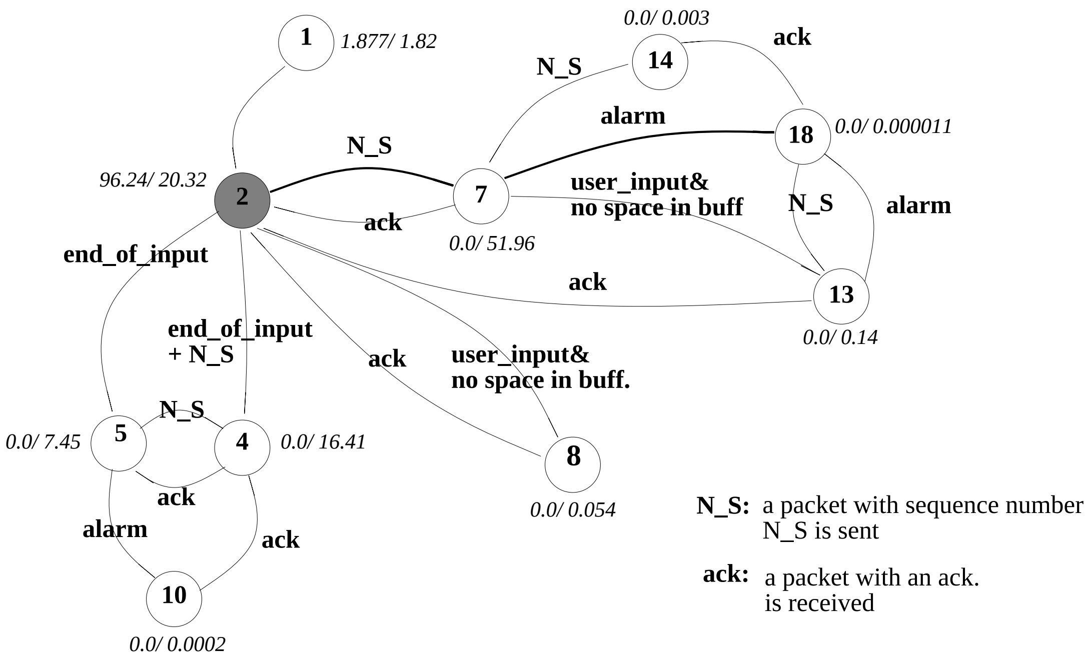

Figure 19: Generated Automaton

This figure also displays the states frequencies obtained with the common path identification algorithm. The frequencies are displayed in italic next to the corresponding state. The first value indicates the state probability for the server automaton, the second value indicates the state probability for the client automaton. Those frequencies are compared with the raw and default situations. The results are displayed in table 1.

The first column displays the states number as they are generated by the Esterel compiler. The following rows, display the state probability values with *raw*, *default*, server and client profiles.

It is interesting to observe that when prediction is used, some states are unreachable (states 4, 5, 7, 8, 10, 13, 14 and 18 for the server). Those states are never

state	raw	default	server profile	client profile
0	14.81	1.72	1.877	1.82
1	14.81	1.72	1.877	1.82
2	23.86	28.54	96.24	20.32
4	25.34	13.7	0.0	16.41
5	23.7	11.34	0.0	7.45
7	1.17	41.76	0.0	51.96
8	0.915	0.143	0.0	0.054
10	4.18	0.076	0.0	0.0002
13	0.048	0.217	0.0	0.14
14	0.118	0.75	0.0	0.003
18	0.0093	0.004	0.0	0.000011

Table 1: State Visit Probabilities (in%)

visited, because some event occurences (and therefore transitions) are impossible in this case.

When specialization is performed at the server side, the number of states is reduced to 3 and state 2 is the only common state. It implements the reception of pure data and transmission of acknowledgments. For comparaison purpose, we also present the states count for the client side. In this case, all the states are reachable (because no prediction value have been set to zero), but some of them have a very low execution probability (states 10, 14 et 18). Three states have a much higher execution probability than the others: state 7 (51.96%), state 2 (20.32%) and state 4 (16.42%). State 7 corresponds to the state implementing the normal case: a packet has been sent on the network, data from the application can be received, and the retransmission timer is set. State 2 is very similar to state 7, but all data have been acknowledged and the timer is reset. In state 4, the application do not send any more data (the *End_ Of_ Input* signal has been received), but the protocol keeps sending the data present in its internal buffer. The other states implements uncommon cases: internal buffer full, packet retransmission ,etc.

Also it is interesting to note that although the *raw* profiling (all predictions set to 50%) gives pretty inacurrate results, the *default* profiling, which uses the default prediction values, gives results that are pretty close to the one obtained with specific application profiling. In fact, when the prediction default values are used, the states

2, 4, and 7 are the most frequently visited states with a visit probability of 28.54%, 13.7% and 41.76%. When application profiling is performed at the client, state 2, 4 and 7 are also detected as the most frequently visited states with a probability of 20.32%, 16.41% and 51.96%. At the server side, state 2 is identified as the most frequently visited state with a probability of 96%.

Those results are important. They show that the common states identification algorithm works also with very rough predictions. Application profiling, which provides more accurate predictions, generates more precise solutions.

When no prediction is used, a set of states, which are candidates to be frequently visited are identified (states 0, 1, 2, 4, 5, 7, 8 and 10). When the prediction default values are used, the results are more precise (states 2, 4, 5 and 7). And finally when application profiling is performed, only the states corresponding to the application are selected (states 2, 4 and 7 for the client and state 2 for the server).

In the following two subsections, we analyse the instruction count gain of the proposed optimizations. The next subsection compares the performance of the fully optimized HIPPCO implementation with the handcoded BSD TCP. The subsection which comes after details the gain of each of the HIPPCO optimizations independently.

5.5.2 General comparison: HIPPCO vs BSD

The number of instructions necessary to process an incoming packet (containing pure data) and of sending pure acknowledgements for the different implementations are presented in table 2 and plotted in figure 20. The first column gives the number of instructions for the input processing, I_c, the second for the output processing, O_c, and the third column, T_c displays the average number of instructions per packet processing. The values of the third column are obtained by dividing the total number of instructions executed, by the number of protocol calls. Since usually in TCP, an acknowledgement packet is sent after 2 incoming packets, we have approximatively: $T_c = (2 * I_c + O_c)/2$.

Those Instruction counts were measured for the BSD implementation (with and without header prediction), for the initial Esterel code and finally for the code automatically generated and optimized by HIPPCO (with all optimizations enabled). For the HIPPCO implementations, we consider here three cases: (1) the code generated by HIPPCO when default profiling is used (HIPPCO), (2) the application profiled implementation (Profiled HIPPCO) and (3) the version that has been specialized (Specialized HIPPCO). The difference between the profiled and specialized

	I_c (input)	O_c (output)	T_c (average)
BSD	186	422	397
BSD without head.pred.	508	422	716
Esterel (array coding)	/	/	2133
HIPPCO	166	168	220
Profiled HIPPCO	143	147	204
Specialized HIPPCO	139	142	193

Table 2: Average Number of instructions/call

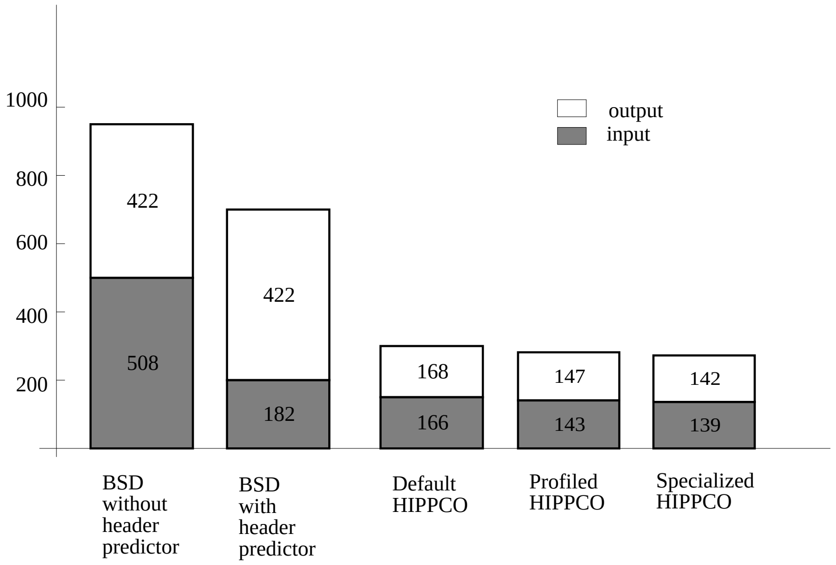

Figure 20: Instruction Counts

versions is that the first one is still a fully functional implementation, whereas the second one has been specialized for an unidirectional transfer.

There are several interesting observations from those results. The first one is that the preconceived idea that Formal Description Techniques (FDT) tools generates poor quality code seems to be also true for the original Esterel compiler (approximatively seven times slower than the BSD implementation). The second observation is that with the proper optimizations, automatically generated codes can be at least as fast and even faster (in term of instruction count) than handcoded codes. In fact, the code generated by HIPPCO is faster than the BSD code whatever specialization is performed or not. For the input processing, HIPPCO code without specialization requires 20 (11%) less instructions than the BSD inplementation and 342 (67%) less instructions than the BSD without header prediction implementation. Morever, HIPPCO code with specializartion is even faster. In this case, the input processing requires 47 (25%)less instructions than the BSD inplementation and 369 (73%) less instructions than the BSD without header prediction implementation.

The third observation is that the BSD-header prediction is quite successfull in speeding up the processing of input packet (it requires 2.73 times less instructions than without header prediction). However the output processing stays poorly implemented (an output prediction should maybe be implemented).

With default profiling information, the code generated by HIPPCO is slighty faster than the BSD implementation. The code of HIPPCO requires in this case 11% (20 instructions) less instructions because of its structure: each state is implemented by a different function, some tests that are necessary for the BSD implementation are not for the HIPPCO implementation. For example, testing that the current state is TCP_ESTABLISHED or that the protocol is not in a retransmitting state is not necessary in HIPPCO, this is implicit. The function processing the incoming packet is the one implementing the common state, which is by definition (or at least in our test) the state in which the connection is established and no retransmission is pending.

When application profiling is performed, the number of instructions is reduced by 23% (43 instructions). This reduction is due to the optimizations which transform the protocol structure in order to fit better that particular application.

When specialization is performed, the code generated is even faster (25% less instructions that the BSD implementation). In this case, more aggressive predictions are performed and some unnecessary tests and functionalities are removed from the common path. For example, in our test, the server only receives pure data and no acknowledgement, therefore branches which process the acknowledgement can be removed from the server common path. The program tree is then smaller and therefore faster. The BSD and HIPPCO header prediction codes are presented in appendixes A and B.

The code to process outgoing acknowledgements is also faster in HIPPCO. The gain compared to BSD is even larger, because the *tcp_ ouput()* function of the BSD implementation does not use any prediction. In HIPPCO (when specialization is performed), the code is specialized for the processing of acknowledgments, whereas in the BSD code, the *tcp_ output* function is more general, because it should process different kinds of outputs (data, acks, retransmission). Also in HIPPCO, the tree representation of the common path generates straightlined code without jumps, which is faster.

5.5.3 Optimizations Gain: Detailed Results

In this subsection, we analyse the gain of each optimization technique implemented in HIPPCO. In the presented results, we associate to each HIPPCO generated implementation a name. Each of thoses names is formed by the combination of the HIPPCO options that has been used for that particular implementation. The HIPPCO options refers to the different possible optimizations. Option O refers to the Esterel tree sharing optimization, S to the code size optimization, E to the Common Branches Extension optimization, P to the function inlining and states pruning (explained later) optimization, U to the Outlining optimization, T to the Branches Pruning optimization, I to the Inputs Rescheduling optimization and finally[5] C to the Tests' Outcomes Rescheduling optimization. For example, the implementation OSEP has been generated by HIPPCO with the options O, S, E and P enabled. The O opimization already existed in Esterel. All other optimizations were implemented within HIPPCO.

Table 3 presents the average number of instructions required by HIPPCO generated code with different levels of optimizations. This average number is computed by dividing the total number of instructions required to process the 4MB by the number of protocol function calls. The (weighted) average is computed according to the T_c formula defined in the previous section.

The highest 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 results was expected. In fact, in the interpreted code, each Esterel action is implemented by a function. A state is then defined by the set of functions, which are called by the interpreter at the execution. Intereprted coding is thus very slow, it requires one function call per elementary action. Inlining the control reduces the number of function calls and therefore the number of instructions required. However, applying this optimization (already supported in Esterel) resulted in a size explosion in our example. We used the Esterel (O) optimization which reduces the code size by sharing identical subtrees in the automaton.

[5] More specific notation details will be given later.

	instructions
Esterel (interpreted code)	2133
Esterel (compiled code + O)	421
HIPPCO (OS)	465
HIPPCO (OSP)	423
HIPPCO (OSEP)	206
HIPPCO (OSEPU)	206
HIPPCO (OSEPUT)	195
HIPPCO (OSEPUTI)	193
HIPPCO (OSEPUTI) raw	
$CPT \geq 0.5$	219.5
$CPT > 0.5$	574

Table 3: Average Number of instructions per call

Adding the code size optimizations (OS) (not presented in this report) increases the instruction counts by 44 instructions (10%). In fact, the code size optimizations share identical trees, which introduces some indirections.

The third optimization (OSP), which inline all functions on the common paths, reduces the number of instructions by 9% (42 instructions).

The fourth optimization (OSEP), which extends the branches on the common path in order to generate straighlined code (i.e. without function calls), reduces the number of instructions by 51% (from 423 to 206 instructions).

The fifth optimization (OSEPU) does not reduce the number of instructions. This optimization outlines uncommon branches or in other words implements uncommon branches as function calls. It therefore does not affect the number of instructions executed on the common path.

The sixth optimization (OSEPUT), which trims the branches of the tree with an execution probability of zero, reduces the number of instructions by 5.4% (11 instructions).

And finally, the seventh optimization (OSEPUTI), which reschedules the input processing subtrees according to the probability of each input events 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 gain achieved by each optimization separately may be considered small, but the combination of all of them leads to a gain of 91% or of 60% if we considered only the compiled code versions of the protocol.

When raw profiling is performed, two cases can be considered. In the first one, all branches with a probability equal or larger than 50 % are considered on the common path ($CPT \geq 0.5$). In the second one, only the branches with an execution probability strictly larger than 50 are considered on the common path ($CPT > 0.5$). This is an important point specially when raw profiling is performed because a lot of probability variables are set to 50%. In the first case the gain achieved is of 73% (1559 instructions). It is equal to 88.8% (1894 instructions) in the second case. The smaller the CPT (common path identification threshold) is, the better the average code performance (in term of instruction number) will be and the larger the code size will be. This is the classic code size/ code speed tradeoff problem.

5.5.4 I-cache Performance

In this section, we analyse the impact of the different optimization techniques on the i-cache behavior. We compare the miss rate (misses/references) achieved with the different optimizations. We also compare those results with the i-cache performance of the BSD implementation.

We performed for each implementation four experiments. We measured the i-cache behavior of the code composed of the protocol and the application. We then measured the i-cache behavior of the protocol code only. This last experimentation is performed to evaluate the i-cache behavior of the generated protocols, independently of the running application. It gives a better evaluation of the optimizations impact on the i-cache. Those two experiments have been repeated with caches of size 8KB and 4KB to study the effect of the cache-size.

The results are presented in table 4 and 5. The column *references* refers to the total number of memory accesses performed by the program. The colum *misses* displays the number of memory misses (the instruction to be accessed is not in the i-cache). And *rate* is the miss rate. It is equal to *misses* divided by *references* expressed in percentage.

A first observation is that the miss rates measured with the protocol linked to the application are in general larger than with the protocol alone. The first reason is that in the first case, the code is larger and the memory conflicts are more frequent. The second reason is that the proposed optimizations were only performed on the protocol code and therefore some i-cache pollution is added by the application. However the

	8KB			*4KB*		
	references	*misses*	*rate*	*reference*	*misses*	*rate*
BSD	1099521	73051	6.64	1087362	196130	18
Esterel (interpreted code)	323102	26620	8.2	323042	417341	12.91
Esterel (compiled code + O)	73044	10067	13	73044	16061	22
HIPPCO (OS)	121955	12565	10.3	121951	36676	30
HIPPCO (OSP)	105950	558	0.5	105950	8673	8.1
HIPPCO (OSEP)	66947	555	0.83	66947	6667	10
HIPPCO (OSEPU)	66972	571	0.85	66968	2690	4
HIPPCO (OSEPUCb)	66972	565	0.84	66972	688	1
HIPPCO (OSEPUCf)	66972	567	0.84	66968	4691	7
HIPPCO (OSEPUCbT)	62948	2541	4	62948	2648	4.2
HIPPCO (OSEPUCfT	62942	552	0.87	62942	677	1
HIPPCO (OSEPUCbT, raw profiling)						
$CPT \geq 0.5$	70950	560	0.78	70956	679	0.95
$CPT > 0.5$	126955	4582	3.6	126954	26689	21

Table 4: I-cache behavior (application + protocol)

effect of the optimizations on the i-cache behavior are identical in those two cases, they only differ by a scale factor. In this section, we only detail the results obtained with the protocol code. The other results are presented for comparaison purposes.

An other observation is that the i-cache misses are higher, when the i-cache size is smaller. This is an expected result. Ideally the i-cache miss rate converges to zero when the i-cache is large enough to hold the entire program.

The code generated by the pure Esterel has a pretty high cache miss rate (3.4 and 8.1%), however this rate gets even larger when the protocol control flow is inlined (26.11 and 32.6%). The reason is that the first case corresponds to an interpreted version. The protocol code (text part) is small (10KB) and therefore generated less conflicts. In the second case, the protocol code is very large (86 KB) and the number of memory conflicts is larger.

With the code size optimization (OS), the i-cache miss rate drops drastically to 4 and 22%. The reason is that the code is smaller and more compact.

States Pruning [Cas95b] is a code size optimization removing all unreachable states. When *Inlining* and *States Pruning* (OSP) optimizations are applied, the i-cache miss rate goes down to 0.08 and 4.7%. This has mainly two explanations.

	8KB			*4KB*		
	references	*misses*	*rate*	*references*	*misses*	*rate*
BSD	197380	458	0.23	197433	33933	17.18
Esterel (interpreted code)	298280	10133	3.4	298280	24140	8.1
Esterel (compiled code + O)	93947	24538	26.11	93947	30663	32.6
HIPPCO (OS)	101048	4082	4	101048	23067	22
HIPPCO (OSP)	85043	68	0.08	85043	4067	4.7
HIPPCO (OSEP)	46044	63	0.13	46044	4059	8.8
HIPPCO (OSEPU)	46049	68	0.147	46049	68	0.147
HIPPCO (OSEPUCb)	46049	60	0.13	46049	60	0.13
HIPPCO (OSEPUCf)	46049	65	0.14	46049	65	0.1412
HIPPCO (OSEPUCbT)	42029	40	0.095	42029	40	0.095
HIPPCO (OSEPUCfT)	42029	43	0.1	42029	44	0.1
HIPPCO (OSEPUCbT, raw profiling)						
$CPT \geq 0.5$	50049	61	0.12	50049	63	0.12
$CPT > 0.5$	98048	81	0.08	98048	16068	16

Table 5: I-cache behavior (protocol only)

The first one is that the code size is reduced from 33 to 7.12KB. The second one is that the generated code is streamlined on the common path and therefore utilize the i-cache more efficiently.

With the *Common Branches Extension* optimization (OSEP), the i-cache miss rate increases, as expected, to 0.13 and 8.8%. The *Common Branches Extension* optimization inlines all the subtrees that are encountred on the common path. The miss rate increase is mainly due to the decrease of the number of instruction references. The number of misses staying the same, the miss rate decreases.

By moving all the uncommon branches out of the mainstream code (OSEPU), the miss rate decreases from 8.8% to 0.147% with a cache size of 4KB. The reasons of this drastic improvement were given in section 3.2. The common path code is more compact and fits entirely in the i-cache. The gain measured with a cache size of 8KB is null. The main reason is that with all the performed optimizations, the code size of the common path got so small that it entirely fits in the i-cache. The effect of the *Common Branches Extension* optimization is not visible anymore.

To study the effect of the *Tests' Outcomes Rescheduling*, we performed two experiments. In the first one (OSEPUCf), we rescheduled all tests' outcomes such that the most frequently outcomes correspond to the forward branches (we call it forward rescheduling designated by Cf). In the second one (OSEPUCb), we did the opposite: we rescheduled all tests' outcomes such that the most frequently outcomes correspond to the backward branches (we call it backward rescheduling designated by Cb). The effects of the optimization are not perceptible with the experiments on the protocol alone. At this level of the test, the protocol versions have been already so well optimized that no optimization benefit can be noticed. Some improvements were observed on the experiments with the application and a cache size of 4KB. As we expected, when the forward rescheduling optimization is performed, the miss rate decreases from 4% to 1%. With backward rescheduling, it increases to 7%. The reasons of this effect are detailled in section 3.2. As a result of the forward rescheduling optimization, all the tests along the common path have their most frequently executed outcome on the forward branch and the common path is therefore streamlined. With the backward rescheduling optimization, all the tests along the common path have their most frequently used outcome on the backward branch and the common path code is polluted by the uncommon branches (figure 12). The miss-rate increases logically.

The *Branches Pruning* (OSEPUCbT) optimization has a small effect on the i-cache performance. With this optimization, the miss rate decreases from 0.13% to 0.095%. This improvement is mainly due to the reduction of instructions on the common path. The number of references and therefore of misses is smaller.

The effect of the *States' Functions Rescheduling* optimization is very small in our example. After the *States Pruning* optimization is performed only two states are left. This gives small optimization opportinuties. Therefore to evaluate the performances of this optimization, we disabled the *States Pruning* optimization and compare the results obtained before and after the *States' Functions Rescheduling* optimization (table 6). The miss rate decreases from 3.8% to 0.87% with a cache size of 8KB, and from 7% to 4% with a cache size of 4KB. The number of memory references is identical in all cases. The gain is achieved by a reduction of misses due to less cache pollution in conformity with the explanations of section 3.2.

Some measures were also performed with raw profiling. As a lot of prediction values are equal to 50%, the optimizations performed quite differently according to the Common Branch identification threshold (CPT). With $CPT = 0.5$, the detected common path is larger and the impact of the optimizations is bigger. The miss rate is equal to 0.12% which is very close to the results obtained when specialization was performed (the best result was 0.095%). With CPT 0.51, the code size is smaller,

	8KB			*4KB*		
	references	*misses*	*rate*	*references*	*misses*	*rate*
OSEPU	66980	2572	3.8	66980	4691	7
OSEPU+ state functions rescheduling	66980	563	0.87	66980	2698	4

Table 6: I-cache behavior without the *States Pruning* optimization (application + protocol)

but the performances obtained are not very good. The miss rate is equal to 0.08% with a cache of 8KB and increases to 16% with a cache of 4KB.

Some observations can be made from those results. The first one is that the code size of a program does not have a direct impact on its i-cache behavior. Its structure is much more important, as illustrated with the case CPT=0.5. The second observation is that as far as optimization is concerned, a conservative approach, which tends to optimize more code than necessary, generates better results than an aggressive approach. Also the presented i-cache optimization techniques are not very sensitive to the precision of the common path detection. Good results were also achieved with only rough predictions.

The performances obtained with HIPPCO were then compared to those of the BSD implementation. The miss rate of the BSD code is equal to 0.23% with a cache of 8KB and increases to 17.18% with a cache of 4KB. The miss rate of the code HIPPCO with specialization is equal to 0.1 (cache of 8KB or 4KB) which is respectively 2.3 and 171.8 times better than the BSD implementation. When raw profiling is performed, the miss rate of the HIPPCO code is 0.08% and 16%, which is still respectively 2.875 and 1.4375 times better than the BSD implementation.

There are several reasons that explain the relatively poor i-cache behavior of the BSD implementation. The first one is that although the code of the BSD input procedure has been fully optimized, the output procedure has not. An other reason is that the BSD implementation is composed of several functions (*tcp_ input, tcp_ output, tcp_ timer, ...*) which interact. For example, when a packet is received, the *tcp_ input* function is called which possibly calls the tcp_output function to send a acknowledgement. This increases the i-cache pollution. In HIPPCO, the code of all protocols is integrated within one procedure. This structures gives more opportinuties to generate straightlined code and gives more control on the i-cache behavior.

5.5.5 Pipelining Performance

In this section, we analyse the impact of the different optimizations on the pipelining performances. The results are presented in the table 7. The first column presents the different implementations used in this experiment. The second column displays the total number of instructions executed, the third column the total number of cycles necessary and the fifth the number of memory stalls. The fourth column display the CPIs, which is computed by dividing the total number of cycles by the total number of instructions.

	instructions	cycles	CPI	stalls
BSD ???	1665554	2137925	1.28	795843
Esterel (interpreted code)	1086706	1653029	1.52	757405
Esterel (compiled code +O)	572351	745414	1.3	314128
HIPPCO (OS)	661352	875418	1.32	362125
HIPPCO (OSP)	589325	786390	1.33	337119
HIPPCO (OSEP)	412327	510380	1.24	204113
HIPPCO (OSEPU)	412352	510434	1.24	204143
HIPPCO (OSEPCb)	411326	506387	1.23	198120
HIPPCO (OSEPCf)	412325	512379	1.24	204113
HIPPCO (OSEPUCb)	411351	506436	1.23	198140
HIPPCO (OSEPUCf)	412354	512434	1.24	204143
HIPPCO (OSEPUCbT)	402198	493249	1.23	187077
HIPPCO (OSEPUCfT)	403202	498248	1.24	202078
HIPPCO (OSEPUCbT, raw profiling)				
$CPT > 0.5$	653360	876443	1.34	354145
$CPT \geq 0.5$	439364	542431	1.23	202137

Table 7: Pipelining Behavior (protocol only)

The CPI, which is the average number of cycles per instruction, decreases from 1.52 to 1.3, when the original Esterel code (interpreted array coding) is compiled (inlined). This result was expected. In fact, in the array coding implementation, all Esterel actions are implemented as a function called by the protocol engine.

Therefore the number of function calls is much higher. Function calls increases the number of memory stalls because the branch PC value has to be computed before the following instructions can be executed. In fact, as it has been explained in section 3.2, all instructions are broken into five stages (*fetch, decode, execute, memory-access* and *write-results*), which are usually pipelined. The first stage, fetches the instruction to be executed but also increases the PC by 4. This new PC is used to fetch, in the next cycle, the following instruction. This pipelining works fine if the current instruction does not modify the PC. When the current instruction modifies the PC to jump somewhere in the code (which is what happens when a function call is performed), the PC of the next instruction can not be evaluate before the current instruction is decoded and executed. This generates three memory stalls.

When the function is inlined, no jump is performed. The instructions to execute are straightlined and no stall is generated. The figure 21 shows the pipelining before and after the inlining has been performed.

cycles	**j**	**j+1**	**j+2**	**j+3**	**j+4**	**j+5**	**j+6**	**j+7**	**j+8**	**j+9**
Branch instruction	**IF**	**ID**	**EX**	**MEM**	**WB**					
Instruction i+1	*stall*	*stall*	*stall*	*stall*	**IF**	**ID**	**EX**	**MEM**	**WB**	

(a) Before Inlining

cycles	**j**	**j+1**	**j+2**	**j+3**	**j+4**	**j+5**	**j+6**	**j+7**	**j+8**	**j+9**
Instruction i	**IF**	**ID**	**EX**	**MEM**	**WB**					
Instruction i+1	*stall*	**IF**	**ID**	**EX**	**MEM**	**WB**				

(b) After Inlining

Figure 21: Pipelining

When the *Common Branches Extension* optimization is used, the CPI goes down to 1.24. With this optimization, all common branches are inlined and the common path is coded as a tree. This CPI decrease is then also explained by the reduction of function calls.

When the *Tests' Outcomes Rescheduling* optimization is performed the CPI decreases to 1.23. This optimization increases pipelining performance by scheduling the most frequently executed outcomes of each test such the predictions made by the processor turn out to be correct on the critical path (section 3.2).

When raw profiling is performed, the CPI differs according to the *CPT*. When *CPT* is set to 50%, the common path is overestimated and the CPI is equal to the CPI obtained when specialization is performed, thus 1.23. When *CPT* is set to 50.1%, the common path is under-estimated and the CPI increases to 1.34.

The CPI of the BSD implementation is equal to 1.28, which is 4% larger than the HIPPCO CPI. The primary reason is that the BSD implementation is composed of several functions (*tcp_input(), tcp_output(), tcp_timer(),...*) which interact, whereas the common path of the HIPPCO code is implemented within one single function.

5.5.6 Code Size: HIPPCO vs BSD

In the section, we analyse the code size of the different automatically generated implementations. Those results, which are presented in table 9, are compared with the sizes of BSD and Esterel implementations.

The code size of the BSD implementation[6] has been computed by adding up the size of the different object files composing the BSD_TCP protocol (tcp_input.o, tcp_output.o, tcp_timer.o, tcp_subr.o and tcp_xsum.o). The connection establishment and termination phases have not been specified yet in the Esterel version of TCP. Therefore to make the code size comparaison fair, we removed from the BSD version all the pieces of code related to the connection establishement and termination. We then measured its size with and without the header prediction code. The results were respectivelly 10.304 KB and 9.92 KB.

The second set of results related to the Esterel codes. The Esterel compiler can generate three different types of implementations.

1. *Interpreted Coding*: The generated program is composed of an automaton engine and a set of lists. Each of those lists describes the sequence of actions

[6]By BSD implementation, we mean a version of the BSD Kernel code that we ported at the user level.

	text	data
BSD	10.304	0.144
BSD (without header prediction)	9.92	0.144
Esterel (interpreted coding)	8.2	155
Esterel (boolean coding)	21.6	1.1
Esterel (compiled coding)	1772	/
Esterel (compiled coding + O)	45.7	0.5
HIPPCO Application Profiled		
OSEPUCb	33.2 (17)	0.248
HIPPCO Default Profiled		
OSEPUCb	38.4 (24.84)	0.2
HIPPCO Specialized		
OSEPUCbT	21.376 (12.66)	0.2
OSPT	3.54 (5.8)	0.2
HIPPCO Raw Profiled		
OS	32.4 (13)	0.2
OSEPUCb	60 (32)	0.5

Table 8: Code Size (KB)

to perform per state. As expected, the size of the text part is small (8.2KB), but its data part is very large (155KB).

2. *Boolean Coding*: The generated program is a list of boolean equations, which results determine the actions to perform. This code size is 21.6KB.

3. *Compiled Coding*: The generated code is a program, whose instructions are inlined. The size of the inlined generated code is so large that we were not able to compile with the same level of optimization (O9). For comparaison purpose, we compile it without any *gcc* optimization and obtain an object file with a text part of 1.772MB. This huge code size is explained by the tree representation of Esterel, which generates a lot of code duplication.

   Esterel features some optimisations that identify indentical subtrees and share their coding in order to minimize code duplication. Those optimisations (O) reduced the code size to 45.7KB.

HIPPCO features some additional optimisations to reduce furthermore the code size. Those optimisations, which are not presented in this report, are referred as “S” and “N”. At the time the experiments were performed the last optimisation (N) was not implemented. We therefore present the code size of the implementations without this optimisation enabled. For comparaison purpose, we displays in parentheses the code size obtained when this optimisation is on. This optimisation only affect the uncommon path. It should not affect the code speed performance. On the contrary, we expect better i-cache performance.

When those two optimisations are applied (without any code speed optimisation), the size of a fully functional TCP implementation is 13 KB.
This code is quite small but pretty slow.In order to get a program with a better size/speed tradeoff, profiling information may be used by the compiler. The performance of the generated code is then greatly improved at the cost of a small code size increase. With application specific profiling information, we obtain a code of 33.2KB (17 KB with the “N” optimisation on). Whe default profiling information is used, a code with a size of 24.8 KB (38.35 KB) is generated. Application specific profiling data refer to profiling data that are specific to the application. Therefore the profiling data at the server side is different than the profiling data at the client side. Default profiling data refers to profiling data that are specific to a given protocol module. The profiling data is then identical at the server and client side. The relatively small code size difference between the application profiled and default profiled implementations is explained by TCP protocol symetry; its profiling data at the server and client side are very similar.

When specialisation is performed (i.e. the generated protocol is not completely TCP functional), the size of generated code is smaller. In fact in this case, all unvisited branches and states are prunned. The code size goes down to 21.376 KB (12.66 KB).

The smaller code size is achieved, when specialisation is performed and no code speed optimisations are applied. The code size can then be reduced down to 3.54 KB (5.8 KB).

When the optimizations are performed with raw profiling (all probabilities are set to 50%), the size of the generated code is equal to 60 KB (32 KB). In this case, the common states are identified from the program structure: the common path is the set of states that are the most frequently refered. This is what is called *static analysis* in the litterature [Hos95, WMGH94]. This approach is not very satisfactory for protocol code. In fact, in those codes, all states have transitions that lead to the states processing the errors. As a result, the states processing the errors could be identified as the common path! A conservative approach is preferable: profiling data should only be used if they really correspondant to the environment; however, as we have seen, those profiling data do not have to be very accurate.

The code size of the BSD implementation seems to be optimum. Code sharing is maximum, and sometimes at the expense of the program performance. With HIPPCO, we showed that we could generate programs with very good performance and still with raisonable code size. The code size of the TCP generated code varied from 3.5 to 24.84KB, according to the importance of the size and speed constraints.

5.5.7 Code Size Optimizations: Detailed Analysis

In this section, we present the code size of the generated implementations with the different levels of optimisations. The goal is to study the effect of the code speed optimisations on the protocol code size.

	text	data
HIPPCO (OS)	32.4 (13)	0.57
HIPPCO (OSP)	31.1 (13.2)	0.536
HIPPCO (OSEP)	32.8 (16.1)	0.536
HIPPCO (OSEPR)	32.5 (15.9)	0.536
HIPPCO (OSEPRO)	33.264 (17)	0.536
HIPPCO (OSEPROT)	5.56 (7.9)	0.424
HIPPCO (OSEPROT_IR)	(7.9) 5.56	0.424

Table 9: Code Size (KB)

The HIPPCO code size optimization [Cas95b], which is an improvement of the Esterel optimization, brings the code size down to 32.4 KB (60% improvement).

With the *Common Branches Extension* optimization, the code size increases logically to 32.8 KB. The code size increase is about 1.7 KB. This optimization inlines function calls on the common paths and therefore duplicates some parts of the code.

The *Tests Outcome Rescheduling* optimisation, which switchs the branches of the tests according to their execution probabilities does not affect the size of the code.

The *Outlining* optimization increases the code size of 0.7 KB. This optimisation replaces straighlined code of the uncommon path, by functions. The size increase corresponds to the function overheads.

The *Branches and States Pruning* optimization which cuts off unreachable branches and states of the tree reduces the code size to 5.56KB. The code decreases drastically when the prediction optimizations are used. In fact, it goes down to 21.3 KB, just by removing unreachable branches.

The *Input Rescheduling* optimization, which just reorganizes the inputs' subtrees, does not affect the size of the code.

6 Conclusions and Future Works

The main contribution of this report is to demonstrate that FDT techniques can be used to generate efficient protocol implementations. It consequently proves the viability of the HIPPARCH project.

In this paper, we present HIPPCO, a protocol code optimizer, which automatically generates highly optimized protocol implementations. HIPPCO is part of the HIPPARCH compiler. It features a set of optimizations that reduces the protocol instruction counts and improve its i-cache and pipelining behaviors.

Experiments with the TCP protocol are reported. It is shown that the HIPPCO automatically generated TCP implementation performs better than the BSD TCP in terms of the number of instructions to process an incoming packet. This difference is due to the code structure. The automatic approach restructures the whole program toward better performance, whereas the manual implementation is a modified version of an initial naive implementation. Generating a manual implementation with comparable performance would require a lot of time and effort. Also, the emergeance of configurable communication systems makes the manual code optimization very difficult and even difficult.

Protocol code optimization is mainly performed by improving the performance of the *common path*, which is composed of the most frequently executed instructions. In the case of a communication system, this common path crosses the communication stack layers. Identifying and coding the common path through the different layers make the manual approach very unpractical. With the HIPPARCH compiler, the implementations are fully integrated. The module abstraction of the specification disappears completely in the generated automaton. The identification and the coding of the critical path accross the modules is therefore considerably facilitated. In the longer term, we envisage to integrate the application within the communication automaton. That would make the automatic approach even more attractive and performant.

The code speed performance results presented in this report are very encouraging. The code sizes obtained with HIPPCO, although considerably smaller than those obtained with the Esterel compiler, are still larger than the handcoded codes. More effort should be spent in devising more efficient code size reduction algorithms. Our final goal is to converge to the TCP-BSD code size. Early experiments seem to demonstrate that this code size reduction can only be performed at the expense of the code speed performance.

7 Acknowledgements

The work presented in this report is the result of the work performed by Claude Castelluccia for his Phd research. This work initiated about two years ago. Collaborations and discussions were numerous. It is therefore impossible to acknowledge all the people which help us in this work.

However the authors would like to thank Christian Huitema and Phillip Hoschka for their valuable comments. Phillip Hoschka collaborated to the work presented in section 4.1.1.

The work presented in section 5 was initiated at the University of Arizona, while one of the authors (Claude Castelluccia) was visiting the Computer Science Department last summer (May-June). The authors would like to thank to members of the CS department of UofA and particulary Sean O'Malley for its collaboration and valuable comments, and David Mosberger for its advices on the performance measurement tools.

A BSD-TCP header prediction

```
/*
 * Copyright (c) 1982, 1986, 1988, 1990 Regents of the University of California.* All rights reserved.
 *
  ...
 *
 *      @(#)tcp_input.c 7.25 (Berkeley) 6/30/90
 */

if (tp->t_state == TCPS_CLOSED )
return(0);

    len = lg;
    tcpip->ti_len = len - tcpip_size + sizeof(struct tcphdr);

#ifndef NO_CHECKSUM
    xsum = TCP_xsum_init(tcpip, 0);

    if (len > tcpip_size)
xsum += in_cksum(data_pt[socket_fd].pointer, len-tcpip_size);
#endif
    xsum1 = TCP_xsum_final(xsum);

    if (tcpip->ti_sum != (short16) xsum1)
printf("received checksum : %x expected checksum : %x tcpip length : %d\n",
       tcpip->ti_sum, (short16) xsum1, len);

    off = tcpip->ti_off << 2;
    if (off < sizeof (struct tcphdr) || off > lg)
{
    printf("tcpu_input: incoherence in off !\n");
    goto drop;
}

    tiflags = tcpip->ti_flags;

    /*
     * Convert TCP protocol specific fields to host format.
     */
    tcpip->ti_seq = ntohl((unsigned int)tcpip->ti_seq);
    tcpip->ti_ack = ntohl((unsigned int)tcpip->ti_ack);
    tcpip->ti_win = ntohs(tcpip->ti_win);
/*
 * Header prediction: check for the two common cases
 * of a uni-directional data xfer. If the packet has
 * no control flags, is in-sequence, the window didn't
 * change and we're not retramsnittting, it's a
 * canditate. If the length is zero and the ack moved
 * forward, we're the sender side of the xfer. Just
 * free the data acked & wake any higher level process
 * that was blocked waitung for space. If the length
 * is non-zero and the ack didn't move, we're the
 * receiver side. If we're getting packets in-order
 * (the reassembly queu is empty), add the data to
 * the in buffer and note that we need a delayed ack.
 */
    if(tp->t_state == TCPS_ESTABLISHED &&
       (tiflags & (TH_SYN|TH_FIN|TH_RST|TH_URG|TH_ACK)) == TH_ACK &&
       tcpip->ti_seq == tp->rcv_nxt &&
       tcpip->ti_win && tcpip->ti_win == tp->snd_wnd &&
       tp->snd_nxt == tp->snd_max)
{
if ((lg - tcpip_size) == 0) {
    if(SEQ_GT(tcpip->ti_ack, tp->snd_una) &&
       SEQ_LEQ(tcpip->ti_ack, tp->snd_max) &&
       tp->snd_cwnd >= tp->snd_wnd) {
```

```
/*
 * This is a pure ack for outstanding data.
 */
if(tp->t_rtt && SEQ_GT(tcpip->ti_ack, tp->t_rtseq)){
    tp->time_send = 0;
    tcp_xmit_timer(tp);
}
acked = tcpip->ti_ack - tp->snd_una;
buff_out_ptr->nbeltq -=acked;
buff_out_ptr->nbeltsnd -= acked;
buff_out_ptr->una = (buff_out_ptr->una + acked)%MAX_BUF;
tp->snd_una = tcpip->ti_ack;
/*
 * If all outstanding data are acked, stop
 * retransmit timer, otherwise restart timer
 * using current (possibly backed-off) value.
 * If data are ready to send, let tcp_output
 * decide between more output or persist.
 */
if(tp->snd_una == tp->snd_max)
    tp->t_timer[TCPT_REXMT] = 0;
else if (tp->t_timer[TCPT_PERSIST] == 0)
    tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
return;
    }
}
else if (tcpip->ti_ack == tp->snd_una &&
 tp->seg_next == (struct tcpiphdr *)tp &&
 (lg - tcpip_size) <= (MAX_BUF-buff_in[socket_fd]->nbeltq))
    {
/*
 * This is a pure, in-sequence data packet
 * with nothing on the reassembly queue and
 * we have enough buffer space to take it.
 */
tp->rcv_nxt += (lg - tcpip_size);
/*
 * Drop TCP and IP headers the add data
 * to in buffer.
 */
data_pt[socket_fd].length = lg - tcpip_size;
tp->t_flags |= TF_DELACK;
tp->time_ack = 1;

}
```

B HIPPCO automatically generated code

```
/*  input processing  */
```

```
if (cb->A_Packet_has_been_received) {
 if ((0==( tcpip->th_sum ))) {
  cb->off = (tcpip->th_off) <<  2;
  if (((cb->off>=20)&&(cb->off<= cb->Packet_Size))) {
   cb->I_flag = (tcpip->th_flag);
   if (!((cb->I_flag&TH_SYN)||!(cb->I_flag&TH_ACK))) {
    if (!((cb->I_flag&TH_FIN)||(cb->I_flag&TH_URG))) {
     if (!(cb->I_flag&TH_RST)) {
      cb->I_win = ntons(tcpip->th_win);
      cb->I_line_size = (cb->Packet_Size - 20);
      if ((cb->ack_now==1)) {
      /* an ack. needs to be send */
      /* the following code send an ack and keep processing
      /* the incoming packet */
cb->ack_now = 0;
cb->_N_S =(((cb->_N_S)>(cb->_?SEG_ACK))?(cb->_N_S):(cb->_?SEG_ACK));
cb->_window_allow =(cb->_?Win-(cb->_N_S-cb->_?SEG_ACK));
cb->_left_data =(cb->_?N_U-cb->_N_S);
cb->len = (((cb->window_allow)<( cb->left_data))?(cb->window_allow):( cb->left_data));
cb->idle = (cb->?SEG_ACK!=cb->Snd_Max);
cb->win = 40000;
cb->flag = TH_ACK;
cb->?len = cb->len;
cb->?SEG_SEQ = ntohl(tcpip->th_seq);
(cb->?I_line = tcpip->msg);
cb->?I_line_size = cb->I_line_size;
cb->Rcv_Wnd = (((40000)>((cb->Rcv_Adv-cb->N_R)))?(40000):
( (cb->Rcv_Adv-cb->N_R)));
cb->todropB = (((0)>( (cb->N_R-cb->?SEG_SEQ)))?(0):((cb->N_R-cb->?SEG_SEQ)));
cb->SEG_SEQ_adj = (cb->?SEG_SEQ+cb->todropB);
cb->todropE = (((0)>(((cb->?SEG_SEQ+cb->?I_line_size)-(cb->N_R+cb->Rcv_Wnd))))?(0):\
                      (cb->?SEG_SEQ+cb->?I_line_size)-(cb->N_R+cb->Rcv_Wnd))));
if (!((cb->todropE!=0)||(cb->todropB!=0))) {
 cb->I_line_size_adj = cb->?I_line_size;
 if (((cb->SEG_SEQ_adj==cb->N_R)&&(cb->Reassembly_Q_Empty==1))) {
  len_msg_out = cb->_I_line_size_adj;
  cb->_N_R = (cb->_N_R +  cb->_I_line_size_adj);
  cb->_?N_R = cb->_N_R;
  cb->adv = (cb->win-(cb->Rcv_Adv-cb->?N_R));
  if (!(cb->I_win!=cb->Snd_Wnd)) {
   if (!((cb->win<(40000/4))&&(cb->win<512))) {
    cb->win = ((((((cb->win)<( 65535))?(cb->win):( 65535)))>(
           (cb->Rcv_Adv-cb->?N_R)))?((((cb->win)<( 65535))?(cb->win):( 65535))):(
           (cb->Rcv_Adv-cb->?N_R)));
   cb->Rcv_Adv = ((((cb->?N_R+cb->win))>(cb->Rcv_Adv))?((cb->?N_R+cb->win)):( Rcv_Adv));
   cb->?Rcv_Adv = cb->Rcv_Adv;

   if (!((cb->left_data==cb->len)&&(cb->?End_Of_Input==1))) {
    frame_out = &(packet->frame);
    frame_out->th_seq = (unsigned int)cb->N_S;
    frame_out->th_ack = (unsigned int)cb->?N_R;
    frame_out->th_win = cb->win;
    frame_out->th_flag = cb->flag;
    frame_out->th_off = 1 ; frame_out->th_urp = 0 ;
    frame_out->th_off = (20>>2);
    start = (cb->N_S-(cb->N_S/40000)*40000);
    frame_out->th_sum = 0 ;
    frame_out->th_sum = 0;
    packet->size = 20;
    *nbpktinq += 1;
          cb->?len = cb->len;
           return 2;
          }
         return out10();
         }
        return out5();
        }
       return out4();
       }
      return out3();
      }
     return out7();
     }
    return out2();
   }
   /* this is a pure data and no ack. needs to be send */
   cb->ack_now = 1;
   cb->?SEG_SEQ = ntohl(tcpip->th_seq);
   (cb->?I_line)=((tcpip->msg)));
   cb->?I_line_size = cb->I_line_size;
   cb->Rcv_Wnd = (((40000)>((cb->Rcv_Adv-cb->N_R)))?(40000):( (cb->Rcv_Adv-cb->N_R)));
   cb->todropB = (((0)>( (cb->N_R-cb->?SEG_SEQ)))?(0):((cb->N_R-cb->?SEG_SEQ)));
   cb->SEG_SEQ_adj = (cb->?SEG_SEQ+cb->todropB);
   cb->todropE = (((0)>(((cb->?SEG_SEQ+cb->?I_line_size)-(cb->N_R+cb->Rcv_Wnd))))?(0):\
                 (((cb->?SEG_SEQ+cb->?I_line_size)-(cb->N_R+cb->Rcv_Wnd))));
   if (!((cb->todropE!=0)||(cb->todropB!=0))) {
    cb->I_line_size_adj = cb->?I_line_size;
    cb->Rcv_Adv = cb->?Rcv_Adv;
     if (((cb->SEG_SEQ_adj==cb->N_R)&&(cb->Reassembly_Q_Empty==1))) {
      len_msg_out = cb->_I_line_size_adj;
      cb->_N_R = (cb->_N_R +  cb->_I_line_size_adj);
      cb->_?N_R = cb->_N_R;
      if (!(cb->I_win!=cb->Snd_Wnd)) {
       if (!((cb->I_flag&0x01)||(cb->I_flag&0x20))) {
return 2;
}
       return out10();
       }
      return out9();
      }
     return out11();
     }
    return out8();
    }
   return out1();
   }
   return out0();
  }
 return 2;
 }
 return 2;
 }
 if (cb->_V5) {
  return out12();
  }
return 0;
}
```

References

[A. 95] A. Ghosh and J. Crowcroft. Some lessons learned from various alf and ilp applications. In *Proceedings of HIPPARCH'95*, Sydney, Australia, December 1995.

[AMP91] C. André, J.P. Marmorat, and J.P. Paris. Execution machines for esterel. In *European Control Conference, Grenoble*, July 1991.

[AP92] Mark B. Abbott and Larry L. Peterson. Automated integration of communication protocol layers. Technical Report TR 92-24, Department of Computer Science, University of Arizona, December 1992.

[BD95] Torsten Braun and Christophe Diot. Protocol implementation using integrated layer processing. In *SIGCOMM*, September 1995.

[BdS91] Frédéric Boussinot and Robert de Simone. The ESTEREL language. Technical Report 1487, INRIA U.R. Sophia-Antipolis, July 1991.

[BG89] Gérard Berry and Georges Gonthier. Incremental development of an HDLC protocol in esterel. Technical Report 1031, INRIA U.R. Sophia-Antipolis, May 1989.

[BSS92] Donald F. Box, Douglas C. Schmidt, and Tatsuya Suda. ADAPTIVE - an object-oriented framework for flexible and adaptive communication protocols. In *Proceedings of the Fourth IFIP Conference on High Performance Networking*, December 1992.

[C. 95] C. Diot and I. Chrisment and A. Richards. Application Level Framing and Automated Implementation. In *Proceedings of HPN'95*, Palma, Spain, September 1995.

[Cas95a] Claude Castelluccia. Automating header prediction. In *1st Annual Workshop on Compiler Support For System Software*, 1995.

[Cas95b] Claude Castelluccia. Code speed/size tradeoffs in protocol design. 1995. In preparation.

[CCD$^+$94] 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 U.R. Sophia-Antipolis, Octobre 1994.

[CD94] Claude Castelluccia and Walid Dabbous. Modular communication subsystem implementation using a synchronous approach. In *Usenix High-Speed Networking*, August 1994.

[CH95] Claude Castelluccia and Phillip Hoschka. A compiler-based approach to protocol optimization. In *High Performance Communication Subsystem*, August 1995.

[CHT91] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. In *Software-Practice and Experience*, June 1991.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of tcp processing overhead. In *IEEE Communications Magazine*, June 1989.

[CT90] 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, September 1990.

[DH92] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. In *IEEE Transactions on Software Engineering*, February 1992.

[Fel93] David C. Feldmeier. A survey of high performance protocol implementation techniques. Technical report, Bellcore, February 1993.

[GC90] R. Gupta and C. Chi. Improving instruction cacge behaviour by reducing cache pollution. In *Proceedings of the Supercomputing Conference*, November 1990.

[GPSV91] Per Gunningberg, Craig Partridge, Teet Sirotkin, and Bjorn Victor. Delayed evaluation of gigabit protocols. In *Proceedings of the Second MultiG Workshop*, June 1991.

[HH93] Philipp Hoschka and Christian Huitema. Control flow analysis for automatic fast path implementation. In *Second Workshop on High Performance Communication Subsystems*, 1993.

[Hos95] Philipp Hoschka. Optimisation automatique dans un compilateur de talon de communication. Technical Report Phd Thesis, INRIA Sophia-Antipolis, July 1995.

[HP90] J. L. Hennesy and D. D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[McF89] Scott McFarling. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[MPO95] 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, January 1995.

[NL88] Hutchinson N. and Peterson L. Design of the x-kernel. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 65–75, August 1988.

[OP91] S. W. O'Malley and L. L. Peterson. A highly layered architecture for high-speed networks. In *Protocols for High-Speed Networks II IFIP*, 1991.

[PH90] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990.

[RdS90] V. Roy and R. de Simone. Auto and autograph. In *Proceedings of Workshop on Computer Aided Verification*, June 1990.

[SKP94] Steven E. Speer, Rajiv Kumar, and Craig Partridge. Improving unix kernel performance using profile based optimization. In *Winter Usenix 1994*, 1994.

[Smi92] A. J. Smith. Cache memories. In *Computing Surveys*, August 1992.

[WMGH94] T. Wagner, V. Maverick, S. Graham, and M. Harrison. Accurate static estimators for program optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85,96, 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399