# Compact and Efficient Presentation Conversion Code

*Philipp Hoschka*

October 26, 1997

INRIA Centre de Sophia Antipolis,
2004 Route des Lucioles, BP-93,
06902 Sophia Antipolis Cedex, FRANCE.

e-mail: hoschka@sophia.inria.fr

**Abstract**

Presentation conversion is a key operation in any development environment for distributed applications, such as Corba, Java-RMI, DCE or ASN.1-based environments. It is also well-known performance bottleneck in high-speed network communication. Presentation conversion code is usually generated by an automatic code generation tool referred to as stub compiler. The quality of the code generated by a stub compiler is often very low. The code is either very slow, or has a large code size, or both. This paper describes the design and experimental evaluation of an optimization stage for a stub compiler. The optimization stage automates the trade-off between code size and execution speed of the code generated by the compiler. This is achieved by using a hybrid of two implementation alternatives for presentation conversion routines (interpreted and procedure-driven code). The optimization problem is modeled as a Knapsack problem. A Markov model in combination with a heuristic branch predictor is used for estimating execution frequencies. The optimization stage is added to the ASN.1 compiler Mavros. Experimental evaluation of this implementation shows that by investing only 25% of the code size of fully optimized code, a performance improvement of 55% to 68% can be achieved.

## 1 Introduction

Presentation conversion is often been identified as "the performance problem of the 90's" in text books on high-speed networking (e.g.( Partridge, 1993), (Peterson & Davie, 1996)). The performance of automatically generated presentation conversion code often prevents applications to take advantage of the performance of high-speed networks. For example, we found that due to presentation conversion code, the throughput of a 155 Mbit/s ATM network can be reduced to 3.1 Mbit/s on the application level, corresponding to a loss in performance by a factor of fifty.

The performance of presentation conversion code can be considerably improved by increasing the size of the code, using techniques such as compilation and inlining. While stub compilers today implement these optimization techniques, their use often leads to code sizes that are unacceptable for many applications in practice.

Further analysis reveals that the code size could be largely reduced if these optimizations where only applied to the parts of the presentation conversion code that are used most often. In other words, rather than applying the same optimization technique (compilation or inlining) to all or the presentation conversion code, it should only be applied to the code parts where it is most beneficial.

This paper shows how a stub compiler can automatically decide which parts of the code are worth optimizing. It is based on three main contributions:

- It is shown that the size/speed trade-off problem in the generation of presentation conversion code can be mapped onto a Knapsack problem.
- Heuristic branch prediction rules are developed that allow determining the execution frequency of different parts of the presentation conversion code.
- Experiments show that adding an optimization stage to stub compilers is worthwhile.

The rest of this paper is structured as follows: Section 2 explains basic concepts of the implementation of presentation conversion routines. Section 3 discusses related work. Section 4 describes the design of the optimizer, including the Knapsack model and the heuristic branch prediction rules. Section 5 gives experimental results for the optimization stage. Section 6 gives our conclusions.

## 2 Basic Concepts

The task of a stub compiler is to translate an interface definition into presentation conversion routines. These routines are required to translate data values from a machine-internal format into a format suitable for network transmission at the sender (*marshalling*, and to do the reverse conversion at the receiver (*unmarshalling*).

An *interface definition* contains declarations of the data types that an application exchanges over a network. Type definitions consist of primitive types (e.g. integer, real, string ) and composed types (e.g. struct, array, union). Consequently, the presentation conversion code generated by a stub compiler contains two types of code, namely code dealing with converting primitive types, and code dealing with converting composite types.

The code dealing with primitive types handles format conversions such as translating between ASCII and EBCDIC character sets, byteswapping for integers or floating point format conversion. These conversions are done by specialized algorithms.Optimizations are specific to the particular algorithm, and not treated in this paper.

The paper focuses on optimizing the code for converting composite types, referred to as *control code*. This code has two purposes, linearisation and realignment. Linearisation is required for data structures stored in non-contiguous memory sections, such as dynamically allocated tree structures. Realignment is required for components of a record or a struct type. Different CPUs may use different conventions for positioning these fields in main memory.

The control code of presentation conversion routines can be implemented using three alternative implementation techniques: interpreted code, procedure-driven code (also referred to as compiled code) and inlined code.

With *interpreted code*, the stub compiler translates a composite type into a set of commands, one for each component. To convert a particular data value, these commands are interpreted by a generic routine. The advantage of using interpretation is compact. The disadvantage is slow execution speed. In our measurements, throughput of interpreted routines was as low as 3.1 Mbit/s.

With *procedure-driven code*, the stub compiler translates each type declaration into a procedure. This procedure contains procedure calls for converting the components of the composite type, interspersed by control code that depends on the type constructor. Since procedure driven code eliminates interpretation overhead, it is faster. We measured speed-up factors between 1.6 to 5.7 when replacing interpreted code by procedure-driven code. However, procedure-driven code can significantly increase the code size. Our measurements show a size increase by a factor of 4 to 7 when changing from interpreted code to procedure-driven code.

With *inlined code*, the stub compiler replaces the procedure calls in procedure-driven code with the body of the procedure that is called. Inlining leads to another improvement in execution speed, but also to a very large increase in code size. Extending the techniques developed in this paper to also handle inlining is relatively straightforward. Inlining is thus not further treated in this paper, and the reader is referred to (Hoschka, 1995) for the details of a solution.

## 3 Related Work

The approach presented in this paper resolves the conflict between the desire to improve the performance of presentation conversion routines and the desire to keep the code size of these routines under control. This conflict is well document in research literature.

Performance problems due to presentation conversion routines have been reported many times from different areas of computer science research (Huitema & Doghrie, 1989), (Clark & Tennenhouse, 1990), (Thekkath & Levy, 1993)). Code size problems have been reported when using stub compilers for developing practical systems, e.g. an X.500 directory service (Sample & Neufeld, 1993) or an operating system ((Jones, 1985), (Kessler, 1994)).

As a result, stub compilers often offer compiler switches to choose between two different code generation strategies ((Zahn et. al, 1990), (Corbin, 1990), (Huitema, 1991), (Sample, 1993), (Kessler, 1994)). The switch only decides which of the alternatives is chosen (interpreted, procedure-driven or inlined). Then, the same alternative is used to generate code for all types in the interface specification. The result is either efficient but bulky code, or compact, but inefficient code. In contrast, the stub compiler presented in this paper provides a switch to specify the maximally allowable code size. The compiler then tries to generate the most efficient code possible under the given size constraint.

USC (O'Malley et al., 1994) implements an optimization method based on inlining that significantly improves presentation conversion speed for TCP/IP packets. However, the method cannot be applied to more complex, application-oriented interface definitions without incurring a significant code size overhead, since it is based on inlining. An optimization stage that follows the principles presented in this paper can alleviate this problem, and thus make the USC optimization method usable for other protocols than TCP/IP.

Designing an optimization stage for size/speed trade-off problems is difficult, even at today's advanced state of the art in compiler construction (Bacon, Graham & Sharp, 1994). Most standard compiler optimizations either decrease the size of the code (elimination of induction variables or of dead code) or have no significant effect on the size (register allocation, instruction scheduling). (Pittman, 1987) presents the general idea of using a hybrid between interpreted and procedure-driven code to solve size/speed trade-off problems. However, in contrast to the work presented in this paper, the trade-off is not done automatically, but requires intervention by the programmer. (Scheifler, 1977) shows that the size/speed trade-off for inlining procedure-calls is equivalent to a Knapsack problem. We extend this result to the trade-off between interpreted and procedure-driven code.

The optimizer presented in this paper relies on execution frequency information. The conventional approach for gathering this information is to collect execution trace data ( gprof (Graham et al., 1983), (Pettis & Hansen, 1990), (McFarling, 1991)). However, this approach makes code optimization very time-consuming for the application programmer. The amount of effort usually becomes prohibitive for distributed programs, since two or more program modules must be run in parallel to collect trace data. An alternative is to derive execution frequency information by having the compiler analyze the structure of the source code, e.g. by heuristic branch prediction (Ball & Larus, 1993). This approach does not require human intervention, and is thus well-suited for optimizing distributed programs. The work presented in this paper develops heuristic branch prediction rules for interface specifications.

Earlier results of this work ((Hoschka & Huitema, 1993), (Hoschka & Huitema, 1994)) have been applied to solve a related size/speed trade-off problem in the area of automatic code generation of protocol automata in (HIPPCO (Castelluccia & Hoschka, 1995), (Castellucia et al., 1997))). The work presented in this paper extends these results by describing how the size/speed trade-off can be resolved by a mapping onto a Knapsack problem. In contrast to (Castellucia et al., 1997), we do not rely on manual annotation for deriving execution frequencies, but use an automatic approach based on heuristic branch prediction.

## 4 Optimizer Design

### 4.1 Knapsack Model

For the following discussion, we define a generic type definition language. The language contains a set of scalar types such as integer or real types, and the following set of type constructors: (1) A *structure* defines a linear sequence of fields of usually different types. (2) A *union* defines alternatives between fields of usually different type. (3) An *array* defines a sequence of fields of the same type.

The generation of optimized presentation conversion routines starts from an intermediate representation of the interface definition. We define the *syntax graph* of an interface specification as a tuple $\{V, E\}$ where $V$ is the set of nodes in the syntax graph for which an optimization decision is required and $E$ is a set of arcs representing the sequence in which the nodes occur in the interface specification.

A syntax graph contains two different classes of nodes: *type definition nodes* and *field nodes*. Each definition of a constructed type in the interface specification is mapped to a type definition node in the syntax graph. A type definition node is labeled by the type constructor of the type definition.

A field in a constructed type is mapped to a *field node* in the syntax graph. There are three different kinds of field nodes, which can be distinguished by a label. A field node can be a constructed type (*embedded type constructor* label), a reference to a pre-defined scalar type (*scalar reference* label) or a reference to another type definition (*type reference* label).

For formalizing the optimization problem, we define the following variables: $S$ is the total size of presentation conversion code before optimization, $S_{opt}$ is the total size of presentation conversion code after optimization, $T$ is the total execution time of presentation conversion code before optimization for a given workload and $T_{opt}$ is the total execution time of presentation conversion code after optimization for a given workload.

The objective of optimization is then to generate presentation conversion routines in a way that minimizes $T_{opt}$ under the constraint that $S_{opt}$ does not exceed a given maximal code size. We define:

- $s_i$ : Size of code template for type definition node $i$ before optimization.
- $s_{i-opt}$ : Size of code template for type definition node $i$ after optimization. This corresponds to the code duplication caused by the optimization.
- $t_i$ : Time for converting values of the type definition mapped onto node $i$ before optimization.
- $t_{i-opt}$ : Time for converting values of the type definition mapped onto node $i$ after optimization. This corresponds to the overhead saving achieved by the optimization.
- $f_i$ : Execution frequency of presentation conversion code for type definition mapped onto node $i$ for a given workload.
- $x_i$ : $x_i = 1$ if node $i$ is optimized, 0 otherwise.

With this, we have:

$$S = \sum_{i=1}^{n} s_i \tag{1}$$

$$S_{opt} = \sum_{i=1}^{n} (x_i s_{i-opt} + (1 - x_i) s_i) \tag{2}$$

$$T = \sum_{i=1}^{n} f_i t_i \tag{3}$$

$$T_{opt} = \sum_{i=1}^{n} f_i (x_i t_{i-opt} + (1 - x_i) t_i) \tag{4}$$

The size/speed trade-off occurring when generating optimized presentation conversion code can be expressed as a *0-1 Knapsack problem* (Martello & Toth, 1990). For this, each type definition node $i$ in the syntax graph is assigned a profit $p_i$ and a weight $w_i$ as follows:

$$p_i = f_i (t_i - t_{i-opt}) \tag{5}$$

$$w_i = s_{i-opt} - s_i \tag{6}$$

Substituting these equations into the general definition of a Knapsack problem results in the following optimization problem:

maximize:

$$\sum_{j=1}^{n} f_j (t_j - t_{j-opt}) x_j \tag{7}$$

subject to:

$$\sum_{j=1}^{n}(s_{i-opt} - s_i)x_j \leq c \tag{8}$$

It can be assumed that both the profit and the weight are positive numbers. A negative profit value corresponds to an optimization that increases the execution time, which is impossible by definition. A negative weight corresponds to an optimization that does not increase the code size. This case occurs for example when very small functions are written inline, since the number of instructions required for function linkage is higher than the number of instructions in the function body.The author of a stub compiler should avoid this by using macros definitions rather than function definitions.

For solving the 0-1 Knapsack problem, we use an approximate algorithm. The advantage over exact algorithms is that it is easy to implement. Moreover, investing much effort into making the Knapsack solution found by the algorithm accurate seems inappropriate, given that the values for weights and profits are also only approximations (see below).

First, items are sorted by their profit/weight ratio or profit per unit weight, i.e. so that

$$p_i/w_i \geq p_2/w_2 \geq \cdots \geq p_n/w_n \tag{9}$$

Then, items are consecutively inserted into the Knapsack in the order of this list, until the first item $s$ is encountered that does not fit.

We use a heuristic for improving this solution which is to continue going through the list items following the critical item and including each item that fits into the residual Knapsack capacity. This algorithm is known as *Greedy algorithm* for finding a solution to the Knapsack problem (Martello & Toth, 1990).

For solving the Knapsack problem, the stub compiler must have information on $s_i$, $s_{i-opt}$, $t_i$, $t_{i-opt}$ and $f_i$. All of these values can generally only be estimated. One reason for this is that the stub compiler generates code in an application programming language. Therefore, the $s_i$, $s_{i-opt}$, $t_i$, $t_{i-opt}$ depend on the machine code generated by the application language compiler. Moreover, $f_i$ depends on the actual workload. Only estimates of this workload are available at the time the presentation conversion code is generated. The following two sections discuss how these parameters are estimated in our implementation.

## 4.2 Predicting Execution Frequencies

The basic hypothesis behind heuristic branch prediction is that the structure of the source code can be used by the compiler to derive estimates on how frequently different sections of the code will be executed. In the following, we apply this idea to an interface definition.

Let $v_i$ be the average number of occurrences of field $i$ in all values of type $t$ in a given workload. For estimating $v_i$, the following cases can be distinguished:

- field $i$ is part of a union type: $v_i = 1/n$, where n is the number of fields in the union.
- field $i$ is part of a structure type:
  - field $i$ is not marked as optional: $v_i = 1$.
  - field $i$ is marked as optional: $v_i = \lambda$, $0 \leq \lambda \leq 1$. As explained below, special care must be taken when calculating $\lambda$ in the case of recursive types.

- field $i$ is part of an array type:
  - the length of the array is unknown: $v_i = \mu$.
  - the length of the array is known: $v_i = n$, where $n$ is the length of the array.

In the case of optional fields, special care must be taken. In many interface definition languages, optional fields can be used to construct recursive types. For our method to work, it must be ensured that the flow along a recursive path is smaller than 1. A flow greater or equal to 1 would correspond to an infinite recursion, which is impossible. Thus, an additional pass must be added to the frequency predictor that loops in the type reference graph caused by recursive optional fields.

We define the *type reference graph* of an interface specification as a tuple $\{V', (E', w)\}$. $V'$ corresponds to the set of all type definition nodes in the interface specification's syntax graph. The set of weighted arcs $((E', w)$ is computed by locating all type reference nodes in the syntax graph, and adding an arc from the type definition node containing the type reference node to the type definition node that is referenced. The weight of this arc is equal to the value of $v_i$ of the type reference node at which the arc originates.

The type reference graph is a weighted graph that can be interpreted as a Markov model for the behavior of the presentation conversion code. $V'$ correspond to the states of the Markov model. The transition probabilities between the states can be derived from the weighted arcs $(E', w)$. Following an approach proposed in (Ramamoorthy, 1965), the graph can be mapped onto a system of linear equations that represents the equations for determining the visit counts of the nodes. The frequency of each type definition node can be computed by solving this LEQ.

## 4.3 Predicting Code Size and Execution Speed

Determining the exact value of $s_i$, $s_{i-opt}$, $t_i$ and $t_{i-opt}$ is cumbersome, since it involves counting assembly code instructions. Instead of estimating these values directly, we therefore use a predictor for estimating the size increase $s_\Delta = (s_i - s_{i-opt})$ and the decrease in execution time $t_\Delta = (t_{i-opt} - t_i)$.

We assume that the code size increases as a linear function of the number of nodes for which procedure-driven code is generated. furthermore, we assume that execution time decreases as a linear function of the number of interpreter calls that are saved when procedure-driven code is generated. Under these assumptions, we derive the following predictors for the three type constructors of the generic type definition language:

- structure type
  - $s_\Delta = 1 + n$, where n is the number of fields in the structure
  - $t_\Delta = 1 + n$, where n is the number of fields in the structure
- union type
  - $s_\Delta = 1 + n$, where n is the number of fields in the structure
  - $t_\Delta = 2$. In a union type, the interpreter is called once for handling the type definition node, and once for handling the union field.
- array type

| Type constructor | Code size increase | Execution time savings |
|---|---|---|
| structure | $1+n$ | $1+n$ |
| union | $1+n$ | 2 |
| array | 2 | $1+\mu$ |

Table 1: Predictors for effect of procedure-driven code on code size and execution time

- $s_\Delta = 2$. Generating procedure-driven code for an array means concerns two nodes, the type definition node for the array, and the field node.
- $t_\Delta = 1 + \mu$. As previously defined, $\mu$ is the number of fields in the array.

Table 1 summarizes these results in compact form.

## 5 Performance Evaluation

### 5.1 Baseline Performance

In order to determine the practical impact of the different code generation alternatives on execution time, we measured the execution time for a number of benchmarks written in ASN.1. Measurements were done on a Sun Sparc 10, Model 40, using gcc version 2.6.0, static linking and optimization level 2 (O2). The presentation conversion code was generated by the ASN.1 compiler Mavros (Huitema, 1991).

We used one benchmark for each of the four type constructors available in the interface definition language ASN.1 (Steedman, 1990). The *Sequence* type corresponds to a structure type in C. The *Set* type is a special case of a structure where structure fields can be reordered before they are sent out on the net. The *Choice* type corresponds to a C union type. The *Sequence Of* type is equivalent to an array in C. In all experiments, the type definition contained ten integer values.

Table 2 gives the throughput measured in these experiments for interpreted and procedure-driven presentation conversion code. For each experiment, we report three values: the throughput of interpreted code, the throughput of procedure-driven code and the factor by which interpreted code is slower than procedure-driven code. The latter serves to eliminate system-dependencies inherent in the absolute values.

From these numbers we see that the speed difference between interpreted and procedure-driven presentation conversion code is significant. Compiled code is consistently faster than interpreted code by a factor of 1.6 to 5.7. Some of the numbers measured for interpreted code are not sufficient to saturate an Ethernet (10 Mbit/s), and none of the alternatives can saturate network connections with higher throughputs such as FDDI (100 Mbit/s) or the ATM configuration commonly used with workstations (155 Mbit/s). Another observation is that unmarshalling is consistently slower than marshalling. This is due to the requirement for error-checking in unmarshalling code.

We conclude that from a performance point of view the use of procedure-driven presentation conversion code is preferable to the use of interpreted code. The choice of the implementation technique for presentation conversion code can decide whether the installation of expensive high-speed network was worthwhile for speeding up a particular

| | Marshalling | | | Unmarshalling | | |
|---|---|---|---|---|---|---|
| | Interpreted | Compiled | Factor | Interpreted | Compiled | Factor |
| Sequence | 15 Mbit/s | 60 Mbit/s | 4 | 11 Mbit/s | 33 Mbit/s | 3 |
| Set | 15 Mbit/s | 60 Mbit/s | 4 | 7.5 Mbit/s | 26 Mbit/s | 3.5 |
| Choice | 10 Mbit/s | 24 Mbit/s | 2.4 | 3.1 Mbit/s | 18 Mbit/s | 5.7 |
| Sequence Of | 18 Mbit/s | 52 Mbit/s | 2.8 | 7 Mbit/s | 11 Mbit/s | 1.6 |

Table 2: Throughput of interpreted and procedure-driven presentation conversion code

| | Interpreted | Compiled |
|---|---|---|
| X.400 | 9 KByte | 37 KByte |
| Z39.50 | 17 KByte | 51 KByte |
| FTAM | 18 KByte | 103 KByte |
| X.500 | 18 KByte | 137 KByte |
| Total | 62+8 = 70 KByte | 328 KByte |

Table 3: Size of interpreted and procedure-driven presentation conversion code

implementation. Installing an ATM network is of little use if the presentation conversion code of an application executes at Ethernet throughput.

However, making presentation conversion code faster has the drawback of increasing its code size. Table 3 shows the difference in code size between interpreted and procedure-driven presentation conversion code when generating presentation conversion code for four different applications whose interfaces are specified in ASN.1. The numbers include both the marshalling and the unmarshalling routines. The numbers in the "interpreter" column give the object size of the interpreter commands generated. The interpreter itself takes up an additional 8 KByte. X.400 is the e-mail protocol defined in the ISO-OSI protocol stack, Z39.50 is an information retrieval protocol, FTAM is the ISO-OSI protocol for file transfer, access and management, and X.500 is the ISO-OSI protocol defined for access to a directory service. The same configuration and compilers as in the throughput measurements were used.

As can be seen from the numbers in Table 3, the size of procedure-driven presentation conversion code is significantly larger than the size of interpreted code. This becomes even clearer when comparing the aggregate code sizes. Aggregate code size is important, since network applications are usually run in the background. If these network applications take up too much memory space, it may impossible to run interactive applications such as a word processor.

## 5.2 Frequency Prediction

In order to evaluate the accuracy of the frequency prediction heuristics, we manually estimated the values of the arc-weights $v_i$ for the ASN.1 specification of an e-mail protocol (X.400 P1) (Hoschka, 1995). We compare these manual frequency estimates with the performance of three different prediction heuristics:

- *Type reference heuristic*: Using this heuristic, the frequency of type $T$ is predicted by the number of times $T$ is referenced by other types. For this purpose, the optional fields are assumed to be always present, and the length of array values is set to 1, i.e. $\lambda = 1$ and $\mu = 1$.

- *Optional heuristic*: This heuristic refines the type reference heuristic by taking into account that types referenced by optional fields will occur less frequently than types referenced by non-optional fields. Using this heuristic, the field coefficient for optional type references is set to a value lower than one. In our experiment, we use the value 0.5, i.e. $\lambda = 0.5$. All array values are assumed to have a length of 1, i.e. $\mu = 1$.

- *Array heuristic*: This heuristic assumes that the array fields dominate the distribution of fields. Therefore, the length of arrays is set to a value higher than one. In our experiment, we use the value two, i.e. $\lambda = 1$ and $\mu = 2$.

Figure 1 compares the results of the manual frequency prediction with each of the prediction heuristics for the X.400 benchmark. We first analyze the global quality of the heuristic predictions when compared to the manual prediction. Looking at the most frequently used types to the left of the graphs, we see that in all heuristics the top two most frequently used types are predicted with excellent accuracy.

The largest error is introduced by the array heuristic. With this heuristic, a type that is rarely used in practice is put into the top 20% of the most frequently used types.

This shows that at least for X.400 P1 the assumption that many types will be transmitted as fields of an array does not hold. The definition of an array type does not necessarily mean that an array will actually be transmitted. Array types are used rather to indicate that the arity of a field can be bigger than one, even when it is equal to one in most practical cases. Given that the array construct is the equivalent to a loop in an interface definition, this is an important important observation. It contradicts the hypothesis used for writing optimizers in general-purpose compilers that a program spends most of its time in loops, and that therefore loop optimizations are more important than others.

The excellent prediction results for type frequencies in X.400 P1 are due to the fact that electronic mail messages contain a "central" data type, namely the e-mail address. Addresses are contained in many places of the message: in the sender field, in the recipient field and in the array tracing the message's route through the e-mail transmission system. Consequently, the types making up the address fields are referenced by many other types in the ASN.1 specification. Many distributed applications contain such central data types. We repeated the experiment with two other ASN.1 specification, the X.500 directory service and the Z39.50 information retrieval protocol, and found that the "central" data structure (directory name and record) always ended up in the top 20% of the most frequent types.

## 5.3 Size/Speed Trade-Off

In a final round of experiments, we validated the quality of the code produced by the stub compiler after an optimization stage was added. We used the ASN.1 specifications for X.400 P1 and Z39.50 as benchmarks. For these benchmarks, we manually estimated the values of the arc-weights $v_i$ (Hoschka, 1995). The parameters used for the heuristic frequency prediction were set as follows: $\lambda = 0.5$, $\mu = 1$. We then varied the compiler parameter $S_\Delta$ controlling the code size increase due to procedure-driven code. $S_\Delta$ is defined

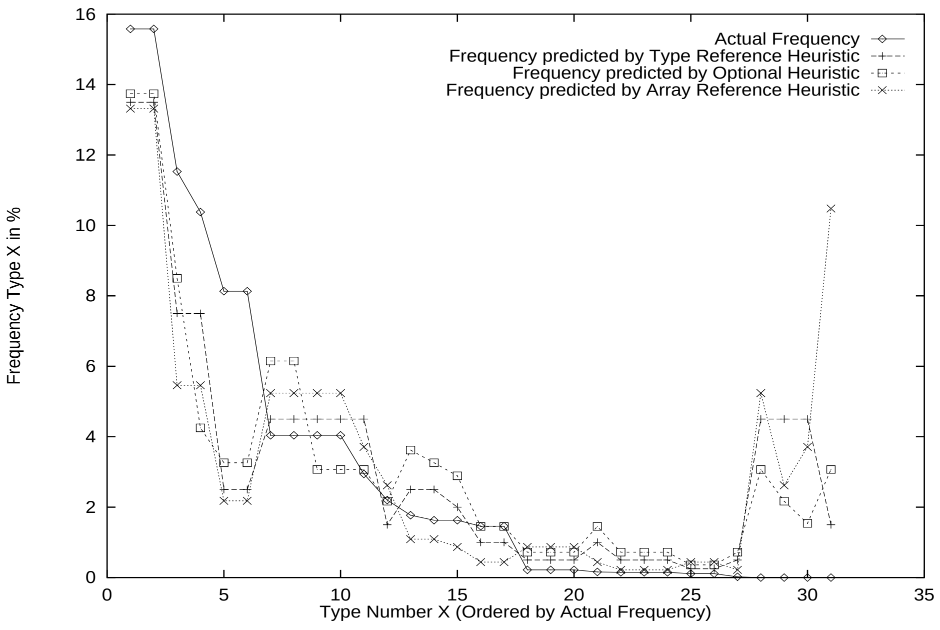
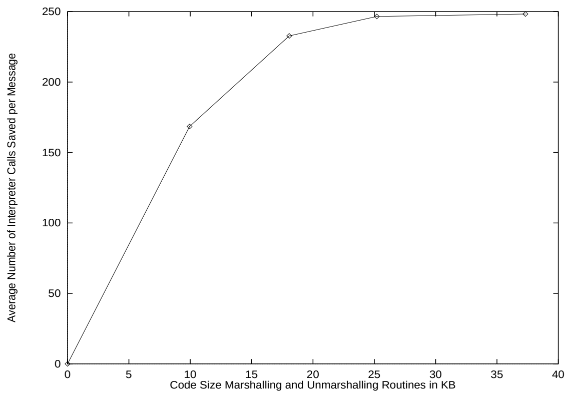


Figure 1: Experimental results of comparing prediction heuristics (X.400 P1 benchmark)



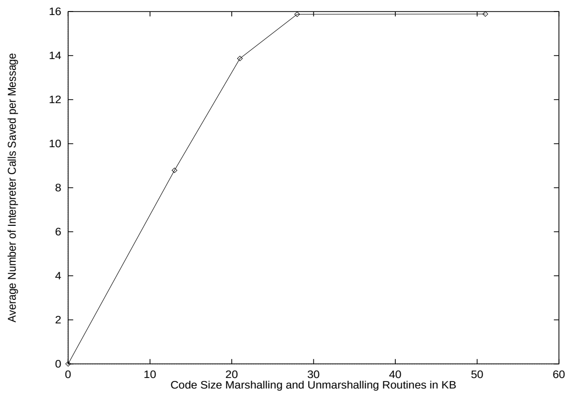


Figure 2: Locality in X.400-P1 (left) and Z39.50 (right) benchmarks

as follows: $S_\Delta = 100 * S_{opt}/S_{max}$, where $S_{max}$ is size of the code when full optimization is used ($x_i = 1$ for all $i$).

$S_\Delta$ was to 25%, 50%, 75% and 100%. Figure 2 shows the results of these experiments. On the x-axis, we give the code size measured for each value of $S_\Delta$. On the y-axis, we give the number of calls to the interpreter that are eliminated by generating procedure-driven code.

Setting $S_\Delta$ to 25% saves 68% of the interpreter calls for X.400 and 55% of the interpreter calls for Z39.50. Setting $S_\Delta$ to 50% saves 94% of the calls in X.400 and 87% of the calls in Z39.50. Thus, we find that due to the optimizer we generate presentation conversion code that is nearly as fast as fully optimized code, but requires only a small fraction of the maximal code size.

## 6 Conclusions

The results presented in this paper support the following three key insights:

- *Marshalling code exhibits locality.* For many applications, a large fraction of the speedup achieved by fully optimizing the presentation conversion code can be achieved by optimizing only a subset of the types in the interface specification. The reason for this is that some of the types defined in an interface specification are used far more frequently than others.

- *Locality can be detected by static analysis.* The number of times that a particular type in an interface specification will be used on run-time can be determined by mapping the type reference graph of the interface specification onto a Markov Model. Used in conjunction with a set of simple heuristics, the solution of these equations gives the frequency of each type in the interface with very good accuracy.

- *The size-speed trade-off can be solved using a Knapsack optimization model.* The optimization problem to be solved is to select the subset of types of an interface specification that should be optimized given a constraint on the maximal size of the presentation conversion code. This can be modeled as a well-known optimization problem (Knapsack problem), and thus solved with standard optimization algorithms.

In summary, the work in this paper has shown that it is possible to automate the size/speed trade-off for presentation conversion code, and has proposed and evaluated a particular optimization method. An experimental evaluation of this method on a set of benchmarks showed that by investing 25% of the code size required by fully optimized code, 55% to 68% of the interpreter calls could be eliminated. Increasing the code size investment to 50% of the maximal code size resulted in saving 87% to 94% of all interpreter calls.

The approach presented in this paper is not limited to the particular interface definition language used in the experiments (ASN.1). Since we start from a general model, it is straightforward to apply the same optimization approach in stub compilers for systems like CORBA, DCE or Java-RMI.

A potential limitation is the heuristic branch prediction approach. It is works well if the type reference graph of an application contains many references to a number of "central" data types. This is very likely in complex end user-oriented distributed applications such as databases or EDI, and these are the applications where the size of presentation conversion routines becomes problematic. As a fallback, Mavros offers the possibility to manually add frequency values to type references.

The results can be further improved by refining the method used for estimating the profit and cost of an optimization. For instance, the measurements presented in this paper show that the performance of marshalling and unmarshalling operations is not symmetric. One amelioration could be thus to introduce one optimization variable $x_i$ per (type, operation) pair, instead of using only an optimization variable per type.

## References

[1] Ball, T., & Larus, J. (1993). Branch Prediction For Free. In ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, (pp. 300-313).

[2] Castelluccia, C., & Hoschka, P. (1995). A Compiler-Based Approach to Protocol Optimization. Proceedings "Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems".

[3] Castelluccia, C., Dabbous, W.& O'Malley, S (1997). Generating Efficient Protocol Code from an Abstract Specification. In IEEE/ACM Transactions on Networking, 5(4), 514-525.

[4] Chung, S., Lazowska, E., Notkin, D., & Zahorjan, J. (1989). Performance Implications of Design Alternatives for Remote Procedure Call Stubs. In Distributed Computing Systems, (pp. 36-41).

[5] Clark, D., Jacobson, V., Romkey, J., & Salwen, H. (1989). An Analysis of TCP Processing Overhead. IEEE Communications Magazine, 23-29.

[6] Clark, D., & Tennenhouse, D. (1990). Architectural Considerations for a New Generation of Protocols. In ACM SIGCOMM '90, (pp. 200-208).

[7] Corbin, J. (1990). The Art of Distributed Applications. 1990: Springer.

[8] Graham, S., Kessler, P., & McKusick, M. (1983). An Execution Profiler for Modular Programs. Software - Practice and Experience, 13, 671-685.

[9] Hoschka, P., & Huitema, C. (1993). Control Flow Analysis for Automatic Fast Path Implementation. In A. Tantawy (Ed.), Second Workshop on High Performance Communication Subsystems, (pp. 29-33).

[10] Hoschka, P., & Huitema, C. (1994). Automatic Generation of Optimized Code for Marshaling Routines. In Manuel Medina & N. Borenstein (Ed.), IFIP TC6/WG6.5 International Working Conference on Upper Layer Protocols, Architectures and Applications, (pp. 131-146).

[11] Hoschka, P. (1995). Automatic Code Optimisation in a Stub Compiler. Ph.D. Thesis, University of Nice/Sophia-Antipolis.

[12] Huitema, C. (1991). MAVROS: Highlights on an ASN.1 Compiler (Internal Working Paper). INRIA.

[13] Huitema, C., & Doghri, A. (1989). Defining Faster Transfer Syntaxes for the OSI Presentation Protocol. ACM Computer Communication Review, 19(5), 44-55.

[14] Jones, M., Rashid, R., & Thompson, M. (1985). Matchmaker: An Interface Specification Language for Distributed Processing. In N. Meyrowitz (Ed.), 11th ACM Symposium on Principles of Programming Languages, (pp. 225-235).

[15] Kessler, P. (1994). A Client-Side Stub Interpreter. ACM SIGPLAN Notices, 29(8), 94-100.

[16] Martello, S., & Toth, P. (1990). Knapsack Problems. Chichester: John Wiley.

[17] McFarling, S., & Hennessy, J. (1986). Reducing the Cost of Branches. In 13th Annual Symposium on Computer Architecture, (pp. 396-403).

[18] O'Malley, S., Proebsting, T., & Montz, A. (1994). USC: A Universal Stub Compiler. In ACM SIGCOMM '94, (pp. 295-307).

[19] Pagan, F. (1988). Converting Interpreters into Compilers. Software - Practice and Experience, 18(6), 509-524.

[20] Partridge, C. (1993). Gigabit Networking. Reading: Addison-Wesley.

[21] Peterson, L. &Davie, B. (1996). Computer Networks: A Systems Approach.

[22] Pettis, K., & Hansen, R. (1990). Profile Guided Code Positioning. In ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, (pp. 16-27).

[23] Pittman, T. (1987). Two-Level Hybrid Interpreter/Native Code Execution for Combined Space-Time Program Efficiency. ACM SIGPLAN Notices, 150-152.

[24] Ramamoorthy, C. V. (1965). Discrete Markov Analysis of Computer Programs. In ACM National Conference, (pp. 386-392).

[25] Sample, M. & Neufeld, G. (1993). Implementing Efficeint Encoders and Decoders for Network Data Representations. In IEEE Infocom '93, (pp. 1144-1153).

[26] Sample, M. (1993). Snacc 1.1: A High Performance ASN.1 to C/C++ Compiler (Manual University of British Columbia, Vancouver.

[27] Scheifler, R. (1977). An Analysis of Inline Substitution for a Structured Programming Language. Communications of the ACM, 20(0), 647-654.

[28] Steedman, D. (1990). Abstract Syntax Notation One (ASN.1) The Tutorial and Reference. London: Twickenham Appraisals.

[29] Thekkath, C., & Levy, H. (1993). Limits to Low Latency Communication on High-Speed Networks. ACM Transactions on Computer Systems, 11(2), 179-203.

[30] Zahn, L., Dineen, T., Leach, P., Martin, E., Mishkin, N., Pato, J., & Wyant, G. (1990). Network Computing Architecture. Englewood Cliffs: Prentice-Hall.