Compact and Efficient Presentation Conversion Code

Philipp Hoschka

April 18, 1998

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

e-mail: hoschka@sophia.inria.fr

Abstract

Inefficient presentation conversion code is a major difficulty for using a distributed-applications development environment (such as a Corba-, Java-RMI-, DCE- or ASN.1-based environment) to build high speed network applications. The stub compilers included in these environments generate code that is either very slow, or has a large code size. This paper presents a technique for automatically generating compact and efficient presentation conversion code. This is achieved by using a hybrid of two implementation alternatives for presentation conversion routines (interpreted and procedure-driven code). The optimization is modeled as a Knapsack problem. A Markov model is used in combination with heuristic branch prediction rules for estimating execution frequencies. An optimization stage based on these ideas was implemented in the ASN.1 compiler Mavros. Experimental evaluation of this implementation shows that investing only 25% of the code size of fully optimized code results in a performance improvement of 55% to 68%.

1 Introduction

Text books on high-speed networking call presentation conversion "the performance problem of the 90's" (e.g. [21], [22]). This is because slow presentation conversion code frequently prevents applications to take advantage of the performance of high-speed networks.

It is well known that the performance of presentation conversion code is highly implementation dependent. Many stub compilers provide parameters to select one of several different implementation techniques ([8], [13], [16], [26], [31]).

However, selecting a high-performance implementation technique will increase the size of the conversion code considerably. For this reason, in practice, many applications are developed without using the high-performance code generation options provided by the stub compiler. This turns presentation conversion into the performance bottleneck when the application is used over a high-speed network.

This paper is based on the insight that compact and efficient presentation conversion code can be generated by using a hybrid of different implementation techniques. Rather than blindly generating all of the conversion code using a single high-performance implementation technique, these techniques should only be used selectively.

Designing an optimization stage built on this insight turns out to be challenging even at today's advanced state of the art in compiler construction [1]. Most traditional performance improvement techniques used by compilers either decrease code size (e.g. elimination of induction variables or unreachable code), or at least have no significant impact on code size (e.g. register allocation or instruction scheduling). Thus, a compiler can use these techniques wherever applicable. In contrast, the performance improvement techniques for presentation conversion code increase the code size to an extent that makes their automatic application impractical. A decision has to be taken on a case-by-case basis.

The paper shows how a stub compiler can automatically decide which parts of the code are worth optimizing. It makes three main contributions:

- The core algorithm for solving the size/speed trade-off problem in the generation of presentation conversion code is derived by mapping the optimization task 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 mapping on a Knapsack problem 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 generate *presentation conversion routines*. These routines are required for converting data values from a machine-internal format into a format suitable for network transmission at the sender (*marshalling*), and reverse the conversion at the receiver (*unmarshalling*).

Presentation conversion routines are generated starting from an *interface definition*. 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 conversion code for 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.

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.

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

Performance problems due to presentation conversion routines have been observed in [7], [14] and [30]. Code size problems have been reported when using stub compilers for developing practical systems, e.g. an X.500 directory service [26] or an operating system ([15], [16]).

USC [19] implements an optimization method based on inlining that significantly improves presentation conversion speed for TCP/IP packets. However, since the method is based on inlining, it cannot be applied to more complex, application-oriented interface definitions without incurring a significant code size overhead. 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.

[24] 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. [28] 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 ([9], [18], [23]). 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 [2]. 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 ([10], [11]) have been applied to solve a related size/speed trade-off problem in the area of automatic code generation of protocol automata in (HIP-PCO ([3], [4])). 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 that that can be easily mapped onto the type definition languages used in today's interface definition languages. 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.

Given an interface specification IF with a set of type definitions, we define the following variables: S is the total size of presentation conversion code for IF before optimization and S_{opt} is the total size of presentation conversion code for IF after optimization. Given a set of values of types defined in IF, T is the total execution time for converting these values before optimization and T_{opt} is the total execution time for converting these values once the code has been optimized.

The objective of optimization is to generate presentation conversion routines in such a way that T_{opt} is minimal under the constraint that S_{opt} does not exceed a given maximal code size.

For each type i in IF, we define:

- s_i : Size of code template for type i before optimization.
- s_{i-opt} : Size of code template for type i after optimization. This corresponds to the code size increase caused by the optimization.
- t_i : Time for converting values type i before optimization.
- t_{i-opt} : Time for converting values of type i after optimization. This corresponds to the overhead saving resulting from 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:

$$S = \sum_{i=1}^{n} s_i \quad (1)$$

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

$$T = \sum_{i=1}^{n} f_i t_i \quad (3)$$

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

The size/speed trade-off occurring when generating optimized presentation conversion code can be expressed as a *0-1 Knapsack problem* [17]. For this, each type definition node i in the syntax graph is assigned a profit p_i and a weight w_i as follows:

$$p_i = f_i(t_i - t_{i-opt}) \quad (5)$$
$$w_i = s_{i-opt} - s_i \quad (6)$$

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

maximize:

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

subject to:

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

In the following, it is 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 macro definitions rather than function definitions.

For solving the 0-1 Knapsack problem, we use the *Greedy algorithm* described in [17]. This algorithm is far easier to implement than any of the algorithms for determining an exact solution to the Knapsack problem. Investing much effort into finding an exact solution also seems inappropriate, given that the values for weights and profits are also only approximations (see below).

The algorithm works as follows: First, the list of items is sorted by their profit/weight ratio, so that

$$p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n \quad (9)$$

Then, items i are consecutively inserted into the Knapsack in the order of this list, and the remaining capacity of the Knapsack is reduced by w_i. The insertion process continues until the first item s is encountered that does not fit into the Knapsack's remaining capacity. Then, list items following s are examined and inserted if they fit into the remaining Knapsack capacity, until the capacity is zero or all items have been examined.

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 set of values to be converted. 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.

v_i expresses how often field i occurs on average in a value of type t for a given workload. For estimating v_i, the following cases can be distinguished:

- t is 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 λ in the case of recursive types.
- t is a union type: $v_i = 1/n$, where n is the number of fields in the union.
- t is 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.

We define the *type reference graph* of an interface specification as a tuple $\{V, ((i,j), v_i)\}$. The set of nodes V contains one element for each type in the interface definition. $((i,j), w)$ is a set of weighted arcs. It contains one element for each type i that references another type j (e.g. a structure type i containing a field of type j).

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 $((i,j), v_i)$. Following an approach proposed in [25], 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 set of linear equations.

Special care must be taken in in the case of optional fields. 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 detects loops in the type reference graph.

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. We therefore 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 saved when procedure-driven code is generated.

For a particular type i, we substitute $(t_{i-opt} - t_i)$ in (7) by a predictor $t_{\Delta i}$ and $(s_i - s_{i-opt})$ in (8) by a predictor $s_{\Delta i}$. The predictors are defined as follows:

- i is a structure 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

  - $s_{\Delta i} = 1 + n$, where n is the number of fields in the structure
  - $t_{\Delta i} = 1 + n$, where n is the number of fields in the structure

- i is a union type
  - $s_{\Delta i} = 1 + n$, where n is the number of fields in the union
  - $t_{\Delta i} = 2$. In a union type, the interpreter is called once for handling the type definition node, and once for handling the union field.
- i is an array type
  - $s_{\Delta i} = 2$. Generating procedure-driven code for an array concerns two nodes, the type definition node for the array, and the field node.
  - $t_{\Delta i} = 1 + \mu$. As previously defined, μ 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. All measurements in this section were performed 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 [13].

We used one benchmark for each of the four type constructors available in the interface definition language ASN.1 [29]. The *Sequence* type corresponds to a structure type in C. The *Set* type is a special case of a structure where fields can be encoded in any order. 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 fields.

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.

These numbers show a significant performance difference between interpreted and procedure-driven presentation conversion code. Compiled code is by a factor of 1.6 to 5.7 faster than interpreted code. Some of the numbers measured for interpreted code are

	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	Factor
X.400	17 KByte	37 KByte	2.2
Z39.50	25 KByte	51 KByte	2
FTAM	26 KByte	103 KByte	4
X.500	26 KByte	137 KByte	5.3
Total	94 KByte	328 KByte	3.5

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

not sufficient to saturate an Ethernet link (10 Mbit/s), and none of the alternatives can saturate network link with higher throughput characteristics such as FDDI (100 Mbit/s) or the ATM links commonly used with workstations (155 Mbit/s).

Thus, from a performance point of view, procedure-driven presentation conversion code is preferable to interpreted code. Making presentation conversion code faster, however, has the drawback of increasing its code size significantly. 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. X.400 defines the format of e-mail messages 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 numbers include both the marshalling and the unmarshalling routines. The interpreter itself takes up 8 KByte.

As can be seen from the numbers in Table 3, procedure-driven presentation conversion code is significantly larger than 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

For the next round of experiments, we implemented an optimization stage in the ASN.1 compiler Mavros that uses the frequency prediction method described in Section 4.2, and the predictors for code size and speed described in Section 4.3.

The first set of experiments evaluate the accuracy of the frequency prediction heuristics. For this purpose, we first manually estimated the values of the arc-weights v_i for the ASN.1

specification of an e-mail protocol (X.400 P1) (see [12] for details). We compared 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 (i.e. $\lambda = 1$), and the length of array values is set to 1 (i.e. $\mu = 1$).

- *"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 one (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. $\mu = 2$). λ was set to one.

Figure 1 compares the results of the manual frequency prediction with each of the prediction heuristics for the X.400 benchmark. The x-axis shows the type definition contained in the X.400 interface specification, ordered by their actual frequency. The y-axis shows the actual or predicted frequency of each type in percent.

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 observation. It contradicts a fundamental hypothesis used in optimizers for general-purpose programming languages, which is that a program spends most of its time in loops, and that therefore loop optimizations are more important than others.

Analyzing the global quality of the heuristic predictions by comparing it to the manual prediction, we find that all heuristics predict the two most frequently used types with excellent accuracy.

The excellent prediction results for type frequencies in X.400 P1 are due to the fact that electronic mail messages contain a *workhorse 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 specifications.

Many distributed applications contain a workhorse data type. 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 workhorse data type for both applications (*directory name* and *record*) ended up in the top 20% of the most frequently used types.

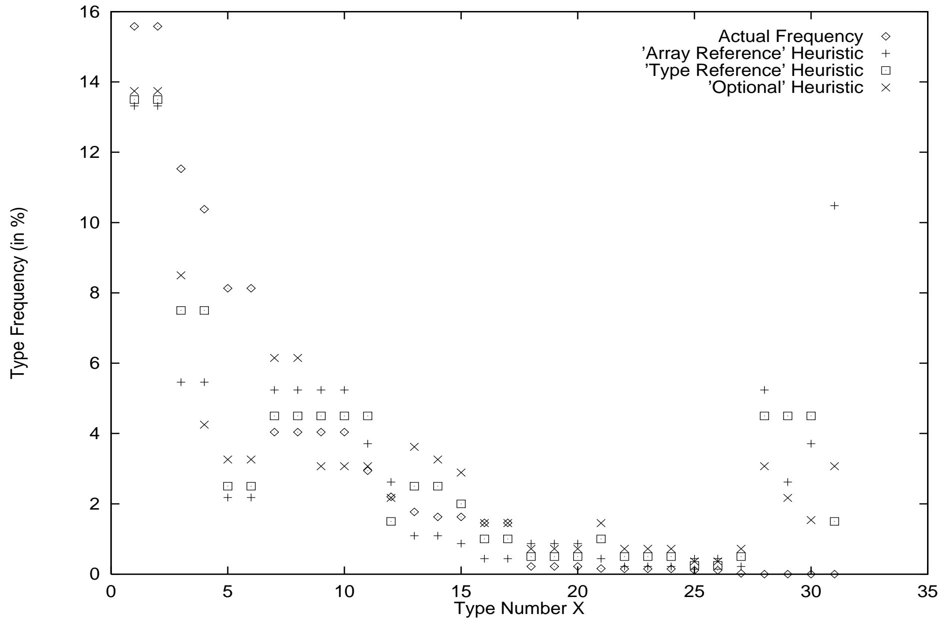

Figure 1: Experimental results of comparing prediction heuristics (X.400 P1 benchmark)

Generally, workhorse data types exist in complex end user-oriented distributed systems such as databases or EDI applications. These are applications where the size of presentation conversion routines becomes problematic. Workhorse data types may not exist in all applications, however, and therefore Mavros offers the possibility to manually add frequency values to type references as a fallback.

5.3 Size/Speed Trade-Off

In a final round of experiments, we evaluated the quantitative impact of the size/speed trade-off for marshalling routines using the extended Mavros compiler.

The experiments used the ASN.1 specifications for X.400 P1 and Z39.50 as benchmarks. In order to get precise measurements on the impact of the size/speed trade-off, the values of the arc-weights v_i were estimated manually (see [12] for details). In the experiments, we varied α, which is the compiler parameter controlling the code size increase due to procedure-driven code. α is a percentage value used by Mavros to determine the Knapsack capacity c in relation (8) as follows: $c = \alpha * S_{max}$, where S_{max} is the value of the left hand side of relation (8) when full optimization is used ($x_i = 1$ for all i).

α was set to 25%, 50%, 75% and 100%. For each of these α values, we measured the value of the objective function given in (7). Since the predictor $t_{\Delta i}$ in Section 4.3 measures interpreter calls, this gives the number of interpreter calls saved for a particular setting of α.

Figure 2 shows the results of these experiments. The x-axis shows the α values. The y-axis shows the number of interpreter calls that are eliminated by generating procedure-driven code in percent of the total number of interpreter calls required in non-optimized code.

The Figure shows that setting α to 25% saves 68% of the interpreter calls in the case of X.400 and 55% of the interpreter calls in the case of Z39.50. Setting α to 50% saves 94% of

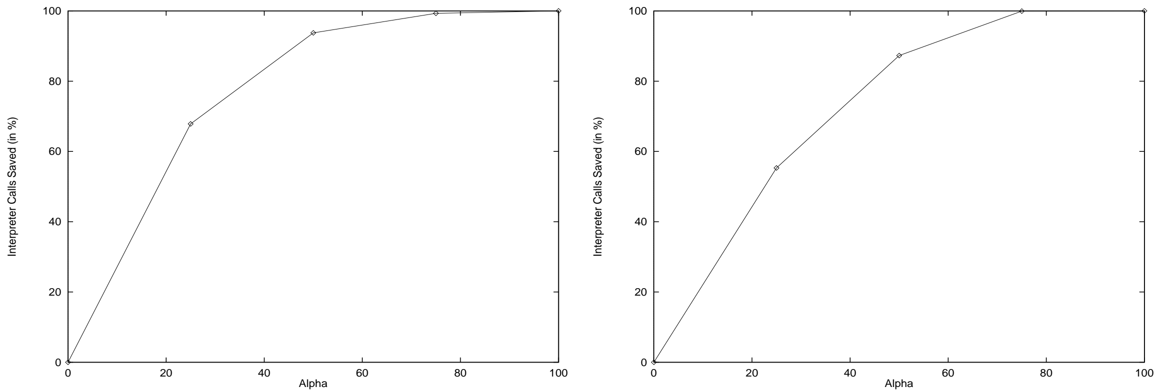

Figure 2: Size/Speed trade-off for X.400-P1 (left) and Z39.50 (right) benchmarks

the calls in the case of X.400 and 87% of the calls in the case of Z39.50. This demonstrates that after adding an optimization stage, Mavros generates presentation conversion code that is nearly as fast as fully optimized code, but requires only a small fraction of the code size overhead of full optimization.

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. This is because 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 estimated by mapping the type reference graph of the interface specification onto a Markov Model. Used in conjunction with a set of simple heuristics, this allows determining the most frequently used types with very good accuracy.
- *The size-speed trade-off can be resolved using a Knapsack optimization model.* The optimization problem to be solved is to select a 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 results demonstrate that it is possible to automate the size/speed trade-off for presentation conversion code using the optimization method described in this paper. An experimental evaluation of this method shows that by investing 25% of the code size required by fully optimized code, 55% to 68% of the interpreter calls can be eliminated. Increasing the code size investment to 50% of the maximal code size results in saving 87% to 94% of all interpreter calls.

The approach presented in this paper is based on a general model rather than on a particular interface definition language. Thus, it is straightforward to use the approach in

stub compilers for any interface definition language (e.g. CORBA, DCE or Java-RMI).

The approach can be further improved by refining the method used for estimating the profit and cost of an optimization. For instance, the measurements presented in Section 5.1 show that the optimization benefit varies for different type constructors and different operations. This is currently not reflected in the optimization model.

References

[1] Bacon, D., Graham, S., & Sharp, O. (1994). Compiler Transformations for High-Performance Computing. ACM Computing Surveys, 26(4), 345-420.

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

[3] 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.

[4] 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.

[5] 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).

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

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

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

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

[10] 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).

[11] 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).

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

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

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

[15] 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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