5
Automating Performance Optimisation by Heuristic Analysis of A Formal Specification

Philipp Hoschka
INRIA
2004, Routes des Lucioles, 069002 Sophia Antipolis, France email:
`hoschka@w3.org`

Abstract
Many advantages have been given for using formal specifications in the design and implementation of communication systems. Performance is usually not among them. It is commonly believed that code generated by an automatic tool from a formal specification is inherently slower than code implemented manually. This paper gives experimental evidence that this contention might be false. The key idea is to integrate heuristics used by a human programmer when optimizing code into the code generation tool. This way, the tool can generate code that is competitive with code written by a human programmer, and even better for specifications of sufficient complexity. Experiments were conducted using the presentation conversion routines generated by an ASN.1 compiler. The paper describes the design and implementation of an optimisation stage that automates the trade-off between code size and execution speed in these routines. For this purpose, a heuristic method to predict the frequency of type usage is developed, based on static control flow analysis of the type reference graph of an ASN.1 specification. Experimental results show that this approach can successfully identify the most frequently used types in a specification.

Keywords
Tools and tool support, ASN.1, Practical experience and case studies

1 INTRODUCTION

Modules in a distributed system often use different representations for the same data. Before such data can be exchanged between the different modules, it is necessary to reformat the data. This operation is commonly referred to as *marshalling* or *presentation conversion.*

The marshalling function has often been identified as a major performance bottleneck in network communication ((Huitema & Doghri, 1989), (Clark & Tennenhouse, 1990)). Our measurements indicate that marshalling can reduce the throughput seen by an application from a 155 MBit/s ATM link by a factor of two to fifty. This is has been confirmed by the findings of others (e.g. (Thekkath & Levy, 1993)).

Because individual marshalling code is required for each application, it has become common practice to generate this code using an automatic code generator referred to as *stub compiler*. The source language of such a tool is referred to as *interface definition language*. The marshalling code generated by these tools is often so inefficient that manual optimisation of time-critical sections of the code is required.

Given that marshalling code is already automatically generated, it is interesting to investigate whether marshalling code can also be optimised automatically. In this paper, we describe the design, implementation and performance evaluation of an *optimising stub compiler*.

The rest of this paper is structured as follows: Section 2 gives some basic definitions required for the understanding of the rest of the paper, and discusses related work. Section 3 describes the design of the optimisation stage. For this, a prediction algorithm for automatically finding "hot spots" is developed. Furthermore, a model for evaluating the cost and profit of optimisation alternatives is developed. Section 4 evaluates the performance of the "hot spot" prediction algorithm. Section 5 gives our conclusions.

2 BASIC CONCEPTS AND RELATED WORK

Marshalling fulfills roughly three different tasks: format conversion, linearisation and realignment. *Format conversion* is required to overcome differences in the data format (ASCII or EBCDIC character format, integer byte order, floating point formats). *Linearisation* is required for data structures that are stored in non-contiguous memory sections, such as dynamically allocated tree structures. *Realignment* is required for the components of record or structure types, since different CPU's use different rules for positioning these fields in main memory.

The different tasks are fulfilled by different parts of the marshalling code. Format conversion is done by specialised algorithms that we will refer to as *marshalling primitives*. These algorithms depend on the particular conversion task at hand, and thus are best optimised individually. The optimisation considered in this paper concern the control code part of marshalling routines. This code sequences the application of the marshalling primitives to individual fields of a message, and also accomplishes the linearisation and realignment task.

Three alternative implementation techniques are commonly used for the control code of marshalling routines - *interpreted* code, *procedure-driven* code (also referred to as *compiled* code) and *inlined* code (Chung, Lazowska, Notkin, & Zahorjan, 1989). These three techniques have a well-known size-speed trade-off. Interpreted code is compact, but slow, inlined code is fast but memory-consuming and compiled code lies somewhere in between these two alternatives.

In order to determine the practical impact of the different code generation alternatives on communication time, we measured the execution time of a 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

	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 1 Impact of compiled and interpreted marshalling on throughput

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 1 gives the throughput measured in these experiments for interpreted and compiled marshalling code. For each experiment, we report three values: the absolute throughputs of interpreted and compiled code, and the factor by which interpreted code is slower than compiled code. The latter serves to eliminate system-dependencies inherent in the absolute values. All absolute numbers were measured on a Sun Sparc 10, Model 40, using gcc version 2.6.0, static linking and optimisation level 2 (O2). The marshalling code was generated by the ASN.1 compiler Mavros (Huitema, 1991).

From these numbers we see that the speed difference between interpreted and compiled marshalling code is significant. Compiled code is consistently faster than interpreted code by a factor of 1.6 to 5.7. Unmarshalling is always slower than marshalling due to the requirement for error-checking. 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).

We conclude that from a performance point of view the use of compiled marshalling code is preferable to the use of interpreted code. More importantly, the choice of the implementation technique for marshalling code can decide whether the installation of expensive high-speed network was worthwhile for speeding up a particular implementation. Installing an ATM network is of little use if the marshalling code of an application executes at Ethernet throughput.

However, making marshalling code faster has the drawback of increasing its code size. Table 2 shows the difference in code size between interpreted and compiled marshalling code when generating marshalling code for four different applications whose interfaces are specified in ASN.1. 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. The numbers include both the marshalling and the unmarshalling

	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 2 Impact of compiled and interpreted marshalling on code size

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.

As can be seen from the numbers in Table 2, compiled marshalling code takes up significantly more object code size than interpreted code. This becomes particularly clear when comparing the aggregate code sizes. The aggregate code size is important, since users generally run several different network applications in the background while using "local" applications such as word processors.

With current stub compilers ((Zahn, Dineen, Leach, Martin, Mishkin, Pato, et al., 1990), (Corbin, 1990), (Huitema, 1991), (Sample, 1993), (Kessler, 1994), (O'Malley, Proebsting, & Montz, 1994)), the user of a stub compiler can only choose between one of the three points on the Space/Speed curve by setting a parameter for the implementation technique that the stub compiler should use when generating marshalling code. In practice, it appears more useful to enable the user to specify a bound on the maximum code size that should be used for marshalling code. This size may lie in between the code sizes resulting from using one of the three "pure" code generation techniques. The stub compiler can thus use more code space and thus produce faster code for meeting the user's code-size constraint than by simply generating interpreted code. The goal of generating execution-time optimal code under a size constraint can be met by generating a hybrid between the different implementation techniques (Pittman, 1987).

3 GENERAL MODEL FOR STUB OPTIMISATION

3.1 Hybrid Marshalling Routines

Many of today's stub compilers allow the user to influence the performance and the code size of the marshalling code for a particular application by choosing one of several code generation strategies such as function inline expansion for scalar types and procedural code or interpreted code for type definitions. However, once the user has fixed the code generation strategy, the stub compiler uses this strategy uniformly to generate marshalling code for all types in the input interface specification.

The realities of today's networking environments often require the implementation of high performance marshalling routines on a machine with memory size constraints. This requires a more flexible code generation strategy than available in current stub compilers. Instead of using the same strategy for all types, different code generation strategies should be used for different types, resulting in *hybrid marshalling routines.*

Using a hybrid code generation strategy in a stub compiler is promising due to a general heuristic in computer science, the principle of locality or "80/20 rule". This principle says that programs spend a large part of their execution time in a small part of the program code (Knuth, 1971). Research on the optimisation of TCP transport protocol implementations has shown that this principle also holds for communication code: large performance improvements have been achieved by optimising only a small part of the implementation (Clark, Jacobson, Romkey, & Salwen, 1989). However, the TCP performance improvements described are achieved by manual code optimisation.

Automatic code optimisation requires a model of how optimisation decisions made by the compiler influence the code size and execution speed of the generated marshalling code. The selection of a code generation strategy for a particular type depends on all of the following three factors: (1) the frequency with which the type occurs at run-time, (2) the execution times of the marshalling routines for the different code generation alternatives and (3) the code size increase incurred by each of the alternatives.

We develop the details of this model in the following sections.

3.2 Basic Definitions

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 optimised marshalling routines starts from an intermediate representation of the interface definition. We define the *syntax graph* of an interface specification to be a tuple $\{V, E\}$ where V is the set of nodes in the syntax graph for which an optimisation 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 corresponds to a type definition node in the syntax graph. Each type definition node has a name. Moreover, each type definition node is labelled by the type constructor of the type definition.

Each field in a constructed type corresponds to a *field node* in the syntax graph. Field nodes are labelled by their class. A field node can be a constructed type (*embedded type constructor*), a reference to a pre-defined scalar type (*scalar reference*) or a reference to another type definition (*type reference*).

A non-optimising stub generator will traverse the syntax graph once and generate marshalling code for each node in the graph. The code generated for each node is determined by a fixed code template depending on the node's characteristics. For example, the code template for a structure type node might consist of a function call, followed by some initialisation code specific to structure types, followed by a place holder for the code for the structure fields and terminated by procedure return code.

An optimising stub generator must select one of several alternatives code templates for each node in such a way that the generated code fulfills a given optimisation criterion. The exact alternatives to be considered depend on the class of the node. For a type definition node, the compiler must decide whether interpreted or compiled code should be generated. For a field node, the compiler must decide whether the marshalling routine for the field should be written inline.

3.3 Model Variables

For formalising the optimisation problem occurring in the generation of hybrid marshalling routines, we define the following variables: S is the total size of marshalling code before optimisation, S_{opt} is the total size of marshalling code after optimisation, T is the total execution time of marshalling code before optimisation for a given workload and T_{opt} is the total execution time of marshalling code after optimisation for a given workload.

The objective of optimisation is then to generate marshalling in a way that minimises T_{opt} under the constraint that S_{opt} does not exceed a given maximal code size. The values of T_{opt} and S_{opt} can be calculated using the nodes in the syntax graph. We define:

- s_i : Size of code template for node i before optimisation.
- s_{i-opt} : Size of code template for node i after optimisation. This corresponds to the code duplication caused by the optimisation.
- t_i : Time for marshalling node i before optimisation.
- t_{i-opt} : Time for marshalling node i after optimisation. This corresponds to the overhead saving achieved by the optimisation.
- f_i : Execution frequency of marshalling code for node i for a given workload.
- x_i : $x_i = 1$ if node i is optimised, 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 optimised marshalling code can be expressed as a *0-1 Knapsack problem* (Martello & Toth, 1990). For this, each node i in the syntax graph is assigned a profit p_i and a weight w_i as follows:

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

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

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

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

subject to:

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

It can be assumed that both the profit and the weight are positive numbers. A negative weight corresponds to an optimisation that does not increase the code size, and thus should be applied in any case. 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. A negative profit value corresponds to an optimisation that increases the execution time, which is impossible by definition.

For solving the Knapsack problem, the stub compiler must have information on both profit and weight for each node. Both values must generally be estimated. One reason for this is that the stub compiler generates code in an application programming language. Thus, the final absolute code size and execution speed of the marshalling code depend on the machine code generated by the application language compiler. Moreover, the profit of an optimisation is a function of the frequency with which the optimised code is executed on run-time. This frequency will depend on the actual workload. Only estimates of this workload are available at the time the marshalling code is generated.

3.4 Predicting Execution Frequencies

Traditionally, *execution traces* have been used for finding program parts that can benefit from code optimisation ((Graham, Kessler, & McKusick, 1983),(Pettis & Hansen, 1990), (McFarling, 1991)). However, this approach makes code optimisation very time-consuming for the application programmer. The overhead is even higher when optimising a distributed program, since wo or more program modules must be optimised independently.

Due to the practical problems with using execution traces, *static control flow analysis* has been proposed recently as an alternative approach (Ball & Larus, 1993). This work motivated proposals to use static control flow analysis for automatic fast path implementation in protocol code in general (Hoschka & Huitema, 1993) and to experiment with static predictors for marshalling code in particular (Hoschka & Huitema, 1994).

Static control flow analysis of a syntax graph of an interface definition must determine two related values. For a type definition node T, we want to compute the number of times values of type T will occur on run-time in order to decide whether the marshalling routine for T should be interpreted or compiled. For a field node f, we want to determine how often f will occur on run-time in order to decide whether the marshalling routine for the field should be written inline or not.

Intuitively, it seems clear that types that occur as fields of an array type will occur frequently, that structure fields that are marked as "optional" will not occur frequently and that types which are referenced frequently by other types will also occur frequently. In other words, the user already has given hints to the stub compiler which parts of the interface specification will be executed frequently, simply by the way in which the interface specification is written ! In the following, we will present an approach to formalise these intuitions and extract the frequency information from the interface specification.

By analysing the control flow within a type definition, we can estimate the frequency with which each field contained in the type definition will occur in the values of the type. We define:

- f_i : frequency of the field node i in a given workload
- f_t : frequency of the type definition node t containing field node i in a given workload
- v_i : the average number of occurrences of field i in all values of type t in a given workload, $v_i = f_t/f_i$. We will refer to v_i as the *field frequency* of i.

For estimating v_i, we use the following rules:

- if field i is part of a union type with n elements, then v_i is equal to $1/n$ or a user-defined value.
- if field i is part of a structure type and field i is not marked as optional, then v_i is equal to one. If field i is marked as optional, v_i is equal to x, where x can be either a default value or a user-provided value between zero and one. As explained below, special care must be taken when calculating x in the case of recursive types.
- if the origin of an arc is an array type, then the arc weight is equal to y, where y can be either a default value, a user-provided value or the array length of a fixed length array in the interface specification.

For structure arcs that are marked as optional, we could e.g. choose a default value of 0.5 in most cases. This means that by default there is a 50% chance that an optional field in a structure type will actually occur in a value of this type. The heuristic assumption that optional fields are infrequent is justified by the nature of protocol development and implementation: optional fields in an interface specification often arise due to factors such as staying backward compatible with a previous version of the application or resolving "political" conflicts within the group that develops the interface specification. In an implementation of the application, only the mandatory components are guaranteed to be implemented. Optional components can be left out.

Care must be taken, though, since most interface definition languages use optional fields for a second purpose, namely to indicate a recursive type definition. To arrive at a legal flow graph, it must be ensured that the flow along a recursive path is smaller than 1. Otherwise, the recursion would not terminate. Thus, an additional pass is required in the calculation of v_i for distinguishing between these two different uses of the "optional" annotation by detecting loops in the type reference graph.

Note that we allow for user-defined values to replace the heuristic predictions for fields in union types, optional fields and array types. This way, the user can replace heuristic predictions by estimates of his own. This is usually only required for the types that represent message specifications. Pre-defined values can also be used by a general protocol compiler to communicate frequency predictions derived from a protocol control flow graph to the code generator for marshalling stubs.

A second level of control flow is defined by the references between type definitions in an interface specification. We define the *type reference graph* of an interface specification as a tuple $\{V', (E', w)\}$, where V' corresponds to the set of all type definition nodes in the interface specification's syntax graph. E' is computed by locating all field nodes in the syntax graph that are type reference nodes, and adding an arc from the type definition node containing the type reference node to the type definition node that is referenced. w is an arc weight that corresponds to the value of v_i of the field node at which the arc starts.

The type reference graph can be interpreted as a Markov model. Type nodes correspond to the states of the Markov model, and arc weights to the transition probabilities between the states. Following an approach proposed in (Ramamoorthy, 1965), the graph can be mapped onto a set of linear equations that represents the equations for determining the visit counts of the nodes. This system can be solved to compute the frequency of each type definition node. By multiplying this frequency with the v_i for each field, the frequency of the field nodes can be determined.

3.5 Trading Off Interpretation and Compilation

Due to space restrictions, we can only discuss automatic ways for an optimal trade-off between interpretation and compilation in this paper. The exact semantics of "interpreted code" and "compiled code" may vary from implementation to implementation. Generally, with interpreted code, the syntax graph of the interface specification is mapped onto a sequence of commands. With compiled code, the syntax graph is mapped onto a sequence of instructions in the application programming language.

The problem of deciding between interpreted and compiled code generation can be modelled as a Knapsack problem. The items to be included in the Knapsack are all type constructors in the syntax graph. In the following, we give general formulas for calculating the execution time and the code size before and after optimisation. From these formulas, the local profit and weight for each optimisation can be derived.

If a type constructor node is interpreted, one interpreter command is generated for the type constructor and one for each field node of the type constructor. Thus, for structure and union types, $1 + n$ commands are required, where n is the number of field nodes in the type constructor. Array types require two commands, one for the type constructor and one for the array element type. We assume that all commands have the same size, and define a to be the size of interpreter command.

If compiled code is generated for a type constructor, the code templates of the constructor and all its fields are written to the marshalling code. The size of a code template depends on the label of a node. For example, different templates will be used for structure types and choice types. We define $S(l)$ to be the size of code template for node with label l, and $l(j)$ to be the label for node j. Table 3.5 shows the resulting size estimation

Type constructor node k	s_k	s_{kopt}
structure	$(1+n)a$	$s(struct)+\sum s(l(j))$
union	$(1+n)a$	$s(union)+\sum s(l(j))$
array	$2a$	$s(array)+s(l(j))$

Table 3 Formulas for estimating size of interpreted and compiled marshalling code

formulas for all three type constructors. The index variable j in the sums runs over the fields of each type constructor.

If a type constructor node is interpreted, the execution of each command incurs a certain interpretation overhead that depends on the exact implementation strategy chosen for the interpreter. Moreover, time is required for executing the code for marshalling the type constructor and each of its fields.

We define I to be the interpretation overhead per command and $t_i(l)$ to be the execution time required for interpreted marshalling a node with label l.

For a structure type, the marshalling code for the structure type constructor and all of the structure fields will be executed. For a union type, the marshalling code for the union type constructor and one of the union components will be executed. The average time for a union type constructor cannot be calculated without reference to the frequency of execution of each union component for a given workload. We define $c = f_i/f_j$ where f_i is the frequency of the union constructor node i and f_j is the frequency for each component j of i in a given workload.

Similarly, for a variable length array with field node j the exact execution time cannot be calculated without knowing how many array elements occur in the workload, i.e. without knowing f_j . For a fixed length array, f_i is equal to the array length.

If a type constructor node is compiled, the execution time will be equal to the execution time for the code template for the type constructor plus the execution time for the fields. We define $t_c(l)$ to be the execution time required for compiled marshalling a node with label l, and $l(j)$ to be the label for node j.

Table 4 shows the resulting time estimation formulas for all three type constructors. The index variable j in the sums runs over the fields of each type constructor.

In practice, not all the variables in the model have to be measured for a given stub compiler. For instance, in many cases the size of a code template will be related to its execution time. This assumes that all calls to the code template will execute all instructions in the template, or at least that all calls spend the same fraction of time in the code template. In this case, it may be sufficient to only measure the size of the code template, and take it as an estimate for the execution time t. Moreover, if the compiled code is derived in a systematic way from the interpreted code as described in (Pagan, 1988), the code templates used for scalar types in the compiled code and in the interpreter are identical.

For solving the 0-1 Knapsack problem, we use an approximate algorithm which has the advantage over exact algorithms that it is easy to implement. Moreover, investing

Type constructor node k	t_k	t_{kopt}
structure	$(1+n)I + t_i(struct) + \sum t_i(l(j))$	$t_c(struct) + \sum t_c(l(j))$
union	$2I + t_i(union) + \sum ct_i(l(j))$	$t_c(union) + \sum ct_c(l(j))$
array	$2I + t_i(array) + f_j t_i(l(j))$	$t_c(array) + f_j t_c(l(j))$

Table 4 Formulas for estimating time of interpreted and compiled marshalling code

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.

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).

4 PERFORMANCE EVALUATION

4.1 Measuring Locality

It is intuitively clear that some of the messages and types in the interface definitions of distributed applications will be used more frequently than others. This is because they accomplish the "real work" of the application. What remains to be shown is that this locality can be exploited to arrive at fast and compact marshalling routines. This is not immediately clear, since it might be the case that nearly all types of the interface specification are used in the most frequently used messages. In this case, many types of the interface specification would have to be optimised, and the difference in code size between full optimisation and optimisation taking into account the locality of interface usage would be negligible. Ideally, we should find that by using our optimiser we arrive at marshalling code that is nearly as fast as fully optimised code, but requires only a small fraction of the size of fully optimised code.

For evaluating the impact of locality on the code size and the execution time, we repeated the experiment used for validating the accuracy of the size estimate for compiled code. In these experiments, we used varying values for the compiler switch c, which is defined as the ratio between the optimised code produced by the compiler and the code size when maximal optimisation is used, i.e.: $c = 100 S_{opt}/S_{max}$ Figure 1 shows the results

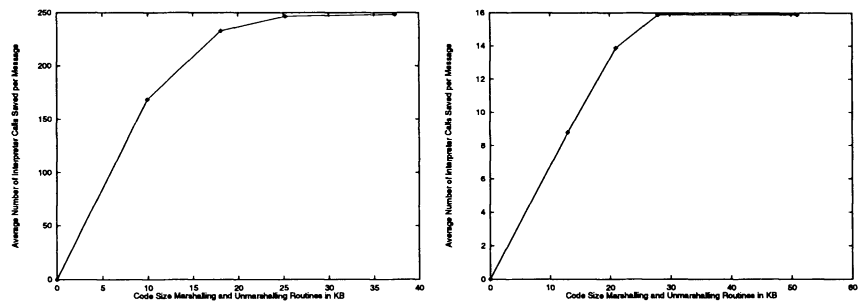

Figure 1 Locality in X.400-P1 (left) and Z39.50 (right) benchmarks

of these experiments for the X.400 P1 and the Z39.50 benchmark. On the x-axis, we give the size measured for each value of *c*. On the y-axis, we give the number of calls to the interpreter that are eliminated by generating compiled code. This serves as an estimate for the time saving achieved by the optimisation.

For both benchmarks the locality in the message use impacts code size and execution time. When 25% of the maximal code size for compiled code is invested, 68% of the interpreter calls can be eliminated in the case of X.400, and 55% of the interpreter calls can be eliminated for Z39.50. Investing 50% of the maximal code size eliminates 94% of the calls in X.400, and 87% of the calls in Z39.50. The maximal code size measured for these benchmarks is relatively low. However, it should be remembered that the numbers reported in these measurements are for two applications only. The object code size required for the full set of Mavros-generated routines can reach up to 1.5 MB when running X.500, FTAM, Z39.50 and X.400 in parallel on a Sparc system.

4.2 Validating Automatic Frequency Prediction

We compare the manual frequency estimates with the performance of three different prediction heuristics:

- *Type reference heuristic*: Using this heuristic, the number of times a type T is referenced by another type is used for predicting T's frequency. For this purpose, the field frequency of all optional fields and the length of all array types is set to one.
- *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. All array types are assumed to have a length of one.
- *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.

Figure 2 compares the results of the manual frequency prediction with each of the prediction heuristics in turn for the X.400 benchmark. We start with analysing the global quality of the heuristic predictions when compared to the manual prediction. By looking first at the most frequently used types to the left of the graphs, we see that in all heuristics the most frequently used two types are predicted with excellent accuracy. For the types that are never used that show up to the right of the figures, all prediction heuristics have an error, since they cannot predict that a type's frequency will be equal to zero.

The most severe error is introduced by the array heuristic. With this heuristic, a type that is never used in practice is put into the top 20% of the most frequently used types. This is because the experiment with the array heuristic did not take into account optional field references. Setting the coefficient for type references in optional fields to a lower value alleviates this problem. In contrast, repeating the experiment with an optional coefficient of 0.5 and an average array length of 5 had the effect that the type in question was predicted to be the most frequently used type in the X.400 specification.

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. This is in contrast to standard hypotheses used in general program optimisation, which concentrates on loops, i.e. components in the control flow graph that are used repeatedly such as arrays. In X.400, 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.

Using these experimental results, we can estimate the efficiency of the optimised code. The interesting question is how many of the values that occur on run-time are marshalled by optimised code. Assume that optimised code is generated for 20% of the type specifications in X.400. From Figure 1 we can calculate that in this case between 61.5% (Type reference heuristic and Optional heuristic) and 57% (Array heuristic) of the types occurring dynamically would be marshalled by optimised code.

The excellent prediction results for type frequencies in X.400 P1 are due to the fact that electronic mail envelopes contain a "central" data type, i.e. the e-mail address. Addresses are contained in many places of the envelope: 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.

Remember that the speedup achieved by optimising the marshalling code for a particular type definition is a product of the frequency of use and the complexity of the control flow through the type definition. Thus, if for example X.400 e-mail addresses would consist of a single integer type, the speedup of using the predicted frequency to guide the application of optimisations would be low. However, complex end user-oriented applications where the size of marshalling routines becomes problematic generally often also have complex "central" data types. Examples of such types are the "record" type used in Z.39.50 for transmitting the results of a query, and the "directory name" type used in X.500 for transmitting the results of a directory lookup. We also applied static frequency prediction to X.500 and Z39.50, and found that the "central" data structure always ended up in the top 20% of the most frequent types.

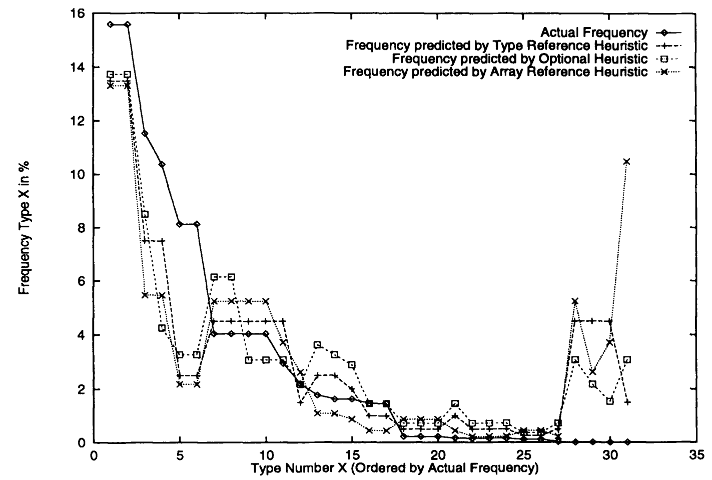

Figure 2 Experimental results of comparing prediction heuristics (X.400 P1 benchmark)

5 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 optimising the marshalling code can be achieved by optimising 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 control flow 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 system of control flow equations. 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 optimisation model.* The problem of selecting the subset of types of an interface specification that should be optimised given a constraint on the maximal size of the marshalling code can be modelled as a classical optimisation problem (Knapsack problem).

An experimental evaluation of this approach on a set of benchmarks for trading off interpreted and compiled code showed that by investing 25% of the code size required by fully optimised 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.

Our results on the optimisation of marshalling routines point to several interesting areas for future research: the development of distributed system benchmarks and the application of control flow graph analysis to general protocol automata specified in a formal description language.

One of the main difficulties we faced in our work was finding benchmarks that are suitable for evaluating our proposed optimisation techniques. There is a pressing need for defining benchmark distributed applications that can gain a similar widespread acceptance as the SPECmarks for non- distributed applications. This would facilitate immensely the further investigations into compiler optimisation techniques for distributed software.

The idea of using control flow analysis for finding points of locality in an interface specification can be extended to the analysis of a full protocol automaton. This is because from the point of view of compiler construction, a protocol automaton is nothing else but a control flow graph. Automating this optimisation is particularly interesting if the protocol automaton is itself derived automatically by a tool. This can be done starting from a high-level specification of the synchronisation requirements of a distributed application in a formal description language such as Estelle (Budowski & Dembinski, 1988), SDL (Belina & Hogrefe, 1989) or Esterel (Berry & Gonthier, 1992). Work in this direction has been started (Castelluccia & Hoschka, 1995).

In summary, the work presented has shown that it is possible to automate the optimisation of marshalling code, and has pointed out ways for automating general communication code. It was found that adding an optimisation stage to an automatic code generator of communication software is worthwhile, and adds considerable leverage to the optimisations implemented in standard application language compilers. This is because an optimiser implemented in a stub generator has access to the domain- specific semantic information expressed in the special-purpose languages used to formally specify the communication software. Therefore, a stub compiler can make more informed optimisations based on domain-specific heuristics than a compiler for a general-purpose language.

REFERENCES

Aho, A., Sethi, R., & Ullman, J. (1986). Compilers - Principles, Techniques and Tools. Reading: Addison-Wesley.

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

Belina, F., & Hogrefe, D. (1989). The CCITT-Specification and Description Language SDL. Computer Networks and ISDN Systems, 16(4).

Berry, G., & Gonthier, G. (1992). The Esterel Synchronous Programming Language. Journal of Science of Computer Programming, 19(2), 87-152.

Budowski, S., & Dembinski, P. (1988). An Introduction to Estelle. Computer Networks and ISDN Systems, 14.

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",

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).

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

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

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

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

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).

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).

Huitema, C. (1991). MAVROS: Highlights on an ASN.1 Compiler (Internal Working Paper 5). INRIA-RODEO .

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

Kessler, P. (1994). A Client-Side Stub Interpreter. ACM SIGPLAN Notices, 29(8), 94-100. Knuth, D. (1971). An empirical Study of FORTRAN Programs. Software - Practice and Experience, 1, 105 - 133.

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

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

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

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

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

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

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

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

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

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

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