Published in: Manuel Medina & N. Borenstein (Ed.), IFIP TC6/WG6.5 International Working Conference on Upper Layer Protocols, Architectures and Applications, (pp. 131-146), Barcelona. Amsterdam: North Holland.

Automatic Generation of Optimized Code for Marshalling Routines

Philipp Hoschka[1] *and Christian Huitema*
INRIA, Projet RODEO, 2004, Route des Lucioles B.P. 93, 06902 Sophia-Antipolis Cedex (France)
e-mail: {hoschka | huitema}@sophia.inria.fr

We describe a new approach to the automatic generation of marshalling code which results in code that is fast and compact at the same time. The key insight behind our work is that certain types in an interface specification occur more often than others at run-time. We exploit this locality to solve a particularily difficult optimization problem in stub generators, namely the trade-off between compact but slow interpreted marshalling routines and fast but large compiled marshalling routines. We resolve this trade-off by using a hybrid between the two techniques. In a simulation study with ASN.1/BER we find that the hybrid approach can lead to fast and compact presentation conversion routines: the results indicate that for the X.400 e-mail protocol 50 % of the speed benefit of the compiled marshalling technique can be achieved by investing only 20 % of its code size overhead.

1. Introduction

Stub generators for marshalling routines translate a set of type specifications into the programming language code that is required to exchange values of these types over a communication link. Today, the code generation stage in a stub generator is generally very simple. In particular, the type specification is not analyzed with the goal of finding a possibility of optimizing the program code generated. In this paper, we give practical evidence that the introduction of an optimization stage into a stub generator can result in substantially improved quality of the generated code in terms of code size and execution speed.

A particular problem in code generation for marshalling routines is the trade-off between compact but slow interpreted marshalling routines and fast but large compiled conversion routines. To combine the size advantage of interpreted marshalling routines with the speed advantage of compiled marshalling routines, we propose to use a novel hybrid approach that uses both techniques in the marshalling routines for a single application.

We aim at incorporating hybrid conversion routines into a stub generator in such a way that little if any decision-making input is required from the user. Consequently, we have to find an algorithm to determine for which types we generate compiled conversion routines, and for which types we generate interpreted conversion routines. Clearly, given a constraint on absolute code size, an “optimal” strategy would use compiled conversion for a subset of types so that the “return-on-investment” in terms of speedup is maximized, and interpreted conversion for the rest of the types. However, from the Knapsack problem it can be shown that an algorithmic solution to this optimization problem is NP-complete. Fortunately, the Knapsack problem is a rather “well-behaved” NP-problem, so that heuristics of good quality exist, and even a solution that is guaranteed to be optimal can be computed without excessive

1. Supported by an individual fellowship of the Human Capital and Mobility Programme HCM of the Commission of the European Communities (grant no. ERBCHBICT920005)

computational overhead for small, but interesting problems. Our contribution will facilitate the implementation of stub generators that offer a more flexible and seamless trade-off between code-size and execution speed than available in today's stub generators, without increasing the specification overhead seen by the user of these tools.

A common task in a distributed application is *marshalling* or *presentation conversion* of values contained in the messages that are exchanged between the distributed parts of the application. Before the message goes out on the network, a marshalling routine goes through the list of values transmitted and converts each value in turn from the local host representation into the common network data format. At the receiver side, the network data representation is converted into the local data representation before the message is processed.

Generally, a stub generator is based on a particular type declaration language and generates code for one or more network data formats. Both the type specification language and the network data format contain a set of scalar data types such as integer, real or string types and facilities to combine these scalar types into composite types by using constructors for record/ struct, union or array types. For the purposes of this paper, we will refer to the set of type declarations for a particular application as the application's *itnterface specification*. (i.e. we abstract from the fact that an interface specification is generally expressed in terms of procedure interfaces, e.g. for remote procedure calls. This is justified, since only the type declarations for the input argument and results of a procedure are relevant for a stub generator for marshalling routines.)

Different alternatives for the type declaration and network data format exist. This paper is based on the combination of the type declaration language ASN.1 (Abstract Syntax Specification Notation 1) and the network data format BER (Basic Encoding Rules). These are used in all OSI application layer standards such as X.400 and X.500. However, they are also used outside the realms of OSI, e.g. for the WAIS information retrieval protocol, the SNMP network management protocol and parts of Signaling System 7.

This paper is organized as follows: in Section 2, we discuss related work. In Section 3, we motivate the use of a hybrid approach between interpreted and compiled marshalling routines, by giving a detailed explanation and example code for both techiques. In Section 4, we discuss the problems of integrating our new approach into a stub generator, and develop solutions to these problems. Section 5 gives an outlook on future work.

2. Related Work

The time spent in marshalling routines has been identified many times as a dominating factor in the overall cost of network communication ([Huit89], [Clar90], [Huit92]). [Lin93] recently challenged this result, arguing that "under certain favorable but realistic assumptions" a throughput for BER marshalling can be achieved that is equivalent to the throughput of a TCP/ IP/ATM implementation (20 Mbit/s). [Lin93] further argued that the impression of marshalling into BER being slow was caused by the use of stub generators that generated inefficient marshalling code.

While it is true that the measurements presented in [Clar90] were based on a prototype stub generator contained in ISODE that generated very slow marshalling routines, it is also true that stub generators have improved considerably in recent years: [Samp93a] reports a speed-up of a factor of 6.5 between the performance of his ASN.1 stub generator snacc [Samp93c] and the performance of the tool included in ISODE. Similar results have been obtained for our ASN.1 stub generator Mavros [Huit91].

Despite of these recent advances in the performance of conversion routines, marshalling can

still be a very expensive operation. This is clarified by a number of experiments reported in [Huit92] which evaluate the performance of an optimized ASN.1 stub generator. The following experiment showed the high impact of marshalling cost on overall communication cost: application to application throughput was measured between two DEC 5000/200 using the TCP protocol alternatively over an Ethernet and an FDDI network. To calculate the influence of BER marshalling on overall throughput, two measurements were performaed, one with marshalling included and one transmitting a byte string without marshalling.

TCP throughput for sending the byte string was about 8.7 Mbit/s over an Ethernet and about 16.7 Mbit/s over an FDDI. Thus, bottlenecks in local protocol processing and hardware left a only twofold throughput increase from the tenfold increase in networking throughput (from 10 Mbit/s Ethernet to a 100 Mbit/s FDDI). Adding BER marshalling further decreased the throughput to about 5.8 Mbit/s over Ethernet, and 7.2 Mbit/s over FDDI. Thus, only a factor of 1.25 is left of the tenfold throughput increase of FDDI when inserting marshalling.[2]

The difference between the result obtained in this experiment and results reported in [Lin93] can be explained by the difference in the "complexity" of the types that were used in the two experiments: [Lin93] used values of simple scalar types (a single octet string and a single integer) to show that marshalling throughput for these types achieves TCP/IP throughput. In contrast, [Huit92] used a complex value of a composite tree type. The nodes of the tree value contained integer and string values.

In summary, the relative overhead incurred by presentation conversion in a particular application depends on the complexity of the messages exchanged. In many practical distributed applications that use ASN.1 to specify the format of their messages such as the X.400 e-mail service, the X.500 directory service or the WAIS information server, these messages consist of rather complex composite data types. For these applications, marshalling overhead remains an important target of performance optimization.

Previous efforts to improve performance of marshalling routines have dealt with optimizing general characteristics of presentation conversion such as the cost of the encoding rules used or the memory management scheme (e.g. [Part88], [Huit89], [Huit92], [Beve92] and [Samp93a]). These optimisations were pursued largely independently of the requirements of a particular application. In contrast, the key idea in our work is to optimize the performance of marshalling routines by exploiting the application-specific knowledge contained in the type specifications of an application.

This paper extends our previous work in this area. We started out by designing an ASN.1-based "profiling method" that allows an application programmer to explicitly describe which parts of an ASN.1 type specification the application will not use, using ASN.1 subtypes. Using this method, a substantial reduction in the code size of an implementation of the upper layers

---

2. Moreover, a surprising result was found when comparing the performance of the BER network data format with the network data format used in Sun's RPC protocol (XDR). It has often been assumed in the past that XDR marshalling stubs are more efficient than their BER counterparts because XDR uses a much simpler data format (e.g. [Part88]). However, in this experiment the BER marshalling stubs generated by Mavros turned out to be faster than the XDR marshalling stubs generated by rpcgen, which is the XDR stub compiler provided by Sun (XDR Ethernet throughput was about 3.7 Mbit/s, FDDI throughput about 4 Mbit/s). Closer examination reveals that the code generation strategy used by the stub compiler has a much higher impact on marshalling speed than the complexity of the network data format in this case. Mavros aims at generating fast marshalling code. Rpcgen, in contrast, seems to favor a small object code size of marshaling stubs to a fast execution time. The low performance of XDR stubs generated by rpcgen when compared to those generated by a good BER stub generator has also been confirmed by others in the meantime ([Samp93a], [OMal94]). In addition, [OMal94] describes an approach that allows the generatin of much faster XDR stubs than those generated by rpcgen.

of the OSI protocol stack was demonstrated (up to 50% reduction in the code size taken up by marshalling routines for the U.S. Government OSI profiles (GOSIP) [Hosc93a]). We then extended this work by using control flow analysis of an ASN.1 interface specification to automatically predict the frequency with which the different types would occur at run-time [Hosc93b]. We suggested the application of the resulting frequency prediction to the problem of deciding between interpreted and compiled conversion routines, using a simple frequency-based ranking of the types. The present paper discusses how this idea can be incorporated into a stub generator. We develop estimates for the code size cost and time savings resulting from compiled marshalling routines. These esimates are based on a detailed analysis and explanation of the trade-offs involved between interpreted and compiled marshalling routines. Moreover, we give a rigorous definition of the optimization problem that must be solved by the stub generator when deciding which types to compile and which types to interpret, and give preliminary results on algorithms that can be used to solve this problem.

3. The Case for Hybrid Conversion Routines

3.1 Interpreted versus Compiled Marshalling Routines

3.1.1 Example Type Declaration

To demonstrate the differences and trade-offs between interpeted and compiled conversion routines, we use the following simple C type declaration as an example throughout this section:

```
typedef struct Foo {int32 i; String string;} Foo;
/* where String is declared as: typedef struct {int32 l; char * v} String; */
```

The local representation of values of type Foo in main memory of the end-system is determined by several factors: the CPU used determines the byte order of the 32 bit integer int32, the C compiler determines the layout and alignment of the struct components in main memory and the type declaration determines that values of this types are generally stored in non-contiguous memory locations. This is because the string value is accessed by a pointer.

A marshalling routine converts values from this local representation into a specific network data format. In our example, the local value representation will be different from its representation in the network data format due to the pointer value for all network data formats in pratical use today (BER, NDR, XDR). Moreover, the network data format may use a different byte order for the integer field and/or different alignement rules for the structure components.

For demonstrational purposes, we use a simplified network data format derived from the BER[3]. In this network data format, each scalar type is represented as a length-value pair. For the example type declaration this means that each component is represented as a length field containing the byte-length of the value followed by the actual value.

3.1.2 Interpreted Marshalling Routines

With interpreted marshalling routines each type declaration of an interface specification translates into a set of marshalling commands. These commands are read on run-time by an

3. This simplification of the BER is made in order to keep the amount of code presented in the following examples within reasonable limits. In addition to the network data format used in the examples, BER requires a tag field to identify each message field. Moreover, an extra tag field and length field is required for the structure as a whole. Extending the example code to include these features is relatively straightforward. Note that the network data format used in the example code still reflects the BER conventions which allow the encoding of lengths of arbitrary values and require integer values to be encoded in a minimal number of bytes.

interpreter routine that does the required format conversion.

Our example type declaration can be translated into the following marshalling commands:

```
cmd Foo_fields[] = {
      { INTEGER, offsetof(Foo, i),      0, (cmdptr)0 },
      { STRING,  offsetof(Foo, string), 0, (cmdptr)0 }
     };
cmd Foo_msg = { STRUCT, 0, 2, &(Foo_fields[0]) };
```

The first two commands in this example contain the information required for marshalling individual components of a composite type. The first field contains the type of the component, and the second field the offset of a particular component from the beginning of the composite type. The third command contains information required for marshalling a composite type. The third field contains the number of components in the composite type and the forth field the address of the marshalling commands for these components.

Given these commands, the following interpreter routine converts values from the local representation into the network data format:

```
char * inter_cod(dst, src, c)
char * dst; char * src; cmdptr c;
{
     switch (c->type) {
     case STRUCT:
     {int i;
      cmdptr cc;
      for (i=0; i<c->field_count; i++) {
           cc = c->field + i;
           src += cc->offset;
           dst = inter_cod(dst, src, cc);
      }
      break;
     }
     case INTEGER:
     {dst = intcod(dst, *(int32 *)src);
      break;
     }
    case STRING:
     {String * x = (String *)src;
      dst = lencod(dst, x->l);
      memcpy((char *)dst, (String *)x->v, (int)x->l);
      dst += x->l;
      break;
     }
     /* ... */
     /* branches for other scalar types and composite types */
    }
    return(dst);
}
```

The routine is called with three arguments: (1) dst, a pointer to the memory location where the encoded value should be stored, (2) src, a pointer to the value in local format and (3) c, a pointer to the command that describes the action required for marshalling the value.

Each marshalling command has its own branch in the switch statement. The components of structure types are marshalled by recursive calls to the interpreter routine. This allows the handling of nested type declarations, and in particular of recursive types. An integer value is marshalled by calling a routine that encodes both the length of an integer and its value in the

minimal number of octets. A string value is marshalled by first calling a routine that encodes the length of the string, and then copying the strings from the source string into the encoded value.

The code shown is of course only one of many possible implementations of interpreted marshalling routines. The code given above reflects the approach used in an experimental extension to the Mavros stub generator. Other stub generators that produce interpreted marshalling routines are ISODE's pepsy, snacc 1.1 and the stub generator of OSF-DCE.

3.1.3 Compiled Marshalling Routines

With compiled marshalling routines each type in the interface specification is translated into a type-specific marshalling routine. In order to marshall a value of a particular type, the marshalling routine corresponding to this type is called.

Our example type specification translates into the following compiled marshalling routine that converts values from the local reprentation into to the network data format:

```
char * Foo_cod(dst, foo)
char * dst; Foo * foo;
{
    dst = intcod(dst, foo->i);
    dst = lencod(dst, foo->string.l);
    memcpy((char *)dst, foo->string.v, foo->string.l);
    dst += foo->string.l;
    return(dst);
}
```

The compiled marshalling routine is called with two arguments: (1) dst, a pointer to the memory location where the encoded value should be stored and (2) src, a pointer to the value in local format.

In principle, the compiled marshalling routine can be understood as sequence of instructions that result from evaluating the interpreter commands for a specific type at compile time. This eliminates the interpretation overhead of decoding the command and calculating the field offsets at run-time, and generally leads to a faster execution time. However, for many interface specifications of practical interest, compiled marshalling routines will have a higher code size than the equivalent interpreted marshalling routines. A look at the example code explains why: for interpreted marshalling routines, the two procedure calls required to marshall a string value are contained only once in the object code. For compiled marshalling routines, these procedure calls have to be duplicated possibly many times in the marshalling routines for types that contain string fields.

Stub generators that produce compiled marshalling routines are the current version of Mavros, ISODE's posy/pepy, snacc 1.1 and Sun's rpcgen.

3.1.4 Optimisation of Compiled Marshalling Routines

Compiled marshalling routines for each individual composite type may offer additional optimisation oppurtunities. This is because we can exploit specific knowledge about the values transmitted in the fields of this type that we do not have in the general case. Knowledge about the length of the network data format is particularily useful.

One example is an optimisation technique that we call *partial inlining*. It is a generalisation of the header prediction technique for marshalling routines presented in [Huit92]. The idea is to duplicate the most frequently used case of a primitive marshalling routine such as lencod or

intcod in the type-specific marshalling routine. Partial inlining is useful for arriving at efficient implementations of marshalling routines without restricting the generality of the network data format by introducing fixed length encodings. In particular, it can be used to speed up the marshalling and unmarshalling of the tag, length and integer value fields in the BER.

For instance, when we know that the length of the string field value in the example is frequently below 128 Byte, this can be reflected in the marshalling code for this field in the following way:

```
if (foo->string.l < 128)
     *dst++ = (char)foo->string.l;
else dst = lencod(dst, foo->string.l);
```

This code avoids the procedure call overhead for the lencod routine in the most frequent case. However, execution time for strings longer than 128 will be slightly higher, since the length is tested twice for the value 128, once in the compiled routine and once in the lencod routine. Furthermore, the additional instructions increase the code size of the marshalling routine.

The increase in code size would be even higher, however, for a second alternative to avoid the procedure call overhead, which is to inline the whole body of the lencod routine. This routine contains code to handle all possible length values. Inlining it for all string fields can lead to a substantial increase in marshalling object code size for interface specifications with many string fields.

A further optimisation is possible when we know that some or all of the message fields will have a fixed length. In this case, we can replace the procedure calles by byte copy instructions.

For instance, assume that we know that the integer value in our example can always be encoded in one byte, and that the string value is always four bytes long. Then, the body of the compiled marshalling routine can be replaced by the following more efficient code sequence:

```
     *dst++ = (char)1;
     *dst++ = (char)foo->i;
     *dst++ = (char)4;
     *dst++ = *(foo->string.v);
     *dst++ = *(foo->string.v+1);
     *dst++ = *(foo->string.v+2);
     *dst++ = *(foo->string.v+3);
```

[OMal94] recently showed that using word-oriented memory accesses in marshalling routines can lead to a substantial speedup for “header marshalling”, a special case of the general marshalling problem that is restricted to marshalling fields found in the TCP/IP protocol suite. Since in this case the message fields all have a fixed byte length, they can be easily mapped onto word-oriented memory access instructions. Word-oriented memory accesses use the CPU-memory bandwidth more efficiently, leading to an improvement in execution time. The BER network data format is explicitly excluded from this optimisation due the use of variable length fields in [OMal94].

However, under the assumption that in addition to knowing the sizes of the individual fields we also know that the network data representation always starts at a word boundary, we can use the word-oriented memory access instructions even for the BER. This may allow for using word-based marshalling routines for protocols such as the network management protocol SNMP, where the use of BER is hardwired into the protocol standard. We are currently investigating the use of profile information to allow the use of word-based access for a non-aligned network data format [Hosc94].

Using word based access in our example results in the following marshalling code:

```
long *word = (long *) dst;

*word++ =   1<<24 |
            (foo->i)<<16 |
            4<<8 |
            *(foo->string.v);
*word++ =   *(foo->string.v+1)<<24 |
            *(foo->string.v+2)<<16 |
            *(foo->string.v+3)<<8 |
            0;
return(dst+7);
```

3.2 The Practical Importance of Code Size Optimization

With the advent of high speed gigabit networks, the "throughput preservation problem" [Part93] due to bottlenecks in end-system hardware occurs. Thus, the advantage of having fast marshalling routines is intuitively clear. However, it might be less clear, in particular to users of today's high performance workstations, why in today's system environments where "memory is cheap" small code size is still a design goal of interest. There are two reasons for this: first, a large class of system environments in use today do have memory size restrictions, e.g. personal computers or mobile systems. Second, and more speculatively, the growing gap between the access time of a system's main memory and CPU cache memory might make a small code size of conversion routines interesting even for today's high performance workstations.

For systems with restricted memory sizes such as personnel computers or embedded systems, compiled marshalling routines for some applications can reach code sizes that are unacceptable. For instance, for all applications we currently use on our Macintosh systems, we found that code size of application programs is substantially below 1 MB. A recent example "from the trenches" of protocol design and implementation helps to put things into perspective: to access the X.500 directory service from a personal computer, a new special-purpose "lightweight directory access" protocol was designed, specified, verified and implemented that completely replaces the OSI upper layer protocols. One of the reasons for this redevelopment effort was that the size overhead of the standard OSI protocol was considered to be too high for a personal computer environment. Marshalling routines are a major contributor to this code size.

Another consideration when tuning the code size of software in general and communication software in particular is making optimal use of the memory hierarchy in a machine. In particular, the gap in access time between the system's main memory and cache memory on the modern RISC CPUs has achieved considerable attention recently in research on implementation of high performance communication systems (e.g. [Clar90], [Drus93]). [Drus93] argues that not CPU performance, but CPU/memory bandwidth will turn out to be the factor inhibiting end systems from making full use of gigabit-network technology in the years to come. The same authors also provides experimental evidence that shows that data caching is not very effective for communication software.

From measurements on other software systems it is known that the instruction stream of a CPU usually exhibits a higher degree of locality than the data stream. Consequently, instruction caching is generally more effective than data caching [Henn90]. In the particular case of "user data" to be sent over the network, low data locality can be explained by the fact the bytes of the user data are usually read only a couple of times before they are sent out onto the "wire". In

contrast, the instructions of a conversion routine for a certain type can be executed very often when values of this type occur frequently in a message - consider the example of a type that is an element of an array. Consequently, in addition to minimizing the number of times “user data” traverses the CPU/memory bus as requested in [Drus93], an equally promising approach appears to be the minimization of the number of times the instructions of the network code traverse the CPU/memory bus.

```
test DEFINITIONS::=
BEGIN

PersonnelRecord::= [APPLICATION 0]
IMPLICIT SET {
      Name,
      title [0] IMPLICIT IA5String,
      EmployeeNumber,
      dateOfHire [1] IMPLICIT Date,
      nameOfSpouse [2] IMPLICIT Name,
      children
      [3] IMPLICIT SEQUENCE OF
      ChildInformation DEFAULT {}
}

ChildInformation::= SET {
      Name,
      dateOfBirth [0] IMPLICIT Date
}

Name::= [APPLICATION 1] IMPLICIT SEQUENCE {
      givenName IA5String,
      initial IA5String,
      familyName IA5String
}

EmployeeNumber::= [APPLICATION 2]
   IMPLICIT INTEGER

Date::= [APPLICATION 3]
      IMPLICIT IA5String
```

```
smithperson PersonnelRecord::= {
      {  givenName “Jean”,
         initial “E”,
         familyName “Smith”
      },
      title “The Big Cheese”,
      99999,
      dateOfHire “19820104”,
      nameOfSpouse {
         givenName “John”,
         initial “L”,
         familyName “Smith”
      },
      children {
      {
         { givenName “James”,
            initial “R”,
            familyName “Smith”},
      dateOfBirth “19570210”
      },
      {
         { givenName “Lisa”,
            initial “M”,
            familyName “Smith”
         },
      dateOfBirth “19590621”}
      }
      }
}

END
```

FIGURE 1. Example: Personnel Record Type and Value (in ASN.1)

For the problem of inlining of procedures in an optimizing compiler, [McFa91] points out that the main criterion for deciding whether a procedure should be inlined is the increase in the cache miss ratio caused by the increased code size after inlining. A similar argument can be made when deciding about whether to generate an interpreted or a compiled marshalling routine for a given type. Interpreted conversion routines, by reducing the code size, can allow keeping more of the conversion instructions in the instruction cache. The savings in memory access time achieved in this way might well be worth the extra cost of the additional instructions executed by an interpreted conversion routine.

3.3 An Experiment with Hybrid Marshalling Routines

To better demonstrate and explain the benefits of hybrid conversion routines we discuss the performance of this new technique on a simple, but nevertheless representative example ASN.1 type and value (see Figure 1).

The example type used is a personnel record (or “struct”) type that contains information about

an employee typically held by a company's administration such as the name, the employee-number, the name of the employee's spouse and eventually information on the employee's children. The personnel record type directly or indirectly references three other ASN.1 type specifications: a "struct" type that represents a person's name, an array type that represents information on the employee's children, and a struct type that contains the particular information required for each child. The personnel record type specification is representative for ASN.1 specifications for real applications in that it contains several composite types of different kinds, and in that these composite types reference each other. Performance measurements were conducted on the ASN.1 value shown next to the type specification in Fig. 1. Fig. 2 shows the time we measured for 100 conversions of a local host representation of this value into the BER (Basic Encoding Rules) representation in four different experiments[4]. In the first experiment we used interpreted conversion for all types. Then, we step by step replaced the call to the interpreter for a specific type by a call to the corresponding compiled conversion routine, and measured the resulting speed-up in conversion time, until we arrived at an implementation where all types are converted by compiled conversion routines.

Overall, we found in this example compiled routines to be about 52% faster than interpreted routines. However, speed does not increase linearly with the number of compiled types: compiling the "name" type, for example, results in a much higher relative speed-up (57.5% of overall speedup) than compiling the "personnel record" type (only 18.5% of overall speedup). This result does not come as a surprise, when we look at the frequency distribution of the types in the ASN.1 value we used for our measurements: we count four occurrences of "name" type values in the benchmark, and only one occurrence of the "personnel record" type.

For a similar ASN.1 type[5] and value, but a different, slower machine environment and ASN.1 compiler (snacc), [Samp93a, Samp93b] found that interpreted conversion was about 73% slower than compiled conversion (124 s : 473 s in absolute time for encoding the value one million times). Thus, we have sufficient reason to believe that the difference in conversion time between interpreted and compiled conversion routines we found in our measurements are not due to the implementation decisions we made in Mavros, but are an indication that at the current state of the art a substantial difference in performance between these two implementation techniques exists

4. Automatic Generation of Hybrid Marshalling Routines

4.1 Requirements

The Personnel record example has served to clarify two points. First, today there is a substantial performance difference between interpreted and compiled conversion routines for ASN.1-BER due to the additional instructions that are executed with interpreted conversion. We have also seen that a substantial increase in speed can be achieved by compiling only one type of the set of types given, thus investing only a small amount of the space overhead required by compiled conversion routines. Thus, our example demonstrates that hybrid.

---

4. All experiments were conducted on a Sparc SS 10 using the gcc C compiler version 2.4.5 with optimization level O2. Time was measured using the "gettimeofday" routine, and clock resolution was 1 microsecond. We used the "indefinite length" encoding option of the BER for all composite types, thus overall length of the produced encoding was 148 bytes. The results for the compiled routines were obtained with the ASN.1 compiler Mavros, version 12.4. The results for the interpreted conversion routines were obtained using the same configuration, and an experimental extension to Mavros that generates interpreter tables.

5. A slight modification of the Personnel Record type specification given in [Samp93a] was necessary in our experiment: due to the prototype status of our interpreter implementation, we had to convert all explicit tags in the specification into implicit tags.

Set of compiled Types	Time 100 encodings in us	Speedup achieved by additional compilation in us, in %
all types interpreted	5,440	n.a.
{Name}	3,825	1615 us, 57.5%
{Name, Personnel Record-children and ChildInformation}	2,882	943 us, 33%
{Name, Personnel Record-children and ChildInformation, Personnel Record} (all types compiled)	2,361	521 us, 18.5%

FIGURE 2. Experiments with hybrid conversion routines for the Personnel Record benchmark

conversion is an attractive technique to combine the benefits of interpreted and compiled conversion routines in a way that results in conversion routines that are fast and small at the same time.

In order to integrate the generation of hybrid conversion routines into a practical stub compiler, two basic approaches are viable: the first approach is to let the application programmer specify explicitly which types of an interface specification require the generation of a compiled marshalling routine, and for which types interpreted routines are sufficient. This can be easily achieved by adding a directive to the stub generator in the interface specification before generating the marshalling code. Specification by the application programmer has the advantage of providing maximum control and the potential for optimal performance, since the application programmer knows (or can find out) which types in the type specification are the "hot spots" in the conversion routines. However, the disadvantage of explicit specification is that it puts an additional load in terms of specification overhead on the programer who is already preoccupied with developing an application. Thus, with explicit specification, our hybrid conversion mechanism risks sharing the fate of many proposals that introduce new optimization parameters: it can be implemented, but will seldom be used because the complexity introduced by the mechanism exceeds its benefit as perceived by the application programmer.

Thus putting hybrid conversion routines into practical use calls for an automatic method where little if any decision-making input from the user is needed. To decide which types in a set of types should be compiled, we need information on the potential cost and benefit of compiling each type. Cost and benefit of compiling a type depend on two factors: the size overhead incurred by a compiled marshalling routine and the potential time savings achievable by such a routine. The potential time savings are the product of the time required to convert an individual value of the type with the number of times a value of this type occurs on run-time.

Thus, to automate the generation of hybrid conversion routines, we need to solve four problems:

- estimate the cost in terms of increased code size of generating a compiled marshalling routine for a type.
- estimate the benefit in terms of decreased processing time achieved by generating a compiled marshalling routine for a type.
- estimate the frequency with which a type will occur on run-time.

- design an optimization algorithm that takes as input the estimates above, the size or speed constraint that has to be satisfied and the relevant set of types. From this information, the algorithm should compute a “good” strategy for generating hybrid marshalling routines.

In the following sections, we will discuss possible solutions to each of these problems in turn.

Type	Code Size Cost	Time Savings
Record/Struct	1+n	1+n
Union	1+n	1+1
Array	2	1+f

FIGURE 3. Formulas for Approximate Estimation of Cost and Savings with Compilation (n denotes the number of components in a composite type)

4.2 Estimation of Code Size Cost of Compilation

The exact code space taken up by the compiled conversion routine for a specific composite type may depend on many factors such as the encoding rules used, the code template used by the stub generator for the composite type, the exact composition of the type of more primitive elements and the extent to which the stub generator uses inlining of procedures. Detailed measurements of these factors are possible and probably required before implementing a stub generator that uses hybrid conversion routines. For demonstrational purposes, we use a more general, but less exact estimation model.

By looking at the code examples given in the previous section, we see that the number of additional instructions that are added when compiling a composite type is proportional to the number of components of this type. Moreover, additional instructions are required in BER for dealing with the composite type itself. We simplify this by assuming that each component of a composite type consumes one “unit of code space”, and an additional unit is taken up by the composite type itself. Column 1 in Fig. 3 shows the resulting approximation formulas for the three basic classes of type constructors contained in the ASN.1 type declaration language-struct types such as “Set” or “Sequence”, union types corresponding to “Choice” types and array types corresponding to “Set Of” or “Sequence Of” types.

4.3 Estimation of Time Savings of Compilation

The exact time it takes to convert values of a specific type depends on factors such as the CPU used, the language compiler used and the concrete value that is converted. However, as in the case of the code size increase estimates, we are only interested in the relative decrease in processing time caused by using a compiled conversion routine for a specific type.

The code examples given show that the additional overhead of interpretation for a composite type grows roughly linearly with the number of components in that type, since the interpreter routine is called once for each component of a component type, plus once for the composite type itself. This means that the time saved in a compiled marshalling routine is proportional to the number of calls to the interpreter routine that can be saved (see second column in Fig. 3). For struct types, the time savings is proportional to one call of the interpreter routine for the composite type itself, and one interpreter call for each of the components. For union types, only one of the components will actually be converted, so that only two “units of time” will be saved by generating a compiled marshalling routine. For array types, the conversion time saved depends on the number of components in the array. Often, this number cannot be determined by static analysis of an ASN.1 interface specification, which contains many array

types with variable lengths. This leads to the more general question of how to estimate the dynamic frequency of occurrence for the types.

4.4 Estimation of Frequency of Type Occurrence at Run-Time

Given the set of types that are used by a distributed application, values of some types in this set tend to occur much more frequently at run-time than others. This is caused by factors such as the use of a type specification as an array element or other multiple uses of a single type in a message, use of optional types and finally different frequencies of the application's use of the different messages.

There are two alternative approaches to gathering information on the frequency of type use: re-looping a trace of the application's run-time behavior into the stub generator and static analysis of the interface description.

To re-loop run-time traces, the application is built and executed and a trace of its behavior, in particular its use of the set of interface types, is produced. This trace is then fed back into the stub generator together with the original set of types, and the stub generator uses the frequency information contained in the trace to produce optimized hybrid conversion routines. This technique is used in several production use general purpose compilers, e.g. in the C compiler delivered by Digital Equipment with the Ultrix operating system. This approach is further pursued in [Hosc94].

Static analysis of the type specification by the stub generator requires less user involvement in the optimization process. In related work, we have already shown that static analysis of a complex set of type specifications can result in very good frequency prediction for a real-world specification with many multiple type references (X.400 e-mail [Hosc93]): eight of the ten most frequently used types in a trace of X.400 messages could be predicted by a very simple prediction algorithm. The main idea of the algorithm was to count the number of times a certain type is referenced by other types. A high number of references for a type indicates a high number of run-time occurrences of values of that type.

We are currently investigating whether this algorithm, which does not take into account the application's dynamic use of the types in the set of type specifications, is transferable to other applications than X.400. Should it turn out that its prediction quality is not sufficient, static analysis can be extended, by taking into account additional specifications of the application's protocol automaton as proposed in [Chri93].

4.5 Optimization Algorithm

In mathematical terms, the optimization problem that occurs in the code generation of hybrid marshalling routines can be described as follows: starting from interpreted marshalling routines for an interface specification containing the type set $T = \{t_1, ... t_n\}$, a set of code size estimates for each type $C = \{c_1, ..., c_n\}$ and a set of estimates for time savings achieved by compiling each type $S = \{s_1, ..., s_n\}$, we would like to optimize marshalling by compiling the conversion routines for some subset of the types in T. Assume that we are given a constraint on the maximal allowable code size cost for marshalling routinesC_{max}, e.g. the size of a CPU instruction cache or of a ROM. Then, the problem may be stated formally as:

Given an upper limit on the code size C_{max} that can be used by marshalling routines, select a subset of types so that the code-size stays below the size limit and time saving is maximal.

*Maximize: sum(s_i*x_i)*

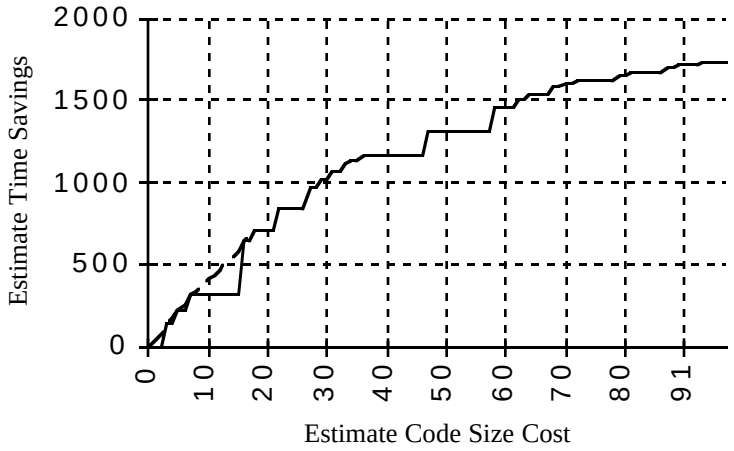
*subject to: sum(c_i*x_i) <= C_{max}*

where the x_i are either 0 or 1, depending on whether the conversion routine for type i is compiled or not.

This problem is a so-called 0-1 Knapsack problem, an optimization problem that is assumed to be computationally intractable, because it is NP-complete. A proof of the NP-completeness of this problem can be found in [Papa82]. However, the Knapsack problem is one of the "easier" NP-complete problems, i.e. good approximate algorithms and heuristics for its solution exist.

To estimate the potential benefits of hybrid conversion routines, we experimented with two algorithms for the knapsack problem. The first is an algorithm of "pseudo-linear" complexity described in [Papa82] that computes an optimal solution of a Knapsack problem (its complexity is O(n**2c), where c is the value of the optimal savings. However, we could not compute solutions for Knapsack capacities that exceeded 18 space units. The second algorithm solves a different Knapsack problem, which assumes that the objects added to the knapsack are divisible. In our case this means that a composite type specification can be partially compiled, i.e. a compiled marshalling routine is generated only for some of its components.

We implemented both algorithms and executed them on the interface specification of the P1 protocol of X.400 e-mail. We used the estimation formulas discussed in the previous section to calculate the individual code size costs C and time savings for each type. The individual time savings were multiplied by the frequency of the occurrence of values of each type in a trace of eighteen X.400 P1 e-mail messages. We calculated optimal solutions for different values of the knapsack capacity, i.e. the maximum allowable code size cost. The results are shown in the following graph:



The cost for generating compiled conversion routines for all types of the X.400 specification is 98 space units. Converting the trace using compiled marshalling routines saves 1733 time units.

From the graph we see that by compiling only 20% (22) of the nodes in the interface specification, we can about 50 % (833) of the maximal speed-up of 1733 time units. Moreover, we see that increasing the code size allowance for compiled marshalling routines leads to diminishing returns for speedup due to locality in the use of the X.400 interface specification,

i.e. certain types of the X.400 interface specification occur more frequently than others at run-time.

We are currently investigating which of the many approximate algorithms and heuristics proposed for the solution of Knapsack problems is best suited as optimization algorithm for the automatic generation of hybrid conversion routines.

5. Future Work

We are currently extending this work in several directions. First, we are conducting measurements to evaluate and refine the cost model developed in this paper. Moreover we analyze whether similar models can be developed for other optimization techniques for marshalling routines.

A further point that needs to be addressed is whether hybrid conversion may help to reduce the miss rate of an instruction cache. While we have argued that hybrid conversion might reduce this miss rate, and thus increase marshalling performance, detailed measurements are required to confirm this intuition. Finally, we have to study how our results for the ASN.1 Basic Encoding Rules change when we change the encoding rules. We expect that other sets of encoding rules will require different estimation formulas than those used for the BER.

References

[Beve92] Bever, Martin and Ulrich Schaeffer. “Coding Rules for High Speed Networks.” Upper Layer Protocols, Architectures and Applications, Proceedings IFIP WG 6.5 Conference, Vancouver: 1992, 113-126.

[Chri93] Chrisment, Isabelle and Christian Huitema. “ROSTAR: a Remote Operations System Tailored to Application Requirements”. ULPAA ‘94.

[Clar90] Clark, David and David Tennenhouse. “Architectural Considerations for a New Generation of Protocols”. SIGCOMM ‘90, 200-208, 1990.

[Drus93] Druschel, Peter, Mark Abbott, Michael Pagels and Larry Peterson. “Network Subsystem Design.” IEEE Network July 1993, p. 8-17.

[Henn90] Hennessy, John and David Patterson. Computer Architecture: A Quantitative Approach. Palo Alto: Morgan Kaufmann Publishers, 1990.

[Hosc93a] Hoschka, Philipp. “Towards Tailoring Protocols to Application Specific Requirements”. IEEE INFOCOM ‘93 Proceedings, Volume 2. Los Alamitos: IEEE Computer Society Press, 1993, 647-653.

[Hosc93b] Hoschka, Philipp. “Using Control Flow Analysis for Space and Time Efficient Stub Generation.” Submitted for publication.

[Hosc93c] Hoschka, Philipp and Christian Huitema. “Control Flow Graph Analysis for Automatic Fast Path Implementation.” Proceedings “Second IEEE Workshop on the Architecture and Implementation fo High Performance Communication Subsystems”, Williamsburg, Virginia, 1993.

[Hosc94] Hoschka, Philipp. “Profile Guided Code Optimisation of Marshalling Routines.” Submitted for publication.

[Huit89] Huitema, Christian and Assem Doghri. “Defining Faster Transfer Syntaxes for the OSI Presentation Protocol.” ACM Computer Communication Review, 19, 5 (October 89), 44-55.

[Huit91] Huitema, Christian. “MAVROS: Highlights on an ASN.1 Compiler”. Internal working paper. Sophia-Antipolis: INRIA, Projet RODEO, 1991.

[Huit92] Huitema, Christian and Ghislain Chave. “Measuring the Performances of an ASN.1 Compiler.” Upper Layer Protocols, Architectures and Applications, Proceedings IFIP WG 6.5 Conference, Vancouver: 1992, 99-112.

[ISO88] International Standards Organization. Specification of Abstract Syntax Notation One: ISO 8224, 1988.

[Lin93] Lin, Huai-An. “Estimation of the Optimal Performance of ASN.1/BER Transfer Syntax”. ACM Computer Communication Review. July 93, 45 - 58.

[McFa91] McFarling, Scott. “Procedure Merging with Instruction Caches.” Proceedings of the ACM SIGPLAN

‘91 Conference on Programming Language Design and Implementation. Toronto: 1991, 71-79.

[OMal94] O’Malley, Sean, Todd Prebsting and Allen Montz. “USC: A Universal Stub Compiler.” Technical Report TR 94-10, Departement of Computer Science, University of Arizona, 1994. To appear in SIGCOMM ‘94.

[Onio89] Onions, Julian and Marshall Rose. “The Applications Cookbook.” Stefferud, Einar, Ed. Proceedings of the Fourth International Symposium on Computer Message Systems. Amsterdam: North-Holland, 1989, 217-231.

[Part88] Partridge, Craig and Marshall Rose. “A Comparison of External Data Formats.” In: 4th International Symposium on Computer Message Systems. September 88, 233-245.

[Part93] Partridge, Craig. Gigabit Networking. Addison-Wesley, 1993.

[Papa82] Papadimitiou, Christos and Kenneth Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Englewood Cliffs: Prentice Hall, 1982.

[Samp93a] Sample, Micheal and Gerald Neufeld. “Implementing Efficient Encoders and Decoders for Network Data Representations.” IEEE INFOCOM ‘93 Proceedings, Volume 3. Los Alamitos: IEEE Computer Society Press, 1993, 1144-1153.

[Samp93b] Sample, Micheal. Private communication.

[Samp93c] Sample, Micheal. Snacc 1.1: A High Performance ASN.1 to C/C++ Compiler. Vancouver: University of British Columbia, 1993.

[Shir92] Shirley, John. Guide to Writing DCE Applications. Sebastopol: O’Reilly & Associates, 1992.