USC: A Universal Stub Compiler*

Sean O'Malley, Todd Proebsting, and Allen Brady Montz
University of Arizona†

Abstract

USC is a new stub compiler that generates stubs that perform many data conversion operations. USC is flexible and can be used in situations where previously only manual code generation was possible. USC generated code is up to 20 times faster than code generated by traditional argument marshaling schemes such as ASN.1 and Sun XDR. This paper presents the design of USC and a comprehensive set of experiments that compares USC performance with the best manually generated code and traditional stub compilers.

1 Introduction

Operating systems in general and network sub-systems in particular are riddled with code that converts data from one format to another. This process is commonly called “marshaling.” We have identified three common types of marshaling code: IPC argument marshaling, protocol header marshaling, and device marshaling. IPC argument marshaling involves converting IPC arguments from the sender's local format to the receiver's local format. Protocol header marshaling involves converting protocol header data from the sending machine's format to the network representation specified in the protocol specification. Device marshaling occurs when hosts must communicate with mem-

ory mapped devices that have different byte orders or alignments.

Marshaling code converts data encoded in one representation to another. This can involve swapping bytes to correct for bad byte order, moving bytes to adjust alignments, changing the size of certain values, changing the representations of complex types such as floats, and providing relocated values for pointers. The correctness, performance, and portability of the marshaling code is critical. Argument marshaling is becoming recognized as one of the last remaining performance bottlenecks in network software[3].

There are two common solutions to the marshaling problem: manual code generation, and automatic code generation with a stub compiler. Most Internet protocols use manual methods for header marshaling and to our knowledge all device marshaling is done manually. Unfortunately, the manual generation of marshaling code is time consuming, error prone, and rarely results in portable code. Most argument marshaling is now done using stub compilers such as Sun's rpcgen. In addition, certain protocols use a stub compiler to marshal their headers. For example, Sun RPC uses XDR[13]. Stub compilers eliminate most portability problems. Unfortunately, the performance of the stubs generated by many existing stub compilers leaves much to be desired. Furthermore, all existing stub compilers are inflexible and can only be used to marshal data to and from a stub compiler defined intermediate form. It is not uncommon for a single host to use manually created stubs for devices and headers and to have to support several different stub compilers.

This paper introduces a general solution to the marshaling problem. We have designed and implemented a new stub compiler, called USC (Universal Stub Compiler), that automatically generates stubs to convert a C data structure with one user-defined format to a C structure with another user-defined format. USC is designed to support the automatic generation of device and header marshaling code as well as provide the

*This work was supported by ARPA under the grant DABT63-91-C-0030.

†Authors' address: Department of Computer Sciences, University of Arizona, Tucson, AZ 85721. Email: {sean,todd,bradym}@cs.arizona.edu.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGCOMM 94 -8/94 London England UK
© 1994 ACM 0-89791-682-4/94/0008..\$3.50

low-level code generator for a more extensive argument marshaling stub compiler. In summary, USC:

- Automatically generates code from a concise specification.
- Generates nearly optimal code. USC stubs are up to 20 times faster than XDR or ASN.1 stubs.
- Is flexible. USC can marshal stubs for headers, devices, and simple IPC interfaces.
- Generates stubs that efficiently peek into the middle of a data structure stored in network format.
- Automatically figures out the alignment and byte order of any C data structure on any C compiler.

The paper is organized as follows: Section 2 introduces the principles underlying the design of USC and Section 3 gives the syntax and semantics of the USC stub definition language. Section 4 then presents a comprehensive evaluation of USC performance. Finally, Section 5 discusses several issues raised by this work, and Section 6 offers some conclusions. A complete description of the USC language is given in Appendix A.

2 USC Design Principles

A marshaling stub compiler generates code fragments that convert data of a given type from one representation to another. A marshaling stub compiler takes as its input a program written in a stub definition language and outputs code in a programming language. To design a new stub compiler one needs to define a type system, the data representations supported, the stub definition language, and the output language.

Since most OS and networking code is implemented in C (or C++), the USC type system is based upon C's. Furthermore, the USC type system is limited to integer types, enumerated types, arrays and structures of these types, and bit-fields. We believe that these types are sufficient to generate most header and device marshaling stubs. Pointers are excluded from the USC type system because they rarely appear in headers (Table 1 gives the distribution of data types found in the protocol headers in our protocol library). require no special treatment in device marshaling (as the device and host share memory), and can easily be handled by a higher level stub compiler with no loss of efficiency. Floating point types are excluded for much the same reasons as pointers. Unions are a more complex problem and are only partially supported in USC. USC

Type	Frequency
short	34%
char	25%
int/long	23%
struct	7%
enumerated	5%
bit-field	4%
array	1%
pointer	1%

Table 1: Type Frequencies

does not support union assignment, but does support access to individual fields of unions.

After defining the legal types supported by USC the next task is to define the range of data representations allowed for any given type. Traditional stub compilers support only two data formats: the local host's native format and the stub compiler's defined intermediate format. Such traditional stub compilers have only two possible representations for any type of data. USC, however, is intended to support the fully general data representations found in device and header marshaling code. USC solves this problem by providing a type annotation which allows a user to specify the precise layout of any given type in the USC type system.

Note that this approach requires that each data structure used be annotated with its layout. Programming USC would be easier if the programmer could simply tag each type with the representation it uses. For example, the user could simply state that this complex type uses XDR representation. USC does not support this style of programming but this would be very easy to implement at a higher level. One simply needs to write a tool that takes an unannotated USC type and produces the correct USC annotations for some data representation. We have implemented such a tool for the local compiler-host data format and it is easy to imagine a tool that would produce annotations for any representation with a well understood data layout algorithm.

Defining the annotation requires a good understanding of the tradeoffs between the power of the annotation and the complexity and performance of the resulting stub compiler. We designed the USC data layout annotation after determining the set of representations for which we could generate efficient code.

The first decision was to provide no support for dynamic encodings. In a dynamic encoding the value of the variable can affect the data layout. The only data representation format known to us that uses dynamic encodings of base types is ASN.1/BER[8][9]. In

ASN.1/BER the value of an integer determines the number of bytes in the encoded format. In theory, dynamic encodings are more compact and flexible than static encodings for the same data. Unfortunately, for small types such as integers the overhead of the tag and length fields required by ASN.1/BER offsets much of the potential space savings. Furthermore, the added flexibility is of no value unless one is generating stubs for a language which also supports the dynamic encoding of integer types (which C does not).

Dynamic encodings of base types are expensive to process. Even with a highly optimizing stub compiler such as the one described in [4], encoding an ASN.1/BER integer requires at least 11 instructions. This is a significant performance penalty when compared to the two instructions possible with a static encoding in common cases.¹

Some argument marshaling systems assume that the encoded version of a data type may be spread across several distinct memory buffers. This creates a problem similar to the use of dynamic encodings; the actual offsets of various fields is not known until runtime. Therefore, USC assumes that both the source and destination of any stub are in contiguous memory. For the data structures found in header and device marshaling this is a reasonable restriction. General purpose argument marshaling does require some support for marshaling types over multiple packets. This support can be provided above the USC level without affecting the performance on small types.

USC also provides no automatic support for tagged data representations such as those used in MIG. Stubs that read and write MIG formatted data are possible but the tag values must be handled explicitly by the stub.

USC provides no support for machines with odd byte sizes and machines that don't use two's complement arithmetic to represent signed integers. Bit-fields are supported in USC by requiring that each collection of bits in a field be drawn from some underlying integer type.

Within these restrictions USC attempts to be as general as possible about the data formats it allows. The USC annotation (described in Appendix A) appears to be sufficient to define all common machine representations, all Internet headers, and XDR representations of USC types. The notation can be used — albeit somewhat awkwardly — to define MIG data. We are still unsure as to its ability to define memory mapped device interfaces. While it can certainly cover a great majority of such interfaces, we are still discovering novel

¹ The 54 instruction minimum presented in [10] appears to be somewhat pessimistic.

formats which may require extensions to USC.

USC generates C stubs. C was chosen for maximum portability. This can create problems when the target C compiler has a poor optimizer. For example, USC cannot take advantage of any special byte swapping machine instructions if the C compiler does not generate such instructions. We have learned to depend as little as possible on the C optimizer. For example, USC can be configured to limit itself to word load and stores and thus avoid the C compiler's potentially inefficient byte manipulation operations.

USC's stub definition language is basically C extended to support the data layout annotations. Stubs are defined in USC as functions defined in ANSI C with two key differences: since layout annotations are supported one can assign between types with different layouts using the assignment operation, and USC supports only a small subset of C's expression grammar. The primary reason for this decision to use the C syntax is that most users already know how to program in C and hence USC should be easier to use. This approach also provides a natural means to allow USC to produce stubs that read and write specific fields or elements in large composite types. For example, because the USC expression grammar supports the C field selection operation, USC can be used to generate a stub that writes the hop count field in the IP header in a network packet. Efficient device driver and router code often depends upon the ability to modify a piece of data in place.

Most importantly, we decided that performance matters. USC generates nearly optimal C code. Optimized code is vital for three reasons. First, modern Internet protocol implementations are very efficient; some TCP implementations require only tens of instructions to process an incoming packet[2]. A stub compiler which required hundreds of instructions to marshal the TCP header would dominate the cost of the protocol implementation. Second, USC can be used to generate very small stubs to access particular fields in structures. Generating such stubs is only useful if the generated code is very efficient. Finally, if programmers believe that a stub compiler is generating bad code they will simply not use it where performance is required.

3 Example USC Specification: TCP Header

The syntax of a USC program is a subset of the ANSI C syntax extended to allow the user to annotate data type definitions with byte order and alignment information. With this syntax the user declares type defi-

initions and functions which manipulate values of these types. With minor exceptions a USC program stripped of its annotations is a valid C program. Below is the USC program tcp.usc.

```
/* define native DECstation base types */
#pragma long(4,4,4,<0..3>);
#pragma int(4,4,4,<0..3>);
#pragma short(2,2,2,<0..1>);
#pragma char(1,1,1,<0>);

/* tcp header in native DECstation format */
typedef struct tcp_native_hdr {
    u_short    sport(2,2,0,<0..1>),
               dport(2,2,2,<0..1>);
    u_int      x2(1,1,4,<0>):4(4,8,0,<4..7>),
               off(1,1,4,<0>):4(4,8,0,<0..3>);
    u_long     seq(4,4,8,<0..3>),
               ack(4,4,12,<0..3>);
    u_char     flags(1,1,16,<0>);
    u_short    win(2,2,18,<0..1>),
               sum(2,2,20,<0..1>),
               urp(2,2,22,<0

} native_hdr(0,22,4,0);

/* tcp header in network format */
typedef struct tcp_net_hdr {
    u_short    sport(2,2,0,<1..0>),
               dport(2,2,2,<1..0>);
    u_int      x2(1,1,4,<0>):4(4,8,0,<4..7>),
               off(1,1,4,<0>):4(4,8,0,<0..3>);
    u_long     seq(4,4,5,<3..0>),
               ack(4,4,9,<3..0>);
    u_char     flags(1,1,13,<0>);
    u_short    win(2,2,14,<1..0>),
               sum(2,2,16,<1..0>),
               urp(2,2,18,<1..0>);
} net_hdr(0,20,1,0);

/* a stub definition */
void tcphdr(net_hdr *src, native_hdr *dest)
{
    *dest = *src;
}
```

A USC program consists of a series of pragmas followed by type and stub definitions. The pragmas define the native format of all base types for the compiler/machine combination for which USC will be generating code (in this case gcc compiling for a DECstation 5000). The type net_hdr is a variant of the TCP header in big endian byte order and is packed into minimal space without regard to the alignment

of any of its fields. The type native_hdr is the same variant of the TCP header in little endian byte order with the structure padded so that each field is aligned appropriately. The stub tcphdr takes a TCP header in network format and copies it to a TCP header in DECstation 5000 format. Running USC on the file tcp.usc will create two files: tcp.c and tcp.h. tcp.c contains the host and compiler specific C implementation of the stub tcphdr. tcp.h contains a prototype of the function tcphdr or a macro implementation of tcphdr.

The specification above is essentially C, with additional information that precisely describes the data layout of each defined data type. This information is encoded as a tuple:

(tsize, msize, alignment/offset, byte order)

In a global type or pragma declaration, the third field is interpreted as the alignment. To describe a structure or union field, the third field is interpreted as the offset. In all declarations except bit-fields, all numbers are in units of bytes; in bit-fields the units are bits. Tsize is the size of the data type (ie, a 4 byte integer) and msize is the actual amount of memory allocated for this type (a 2 byte short may actually occupy 4 bytes). Below we explain some specific lines of the previous example:

Pragma Declarations:

```
#pragma long (4,4,4,<0..3>);
```

The pragma above defines the layout of a long on the compiler/machine combination for which USC will generate code. USC will take this definition to mean that longs are 4 bytes long, require 4 bytes of storage, and are aligned on 4 byte boundaries. The last field, which could also have been written {0, 1, 2, 3}, means that the least significant byte of a long is at offset 0 from its address, and the most significant byte is at offset 3 from its address. In other words, longs are little endian.

Global Type Declarations:

```
typedef struct tcp_native_hdr {
    ...
} native_hdr (0,22,4,0);
```

A structure's annotation is similar to a pragma declaration, except that byte order field has a value of 0. A 0 Tsize field means that the USC compiler will determine the size of the structure itself. This structure requires 22 bytes of storage, and is 4-byte aligned.

Field Declarations:

```
u_short    sport (2,2,0,<0..1>),...
```

Sport is a field of the struct `native_hdr`. It is a little endian 2 byte short that occupies 2 bytes of space, and is found at offset 0 (the third field's value) from the start of the structure.

Bit-Fields:

```
u_int x2 (1,1,4,<0>) : 4(4,8,0,<4..7>),
    off(1,1,4,<0>) : 4(4,8,0,<0..3>);
```

`x2` and `off` are bit-fields that occupy the same byte in the structure. The first tuple: `(1, 1, 4, <0>)` declares the underlying integer type with respect to which the bit-field is defined. Notice that both bit-fields are declared to have an offset of 4, so they are found in the same byte. After the colon, units are in bits. `x2` is a four-bit bit-field, which has size 4 bits, and occupies 8 bits. The offset field of a bit-field has no meaning, so it is left at zero. The last field describes the bit order. The least significant bit is bit 4 with respect to the underlying integer type declared for `x2`.

In addition to the four data types shown in this example, USC can handle arrays, including variable length arrays. Please refer to Appendix A for a more in-depth treatment of the USC syntax and semantics of data type and stub declarations.

4 Evaluation

4.1 Device Marshaling

As an experiment we have used an early version of USC to write a DEC Alpha Ethernet driver (AMD Lance). Preliminary performance comparisons between this driver and the Mach Lance driver show a 3% performance improvement. This performance improvement seems to be entirely a result of USC's ability to generate small stubs which read and write fields in the middle of large data structures. In contrast the Mach Lance driver copies an entire structure twice to write a single bit-field. While certainly not conclusive this result provides evidence that USC-generated device marshaling stubs can perform competitively with manually generated stubs.

4.2 Header Marshaling

To evaluate the performance of USC we ran a series of performance tests comparing USC generated stubs with stubs generated manually or by other stub generators. These tests were run on 5 different machines: a SPARC 1, a SPARC 10, a DECstation 5000/200, an Intel 486, and an HP 735. The marshaled data

structures consisted of a 4 byte long, an 8 byte UDP header, and an 82 byte large composite header (called big header) constructed by concatenating an Ethernet header, an IP header, a TCP header, and an ARP header. Big header represents what we consider to be a reasonable size for all network headers affixed to a single packet. Each test case marshals a data structure to and from network form twice. This corresponds to a complete round trip of a protocol. All tests were run using various versions of gcc. The tests were constructed to ensure that no usable data was saved in registers between each marshaling. Unless otherwise noted all test data was in the machine's cache.

The first test compares the performance of USC generated stubs to that of the gcc implementation of structure copy. We defined USC stubs which copy the three data structures without byte swapping and compared the performance of these stubs against the performance of a simple C assignment of the three data structures. The results of this test are given in Table 2. Note in Tables 2 through 5 each column is divided into three sub-columns: the first gives the performance of a stub marshaling a long, the second a stub marshaling a UDP header and the third a stub marshaling a big header.

Surprisingly, USC appears to generate better code than gcc in several cases. The reason for this in the UDP header test case is that the USC program defining the UDP header specified 4-byte alignment and the compiler assumed 2-byte alignment. USC safely generates word loads and stores; gcc must generate short loads and stores. This optimization accurately reflects typical network code: in general the network UDP header is four byte aligned in packets and it is easy to force the alignment of the local instance of the UDP header to a four byte boundary. The performance of a USC stub which copies a two byte aligned UDP header is identical that of structure copy. For the big header test case on the two Sparcs, gcc uses `memcpy` to copy the bytes, which is apparently inferior to the straight inline code generated by USC for data structures of this size (82 bytes). From this test case we conclude that the performance of USC is effectively optimal in the degenerate case where no byte swapping is required.

The second test compares the performance of USC generated stubs with that of manually generated portable stubs implemented using the BSD `ntoh` and `hton` functions to swap bytes where needed. For this test, the appropriate stubs were generated for each machine. For big endian machines such as the Sparcs and HP 735 no bytes are swapped by the stubs. For little endian machines such the DECstation and the 486 bytes were swapped for every field of the headers except the IP and Ethernet addresses. The results of this

	Sparc 1			Sparc 10			DEC 5000			486			HP 735		
	long	udp	big	long	udp	big	long	udp	big	long	udp	big	long	udp	big
scopy	1.8	6.6	49	0.3	0.6	7	0.5	2.2	12	0.4	0.8	8.6	0.2	0.9	3.5
usc	1.8	4	31	0.3	0.6	4.5	0.5	0.9	10	0.3	0.7	8.0	0.2	0.4	3.5

Table 2: USC vs. C structure copy (usec)

	Sparc 1			Sparc 10			DEC 5000			486			HP 735		
	long	udp	big	long	udp	big	long	udp	big	long	udp	big	long	udp	big
ntoh	1.8	6.6	92	0.3	0.7	10	2.2	6.2	72	0.7	1.5	15	0.2	1	13
usc	1.8	4	31	0.3	0.6	4.5	1.7	3.5	22	1.0	1.2	13	0.2	0.4	3.5

Table 3: USC vs. ntohs (usec)

test case are given in Table 3.

These results show that the performance of USC generated stubs is generally superior to ntohs stubs on both big endian and little endian machines. The reason for this difference on big endian machines is that USC generates a series of word load and stores while the ntohs stubs load and store each field of the data structure separately. The ntohs stubs use some half-word and byte loads and stores for the data structures tested. When byte swapping is required (on the Dec and Intel) USC stubs are also generally faster than ntohs stubs. The reason for this is again that USC takes advantage of knowing that the headers are four byte aligned. For example, the USC generated code to swap the bytes of the first two fields of a UDP header is given below. Note that the code swaps the bytes a full word at a time:

```

r0 = *(int *)((char *)l_src + (0));

*((int *)((char *)l_dst + (0))) =
  (((r0 >> 8) & 0xff00ff) |
   ((r0 << 8) & 0xff00ff00));

```

Next, the performance of stubs written using the Sun XDR library was tested. Because the XDR encoding format encodes all integer types in four byte quantities the encoded UDP and big header data structures are longer than the native data structure. In addition because XDR does not support bit-fields each set of bit-fields is treated as an integer in the XDR stubs. The results of this test are given in Table 4.

The first row of Table 4 gives the performance of stubs that were generated by rpcgen with the rpcgen

optimizer disabled. These results are clearly orders of magnitude worse than either the USC stubs or the ntohs stubs. The second row of Table 4 gives the performance of stubs generated using rpcgen with the optimizer engaged. The UDP header test shows some significant improvement but the optimized stubs are still an order of magnitude slower than USC generated stubs. The big header test shows only minor improvement. The reason for this is that the XDR library implements only four macros that the rpcgen program uses to optimize the conversion of shorts and longs in XDR defined structures. Unfortunately, the big header contains char's and fixed length opaque byte strings which rpcgen does not optimize. The third row of Table 4 gives the performance of USC generated stubs that encode the data structures into XDR format. These results clearly show that it is the XDR/rpcgen implementation rather than the XDR encoding scheme that causes the poor performance of rpcgen. Closer examination of the XDR library helps explain this problem. The standard XDR library incurs at least one procedure call per base type and another procedure call for each word of the encoded format. The poor performance of XDR stubs has been noted by others[11] [7].

Next the performance of the MAVROS[5] ASN.1 compiler was tested. MAVROS was selected for this experiment because MAVROS generated stubs have been shown to be at least as fast as stubs generated by SNACC and much faster than stubs generated by ISODE. As in XDR, the encoded forms of the ASN.1 headers differ greatly from the standard header definitions. In addition, the big header was simplified by replacing the 6 byte Ethernet address fields with 4 byte integers. The results of this test are given in Table 5.

	Sparc 1			Sparc 10			DEC 5000			486			HP 735		
	long	udp	big	long	udp	big	long	udp	big	long	udp	big	long	udp	big
rpcgen	20	64	679	4	16	148	15	51	462	10	39	372	5.7	19	198
rpcgen-opt	20	25	530	4	15	110	14	21	360	11	14	299	5.8	5.6	151
usc-xdr	1.7	6.5	77	0.5	0.7	9				1.0	1.2	13	0.5	0.7	7

Table 4: XDR Performance (usec)

	Sparc 1			Sparc 10			DEC 5000			486			HP 735		
	long	udp	big	long	udp	big	long	udp	big	long	udp	big	long	udp	big
asn.1/ber	12	46	263	2.7	9	52	11	37	194	10	29	160	2.4	7.7	37
asn.1/opt	5	17	78	1	3.3	12	4.4	16	47	4	11	26	1	3.2	8.5

Table 5: ASN.1 Performance (usec)

The first row of the table gives the performance of MAVROS stubs using the ASN.1/BER syntax. The performance of ASN.1/BER stubs is much worse than that of USC or ntoh stubs but is significantly better than XDR stubs (see Tables 2-4). Again the question is whether or not the poor ASN.1 performance is a result of the ASN.1/BER syntax or is simply a function of MAVROS. Fortunately, MAVROS supports an experimental simplified ASN.1 encoding scheme which uses static encodings for base types. The last row in Table 5 gives the results of using MAVROS to generate stubs using the experimental ASN.1 encoding format. These stubs perform much better than the ASN.1/BER stubs. The poor ASN.1 performance is clearly caused by the dynamic format defined in BER.

Note that in [6] the reported results on this new encoding format were discouraging. There are several possible explanations for the discrepancy between their results and ours. The first explanation is that the test cases used in the paper were large and complex data structures that require the use of dynamically allocated storage. While the paper claims to have eliminated this bias by implementing a special version of malloc, there may have still been significant malloc overhead. For small simple header data structures the lightweight syntax is clearly superior. The second explanation could be that while MAVROS generates good code for the lightweight syntax it does not generate great code. It makes little use of macros and still requires several procedure calls per data structure.

In the previous tests the data structure to be copied was always in the cache. In an actual protocol application the arriving header is rarely in the cache. To

determine the potential effect of cache misses on stub performance we ran modified tests on the HP 735 for stubs encoding the UDP and big header. The HP 735 has a 256k direct mapped cache. By staggering the headers to be copied at 256 kilobyte intervals the tests cause a cache miss on every copy (four per round trip). The performance of this test is given in Table 6. USC maintains a reduced but significant performance advantage over the other stub generation techniques even when cache effects are included in the test. Clearly, the performance of XDR is still unacceptable. However the performance of ASN.1/BER stubs generated by MAVROS is surprisingly close to the performance of the ntoh stubs.

The final question is whether or not the potential performance gain in header marshaling code could affect the measured performance of actual protocols. A rough estimate of the potential performance effects of using a stub generation technique to marshal headers can be obtained by comparing the round trip encoding costs for the big header to the round trip performance of actual protocol implementations. Table 7 gives the ratio of the total time required to marshal a big header four times to the round trip time recorded for three different protocol implementations on the DECstation 5000/200: Ultrix user-to-user UDP/IP (1200 microsecond round trip), Mach kernel-to-kernel UDP/IP (800 microsecond round trip), and the RPC over ATM protocol presented in [14] (170 microsecond round trip).

These results show that for standard protocol implementations the performance advantage of USC over ntoh stubs would probably be undetectable. However USC might produce detectable performance im-

HP 735	ntoh	usc	rpcgen	rpcgen-opt	ans1/ber	ans1/opt
udp hdr	4.7	3.8	22	9	10	6.3
big hdr	37	16	212	165	46	19

Table 6: Cache Effects (usec)

DEC 5000 RIT	usc	ntoh	rpcgen-opt	asn1/ber
1,200 μ s	2%	6%	30%	16%
800 μ s	3%	9%	45%	24%
170 μ s	13%	42%	212%	114%

Table 7: Relative Costs

provements when used to marshal headers of very lightweight protocols. Clearly, the cost of using heavy-weight stub compiler such as rpcgen to generate stubs for 82 bytes of network header could have a noticeable effect on standard protocols even when running on a reasonably fast machine. For very low latency RPC implementations the cost of using such stubs could dominate the rest of the implementation.

4.3 Argument Marshaling

A critical question in the evaluation of USC is how much of its header marshaling performance will translate into improved argument marshaling performance for a higher-level stub compiler. While a definitive answer to this question must wait until we have finished implementing a higher-level stub compiler on top of USC there seems to be strong evidence that a significant performance improvement will result. First, we see no reason why the addition of support for pointer types and multi-part messages should have any detrimental effect upon the performance of simple stubs which do not use pointers and are contained in a single packet because we plan to use USC directly in such cases. Second, we expect simple stubs to dominate the distribution stubs in any real distributed system. We believe that complex pointer based types are rarely sent if only because they are so expensive. It is much more efficient to structure a distributed system such that large complex types don't move and use simple IPC's to update them in place as is recommended in [12]. Finally, we believe that the poor performance of traditional stub compilers when marshaling base types and structures has masked potential improvements in support for marshaling pointer types.

5 Discussion

5.1 USC Implementation

USC generates efficient code by minimizing loads and stores and doing algebraic transformations[1] to optimize representation changing operations (eg., masks and shifts). The optimizer's first priority is to minimize memory accesses. The test data we've seen supports the assertion that memory access is the primary hindrance to efficient stubs. The alignments of source and destination are used to determine the maximum size of loads and stores. For example, on a Sparc, if the source data is known to be aligned on 2 byte boundaries, then data can be read in chunks of 2 bytes. A chunk of data which is read or written is called a bucket. The first pass of the optimizer partitions the bits of the source and destination into buckets.

USC translates the data conversions into a series of bit assignments, which may be one of three types: plain, sign, and zero. A plain bit copy is the standard bit i gets bit j . A sign bit copy signifies that bit i gets bit j , and that bit j is copied to more than one bit in the destination. Sign copy is used to simplify certain sign extension optimizations. A zero bit copy indicates that bit j gets 0. Zero copy is used to differentiate between bits in the destination which must be zero, and bits in the destination which can have garbage left in them.

The intermediate code is then transformed into a series of bucket assignments of the form

```
out_i <- ((in_j << shift_1) & mask_1) |
        ((in_k << shift_2) & mask_2) | ...
```

in which the shifts and masks are constants. These are then optimized using algebraic simplifications. The

most common simplification is the application of the distributive properties of ANDs over ORs:

$$((x \ll S) \wedge M_1) \vee ((x \ll S) \wedge M_2) \Rightarrow (x \ll S) \wedge (M_1 \vee M_2).$$

Then the optimizer applies various peephole optimizations and passes the simplified intermediate code to the code generator, which is responsible for generating the C code using a minimal number of registers.

Unlike most existing stub compilers (and most manually generated stubs), USC will optimize across field boundaries in structures. For instance, if two short fields may be correctly copied as a single word, USC does so. The complexity of the generated stub is proportional to the amount of *data movement* rather than the complexity of the original type definition. Thus, USC efficiently supports bit-fields and 1 and 2 byte integers. In contrast, XDR limits the representation of all integers to four bytes to simplify the generation of stubs and increase efficiency. We believe this approach is misguided. We have found that marshaling a native 2 byte integer to XDR's 4 byte representation is significantly more expensive than marshaling the same native representation into a two byte representation. Using XDR format results in a larger representation and eliminates the possibility of optimizing across field boundaries.

The optimizer and code generator are specifically designed to be as general as possible. The user only specifies the native byte order, register size, and the types of load and store operations to generate code for a new machine.

5.2 USIT: The USC Inference Tool

The correctness of a USC stub is dependent upon the accuracy of the data layout annotations. For headers in network format this is generally not a problem because the precise data layout of the header is included in the standard and once a USC type has been defined for that layout it can be used on all hosts and compilers. Getting the correct layout of the native compiler format of a network header is another matter. It is rarely specified by the compiler documentation and it changes for each host/compiler pair. Annotating such types manually could be as error prone and time consuming as writing byte swapping code by hand.

To eliminate this problem we have written the USC Inference Tool (USIT) to determine the alignment and byte order of native variables. USIT takes a file containing valid C type and variable declarations without any USC annotations and outputs a USC program with those types and variables properly annotated for the local compiler/host pair. USIT generates and runs a C program to infer the annotations.

5.3 Future Work

We plan to extend USC to add support for the C equality operator (==) for all types, including structured types. Protocols often must map some arbitrary key to some local state. This is often done using BCMP, which can have unpredictable results on unpacked structures. The USC annotations provide enough information to generate correct structure comparisons that are statically optimal. We plan to add a pragma that will allow the user to specify the order in which to compare the bytes in a structure.

The extension to USC that would most significantly improve the protocol writing process would be to integrate the USC annotations into the C programming language directly. While USC automates the generation of byte order and alignment specified code the protocol writer is still responsible for invoking that code in the correct places in the protocol implementation. In a USC-enhanced C the programmer would only have to correctly annotate any network data — the compiler would handle any conversions needed.

Existing presentation layers are recognized as the most serious remaining bottleneck in the network data path[3]. USC could provide the basis for a simpler and much faster presentation layer stub compiler. The correct way to do this would probably be to select an intermediate form and write a layer on top of USC that supports the marshaling of arbitrary pointer based objects into and out of network form. Such a USC based stub compiler would be able to generate stubs for the entire C type system with a performance close to that of a simple data copy in most common cases.

6 Conclusion

We have designed and implemented a stub compiler that is flexible enough to eliminate the need for the manual generation of byte order and alignment dependent code in network software implementations. This stub compiler is fast enough that users have no incentive to bypass the stub compiler. Perhaps most importantly this work shows that presentation layer processing is not intrinsically slow and that careful application of modern compiler techniques can produce stub compilers that generate nearly optimal code.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

- [2] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.
- [3] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, Sept. 1990.
- [4] P. Hoschka and C. Huitema. Automatic generation of optimized code for marshalling routines. In *Proceedings of the IFIP Conference on Upper Layer Protocols, Architectures and Applications ULPAA '94*, Barcelona, Spain, 1994.
- [5] C. Huitema. Mavros: Highlights on an asn.1 compiler. Technical Report Internal Working Paper, INRIA Project RODEO, 1991.
- [6] C. Huitema and G. Chave. Measuring the performance of an asn.1 compiler. Technical Report Internal Working Paper, INRIA Project RODEO.
- [7] C. Huitema and A. Doghri. A high speed approach for the osi presentation protocol. In *Protocols for High-Speed Networks*. Elsevier Science Publishers, 1989.
- [8] ISO. Specification of abstract syntax notation one. Technical Report International Standard ISO 8824, International Standards Organization.
- [9] ISO. Specification of basic encoding rules for abstract syntax notation one. Technical Report International Standard ISO 8835, International Standards Organization.
- [10] H. Lin. Estimation of the optimal performance of asn.1/ber transfer syntax. *Computer Communications Review*, 1993.
- [11] R. G. Minnich. Methernetfs: A modified nsf which supports virtual shared memory. In *Proceedings of the Distributed and Multiprocessor Systems (SEDMSIV) Conference*, 1993.
- [12] J. Purtilo. Polyolith: An environment to support management of tool interfaces. In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 12–18, July 1985.
- [13] Sun Microsystems, Inc. *XDR: External Data Representation*, 1987.
- [14] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2), May 1993.

A USC Language Description

A.1 USC Type System

The USC type system is simple and supports the type void and the base types char, short, int, long, and enum. In addition, USC supports structures, unions, and arrays of these types, as well as bit-fields. Pointer types are only allowed in stub parameter and return value declarations. Pointer types are used to pass data by reference and to return values of the address of operation. The typedef operation is supported. USC does not support floating point types, or arbitrary pointer-based objects. Unions are only partially supported. USC allows unions to be defined, and a union's fields to be selected, but union assignment is not currently supported. Since C supports untagged unions there is no information indicating which union field is currently active and thus USC cannot predict which field to convert. Furthermore, USC must assume that the length of a union is determined by its largest field. In most network representation the length of a union is determined by the field which is currently active. Thus the USC union type cannot be used to generate stubs for XDR unions with are variable length.

While C does not support the declaration of variable length arrays, C programmers often get around this restriction by allocating arrays larger than that defined in the type. To support this, the USC type system is modified to allow the user to declare a variable length array. Variable length arrays may appear standalone or as the last element of a structure. A variable length array is defined as follows:

```
int a[name];
```

where name is a C variable name. The name used in the type definition must correspond to an integer parameter in the USC stub definition. This parameter is used to pass the actual size of the array to the stub.

The USC type system differs from the ANSI C type system in that USC allows different enumerations to define the same enumeration constant. This extension is needed to allow USC users to define stubs which convert between enumerations with different values for the same constant.

A.2 USC Data Layout Annotations

USC provides a notation for precisely defining the layout of each variable passed to a USC stub. USC makes no assumptions about the byte order of any defined type. The input file must precisely specify the correct byte order and offset of every type. Pragmas are used to inform USC of the native format in the compiler/host combination that will be used to compile and execute the generated stub.

All USC annotations are lists of four properties. The exact properties in the list is determined by context. A USC annotation found after a variable or parameter name is defined as follows:

```
type var(tsize, msize, alignment/offset, byte
order);
```

Where *tsize* is the number of bytes needed to represent the data type and *msize* the number of bytes the compiler has allocated to store this data type. *Tsize* must be less than or equal to *msize*. The third field is interpreted as an alignment in all cases except that of a field in a structure of union, in which case it is interpreted as the offset of that field within the immediately enclosing structure or union. The alignment field is a guarantee to the compiler that the address of the annotated variable modulo alignment is equal to zero. An alignment of 1 will always generate correct code. In general the higher the alignment specified the better the code USC will generate. It is possible to specify an alignment for a type that is more restrictive than the alignment used by the compiler. The byte order field is used to specify which memory bytes, in what order, are used to represent a given type. The syntax of the byte order field is a comma separated list of *tsize* distinct integers between 0 and *msize*-1 enclosed in angle brackets (*<1,2,3>*). A range may be used to abbreviate a list of integers. A range has the form *n..m* and is equivalent to the list *n, n+1, ... m* if *m* $\geq$ *n*. If *n* $>$ *m* the range *n..m* is equivalent to the list *n, n-1, ... m*. This list is interpreted as a transformation from the byte number of the variable to the offset of that byte from the start of the variable in memory. The USC type annotation for a 4 byte word aligned big endian integer is:

```
int a(4,4,4,<3..0>);
```

When describing a structure, the byte order field must be zero. USC derives the actual information from the annotations of the structure's fields. This notation can describe any C array type. For example an array of 10 shorts where each short is stored in the last two bytes of a word could be described as follows:

```
short a(2,4,4,<2,3>)[10];
```

The annotations found after a field name are identical to the annotations found after a variable except that the third element specifies the exact offset in bytes from the beginning of the structure. Given the offset USC can determine the alignment of any field in a structure from the alignment of the structure.

USC uses two separate annotations to describe bit-fields. Each bit-field name is annotated with the format of the underlying integer type. If several bit-fields are contained in the same integer they will have the same offset. After the bit-field size specifier another annotation specifies which bits in the underlying integer make up the bit-field. This annotation is analogous to the previous one, except that all of the values are in bits rather than bytes and the offset field must be zero. Note that the bit order of a bit-field is described relative to the byte order of the underlying integer type. For example two four bit bit-fields arranged in the same byte in little endian bit order at offset 4 from the beginning of a structure would be defined as follows:

```
u_int  x2  (1,1,4,<0>):4(4,8,0,<4..7>),
         off (1,1,4,<0>):4(4,8,0,<0..3>);
```

Note that type annotations are unrelated to the host for which USC generates stubs. Thus it is possible to generate stubs for an Intel x86 which converts a type in native DEC C, VAX format to native gcc SPARC format.

A.3 USC Type Compatibility

The introduction of data layout annotations introduces three distinct levels of type compatibility into USC. Two USC types are type compatible if their underlying ANSI C types are structurally compatible. Two USC types are copy compatible if they are type compatible and their annotations differ only in byte ordering. Two USC types are identical if they are type compatible and have identical annotations.

A.4 USC Stub definitions

Stubs are defined in USC as functions are defined in ANSI C. The user defines the parameters and return value to a stub exactly as they would in a C function except that USC's type system is used. The body of USC stubs are defined using a restricted subset of the ANSI C statement grammar. Only expression statements and return statements are supported. Statements are defined using a restricted subset of the C

expression grammar. The key feature of this expression grammar is that it works on annotated USC types. Assignments will correctly convert values when assigning between to structures with different layouts.

In the USC expression grammar component selection ($\rightarrow$ and $\cdot$), array subscription ($[]$), indirection ($*$), `sizeof` and address of ($\&$) are supported on all appropriate types. The assignment operation is supported between all type compatible USC types. Unlike C, assignment between array types is supported. The type of array indices and the type of the operands of the operations addition(+), subtraction(-), multiplication(*) and division(/) must be identical to one of the native base types given in the pragmas at the beginning of the USC program.

Parameters to USC stubs must be either pointers to any USC type, or a type copy compatible to a native base type. The value returned by a USC stub must be type void, a pointer to any USC type, or a type copy compatible to a native base type. Thus, USC stubs can take as parameters or return base types in any byte order. USC requires stub parameters that give the lengths of any variable length array arguments. The code below copies one variable length character array to another. `len` gives the current length of the arrays.

```
void foo (len, str1, str2)
    int len; /* Must be a native type */
    char str1[len];
    char str2(1, 4, 4, <0>)[len];
{
    str1 = str2;
}
```

Any USC stub that contains a variable length array definition as a parameter must have a native integer parameter whose name matches the name given in the array definition.

USC supports the inline qualifier found in many C compilers. A USC stub declared as inline will generate an inline function. In addition, USC supports the qualifier macro which directs USC to produce a C macro implementation of the specified stub. Currently the qualifier macro may only be used on stubs returning type void.

The stub `tcphdr` defined in Figure 1 shows how to define a stub to copy a TCP header from network format to DECstation 5000 native format. The generated C code swaps bytes and realigns the data. Less traditional stubs can also be generated. It is often useful to read and write fields into a network header stored in network format. A stub that peeks into a TCP header in network format and returns the offset field in a four byte, big endian integer would be defined as follows:

```
int(4,4,4,<3,2,1,0>) tcpgetoff (net_hdr *hdr)
{
    return hdr->off;
}
```

USC also provides for in-place modification of a data value. The pragma alias can be used to inform USC that two parameters will be aliased every time the procedure is called. To generate a stub which is intended to do an in-place translation of a tcp header one would use the following USC stub definition:

```
void tcphdr(net_hdr *src, native_hdr *dest)
{
    #pragma alias(src,dest)
    *src = *dest;
}
```

The stub generated assumes that the parameter `dest` is aliased with the parameter `src`. On machines where the layout of `network_hdr` and `native_hdr` are the same no code will be generated. USC will generate correct code if parameters to a stub are aliased regardless of the use of the alias pragma. However, such code will not be optimal. Note that only pointers may be aliased in this way and in-place conversion between two types with different lengths can be dangerous.

A.5 Failure Model

Generated stubs that raise runtime exceptions are very unpopular. USC generated stubs will not raise an exception if the following conditions are met: the alignment and offset information given in the USC source are correct, the user uses the USC `sizeof` operator to compute buffer length, and all values passed to USC stubs match their definitions in type and format. Because USC does not support pointer types this failure model requires no error checking in the stubs, which improves performance.