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];           /* May be native or not */
  char str2(1, 4, 4, <0>)[len];/* May be native or not */
{
  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 inplace conversion between two types with different lengths can be dangerous.

```
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 type 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 USC generates stubs for. 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.

I.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.

I.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. Thus assignments will correctly convert values when assigning between to structures with different layouts.

In the USC expression grammar component selection (-> and .), 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

Appendix I) USC Language Description

I.1) USC Type System

The USC type system is a subset of the ANSI C type system with a few extensions. Given the special purpose nature of USC only those ANSI C types which are commonly found in network headers were included in the USC type system. As stated earlier 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* and *arrays o*f these base types, as well as *bit-fields*. *Unions* are supported, although incompletely as union copies are not currently allowed. 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.

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.

I.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. For example, a USC annotation found after a variable or parameter name is defined as follows:

```
int a(tsize, msize, alignment, 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 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 **byetorder** 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 > 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:

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[1]. 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, 1986.
2. D. D. Clark and D. L. Tennenhouse, Architectural Considerations for a New Generation of Protocols. In Proceedings of the SIGCOMM' 1990.
3. D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, An Analysis of TCP Processing Overhead, IEEE Communications Magazine, June 1989.
4. C. Huitema and G. Chave, Measuring the Performance of an ASN.1 Compiler, INRIA.
5. C. Huitema and A. Doghri, "A High Speed Approach for the OSI Presentation Protocol", In H. Rudin and R. Williamson, editors, Protocols for High-Speed Networks, Elsevier Science Publishers, May 1989. IFIP.
6. ISO 8824 Specification of Abstract Syntax Notation One.
7. ISO 8835 Specification of Basic Encoding Rules for Abstract Syntax Notation One.
8. H. Lin, Estimation of the Optimal Performance of ASN.1/BER Transfer Syntax. Computer Communications Review, July, 1993.
9. R. G. Minnich, Mether-NFS: A Modified NFS Which Supports Virtual Shared Memory, In the Proceedings of the Distributed and Multiprocessor Systems (SEDMS IV) Conference, 1993.
10. Sun Microsystems, Inc., XDR: External Data Representation, 1987.
11. C. A. Thekkath and H. M. Levy, Limits to Low-Latency Communication on High-Speed Networks, ACM Transactions on Computer Systems, Volume 11 Number 2, May 1993.

and stores that can be done. 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. This is used to simplify certain sign extension optimizations. A zero bit copy indicates that bit j gets 0. This 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) | .
```

which are then optimized using algebraic simplifications. 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 compliers (and most manually generated stubs), USC will optimize across filed boundaries in structures. For instance, if two short fields may be correctly copied as a single word, USC does so. The optimizer and code generator are specifically designed to be as general as possible. The user only needs to specify 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 entirely 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 progam 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) Current Limitations and Future Work

USC assumes that the machine in question has 8 bit bytes and the size of all types other than bit-fields are multiples of 8 bit bytes in length. USC also assumes that all integers are represented in two's complement form. USC has not been tested on word addressable machines; however, no major difficulties are foreseen. As stated earlier USC cannot be used to representing dynamic encodings such as ASN.1/BER.

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

HP 735	ntoh	usc	rpcgen	rpcgen-opt	asn1/ber	asn1/opt
udp hdr	4.7	3.8	22	9	10	6.3
big hdr	37	16	212	165	46	19

Table 6: Cache Effects

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 results of this test are given in Table 6. The 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 totally unacceptable. However the performance of ASN.1/BER stubs generated by MAVROS is surprisingly close to the performance of the noth stubs.

Dec 5000 RTT	usc	ntoh	rpcgen-opt	asn1/ber
1,200 usec	2%	6%	30%	16%
800 usec	3%	9%	45%	24%
170 usec	13%	42%	212%	114%

Table 7: Relative Costs

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 Dec Station 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 [10] (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 improvements when used to marshal headers of very lightweight protocols. Clearly, the cost of using heavyweight 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.

5) Discussion

5.1) USC Implementation

USC generates code by minimizing loads and stores and doing algebraic optimizations[1] to the resulting mask and shift operations. The optimizer's first priority is to minimize memory access. 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 sized loads

	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)

gen with the optimizer enable. 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[8][3].

Next the performance of the MAVROS[3] ASN.1 complier was tested. 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
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

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. 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 [3] 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.

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 memcopy 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 IP and Ethernet addresses. [1] The results of this test case are given in Table 3.

	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	1.0	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. ntoh (usec)

These results show that the performance of USC generated stubs is generally superior to ntoh 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 ntoh stubs load and store each field of the data structure separately. The ntoh stubs use some halfword 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 ntoh 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 a UDP header is given below. 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));

r0 = *(int *)((char *)l_src + (4));

*((int *)((char *)l_dst + (4))) = (((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. 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 ntoh stubs. The second row of Table 4 gives the performance of stubs generated using rpc-

1. Because no calculations are performed on addresses it is common BSD practice to leave address in network byte order.

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 bitfield, 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, and unions. Please refer to Appendix I for a more in-depth treatment of the USC syntax and semantics of data type and stub declarations.

4) Evaluation

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 data structures marshaled consisted of a 4 byte long, an 8 byte UDP header, and a 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 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 Table's 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.

	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	.3	.6	4.5	0.5	0.9	10	o.3	0.7	8.0	0.2	0.4	3.5

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

```
  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;
}
```

Figure 1: The USC program tcp.USC.

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 that USC will generate code for, in this case gcc compiling for a DECStation 5000. The type net_hdr is a TCP header[1] 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 a TCP header in little-endian byte order with a 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 code shown in figure 1 is essentially C, with additional information that precisely describes the data layout of the types defined. 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, and 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 example lines from figure 1:

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

1. Actually a minor variation of the tcp header format is used to better demonstrate the features of USC.

Unfortunately this notation cannot be used to describe types encoded using a dynamic format; a format where the encoding depends upon the runtime value of a variable. For example in ASN.1/BER format the encoding of an integer varies in size based upon the value of the integer. Dynamic formats are very expensive to encode and decode: it takes a minimum of 58 instructions to encode an ASN.1/BER integer[8] as apposed to 2 instructions for most statically encoded integers. Dynamic formats are to our knowledge only found in intermediate forms. All hosts and compilers use static forms to represent native data. While USC's inability to specify dynamic formats means that it cannot be used to read and write headers in ASN.1/BER format, it also allows USC to concentrate on the efficient conversion of the more common static formats.

USC uses modern compiler optimization techniques to generate 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 headers. 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.

USC assumes that both the source and destination of any stub are in contiguous memory. Some argument marshaling systems assume that the encoded version of a data type may be spread across several distinct memory buffers. For short headers it is much more efficient to copy a header broken across two buffers into contiguous storage rather than checking the buffer size before every header access.

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. The user uses this syntax to declare type definitions 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>);
```

tion layer stub compilers. Most of these differences are a result of differences between the header marshaling problem and the data marshaling problem. This section attempts to describe and motivate each of these differences.

The biggest difference between header marshaling and data marshaling is that in header marshaling the network formats for headers are fixed by the protocol definition and nearly impossible to change. Thus USC must generate stubs to pre-existing formats and because of this USC has no fixed intermediate form. USC converts a given data type in one specified format to another specified format. To marshal a TCP header a USC user defines a C structure to represent the TCP header and specify two formats for that data structure: the local compiler/host format and the network format. The user would then define USC stubs that convert a TCP header in one format to another. USC allows the user to describe nearly arbitrary layouts of data types and conversions between them. In contrast, traditional stub compilers convert data to and from network form as follows. Sender data, in native host/compiler format, is converted into a network independent format for transmission to a receiver who will convert the independent format to its native host/compiler format. The intermediate format is generally fixed. Thus the stub compiler need only know of the local host/compiler format and the fixed intermediate form. A fixed intermediate form makes it impossible to use a stub compiler to generate header marshaling code for an existing protocol such as TCP.

The lack of a fixed intermediate form raises the question of coverage: exactly what data formats will be supported by USC. Our first assumption was that any network protocol would be written in C or C++. Thus the USC type system is based upon the C type system. Because the types commonly encountered in header marshaling are quite simple USC uses a limited subset of the C type system that does not support pointers. Table 1 gives the distribution of data types found in the protocol headers in our protocol library. The only pointers present in this distribution are found in Sun RPC; a protocol that uses a presentation layer stub compiler to marshal its header[1]. In addition to being rare, pointers are difficult and time consuming to marshal and require significant amounts of storage management and error checking code.

Type:	Frequency:
short	34%
char	25%
int/long	23%
struct	7%
enumerated	5%
bitfield	4%
array	1%
pointer	1%

Table 1: Type Frequencies

Given this restriction to simple C types, the question becomes what formats of these types will USC support. We decided that at a minimum USC must allow a user to specify any data format found on any actual host/compiler combination. In general the host determines the data representation of the base types and the compiler determines the alignment of data in composite types. The demise of most ones complement and odd byte size architecture implies that a relatively small set of potential data formats will cover almost all modern architectures. However as the C standard places very few restrictions on the potential alignment of data types so the alignment annotations in USC must be flexible. Therefore we have defined a notation (given in the Appendix) which supports the nearly arbitrary alignment of a set of simple base type.

1. Note that USC can marshal Sun RPC headers. USC one would treat the Sun RPC header as series of simple C structures.

1) Introduction

Presentation layer processing is becoming recognized as one of the last remaining performance bottlenecks in network software[2]. Presentation layer processing, also called *data marshaling*, involves converting user data between different host formats. What is less commonly understood is that the network code, itself, faces a similar problem. Network software is riddled with code that converts data from one format to another, the most common example being reading and writing headers. We call this *header marshaling*[1].

There are two common approaches to solving the header marshaling problem. One is to use presentation layer stub compilers to generate header marshaling stubs. For example, Sun RPC uses XDR[9], and the entire OSI protocol suite uses ASN.1/BER[6][7]. A second approach is to implement header marshaling code by hand, perhaps with the aid of simple macros such as the ntoh suite found in Unix. Both of these approaches have significant problems. In the former case, the use of a heavy-weight mechanism adversely effects protocol latency. In the latter case, manually generated code is difficult to write because it requires knowledge of details about the compiler and host architecture. This also tends to make the code non-portable. To make matters worse, one is likely to have to support several such mechanisms on any given host.

This paper introduces a simple solution to the header marshaling problem. We have designed and implemented a new special-purpose 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 combines the best features of manual and automatic generation of header marshaling code. In summary USC:

- Automatically generates code from a concise specification.
- Generates nearly optimal code. USC stubs are as fast or faster than hand coded stubs and up to 20 times faster than stubs generated by presentation layer stub compilers.
- Is protocol independent. USC can be used to marshal the headers for most existing protocols including headers that are defined using some existing presentation layer formats.
- Provides nearly unlimited access to network data. USC can generate stubs which efficiently peek into the middle of a large data structure stored in network format.
- Is easily portable. There are no USC libraries, include files, or ifdefs.
- Automatically figures out the alignment and byte order of any C data structure on any C compiler.

While this paper concentrates on the use of USC to solve the header marshaling problem. We believe that USC provides the basis for an efficient solution of the data marshaling problem. USC can be viewed as a highly optimized code generator for a more complex presentation layer stub compiler. Using USC to generate the code to copy and byte swap simple composite types could result in significant performance improvements compared to traditional presentation layer stub compilers.

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 more thorough description of the USC language is given in Appendix I.

2) USC Design Principles

The basic design of USC differs in several critical ways for the design of traditional presenta-

1. Note we also include such problems as reading and writing the control registers on devices with non-native byte orders as part of the header marshaling problem.

USC: A Universal Stub Compiler[1]

Sean O'Malley, Todd Proebsting, and Allen Brady Montz

TR 94-10

Abstract

USC is a new stub compiler which can be used to generate stubs which perform a wide variety of data conversion operations. USC is flexible and can be used in situations were previously only manually 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 designed to compare USC performance with the best manually generated code and traditional stub compilers.

March 15, 1994

Department of Computer Science
University of Arizona
Tucson, AZ 85721

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