Fast Decoding of Tagged Message Formats

Trevor Blackwell
Division of Applied Sciences
Harvard University
Cambridge, MA 02138
tlb@eecs.harvard.edu

Abstract

Many important protocols, such as Q.2931 or any protocol based on the ASN.1 Basic Encoding Rules, are transmitted using tagged message formats, in which a message can be considered as a sequence of interleaved tag and data fields, where tag fields define the meaning of subsequent fields. These messages are computationally expensive to decode, partly because decoding each data field requires testing one or more tag fields. Evidence suggests that in some applications, although the potential space of message encodings may be very large, only a small number of message layouts are seen frequently, and thus some of the work required in decoding can be amortized over many messages. This paper analyzes the use of run-time code generation to generate optimized decoding instruction sequences for received messages matching previously observed layouts, and describes a prototype system that applies the techniques to decoding the Q.2931 and ASN.1 BER protocols. In the average case, substantial performance gains are seen.

Key Words: Encoding Rules; Protocol Implementation; Q.2931; ASN.1; Runtime Code Generation.

1. Introduction

A number of important network protocols, such as Q.2931 [26] (a standard protocol used to request connections in ATM networks,) or any protocol using ASN.1's Basic Encoding Rules (BER) [27,28] use tagged message formats. An encoded message consists of tag fields interleaved with data fields. The tag fields define the size, position, format, and meaning of subsequent fields.

Before software can take action based on these messages, they must be unmarshalled into a format convenient for further processing, which will depend on the implementation language. This is called presentation layer processing.

It has been widely recognized that presentation layer processing can be a significant bottleneck in high-performance systems. Performance of both the upward direction (decoding or unmarshalling) and the downward direction (encoding or marshalling) are important topics; however this paper addresses only the upward direction.

Decoding tagged messages is time-consuming. For each field, the software must make one or more tests based on the tags, and dynamically choose different parts of the decoding algorithm for each field. The cost of testing and choosing greatly dominates the cost of simply transforming the values from the source format to the convenient internal format.

Sample [10] showed that with some effort, an ASN.1 BER decoder can be made as efficient as a naive XDR implementation; however, this performance is still much slower than it could be. A highly optimized decoder presented by Lin [7] showed that the best achievable performance for decoding any ASN.1 BER message is fairly slow: 37 to 56 SPARC instructions for each integer-valued field. Lin used a hand-optimized decoder, with most error checks omitted, and placed some restrictions on the values allowable in tag fields. Lin does not give timings for these instruction sequences, but as the decoder contains mostly load-test-branch sequences, most processors will require more cycles per instruction (CPI) than for "average" code [23]. Analysis of ISODE [25], a widely used presentation layer building toolkit, shows it to be about 25 times slower than Lin's optimal results.

Significant efforts [11,12,13,14,15,16,17] have gone into in developing "lightweight" transfer syntaxes with similar functionality to the BER, to reduce the cost of encoding and decoding. Mitra [18] gives an overview of various alternative encoding rules. Proposals for parallel processing [19] and hardware-based [20] decoding have also been given. It is hoped that the techniques presented here can be applied to achieve good performance from standardized transfer syntaxes, without requiring special or expensive hardware.

This paper is organized as follows. Section 2 describes the steps required for decoding in general.

To appear in IEEE Infocom '96

Section 3 discusses how the techniques of run-time code generation (RTCG) and pattern matching can be applied to the task of fast message decoding. Section 4 describes the implementation of a prototype RTCG-based decoder. Section 5 describes related work. Sections 6 and 7 analyze the performance trade-offs in using RTCG. Section 8 describes alternative implementation strategies. Section 9 concludes with some comments about the appropriateness of RTCG for message decoding in various environments.

2. Message Decoding

The presentation layer has the task of converting messages from a form convenient for processing (i.e. language-specific data structures) to a form that can be transmitted over a network, and back to a (generally different) convenient data structures on the receiving end. A message consists of a sequence of records, each of which can be either a primitive type, such as an integer or character string, or a composite type which contains other records. Records can be preceded by a type tag; in ASN.1 BER every record is tagged, whereas in Q.2931 a single tag can identify a structure containing several elements.

The encoding rules studied here are defined such that messages can be decoded a byte at a time, from beginning to end. Decoders are usually written as recursive descent parsers, such that the programmer (usually aided by some automated tools) defines a function for every record type that unmarshalls the data from the transmitted format to the convenient internal representation. Thus, the decoder spends much of its time examining tag fields and dispatching to code to handle subsequent fields.

3. Exploiting Locality in Message Layout

For the purposes of this paper, we will say that two messages have the same *layout* when all the data fields are of the same size and in the same place in the received message. This implies that all the tag fields are identical. A stream of messages where relatively few distinct layouts occur is said to have good message layout locality.

The number of possible message layouts is very large for any non-trivial protocol. Even among messages having the same meaning to the receiver, the number of possible layouts is exponentially large, for at least the following reasons:

- Optional fields often have defaults. Thus, the presence of a field with the same values as the default does not change the meaning
- Fields can often be marshalled in any order
- There may be “extended” fields, ignored by the particular receiver.

Despite the large space of possible encodings, many network entities tend to produce only a few distinct message layouts under normal operation, although the common set of encodings is hard to predict in advance, and may change over long periods of time.

For a given message layout, a *decoding sequence* can be generated that transforms a message having that layout directly into the convenient internal format. The decoding sequence is a piece of straight-line executable code consisting of loads, simple data transformations, and stores.

The optimized decoding sequence can be much faster than a general decoding algorithm. First, all tests and branches are eliminated. Essentially all pointer arithmetic, a significant cost in conventional decoders, is eliminated. Because the alignment with respect to word boundaries of all elements in the message is known, individual byte loads and stores can be merged into words loads and stores. Because the code is straight-line, and all memory reference dependencies are known, the code can be multiply issued on superscalar processors.

By generating optimized decoding instruction sequences for each message layout seen by the decoder, a substantial performance gain can be achieved. The process is as follows. When the system receives a message, it is compared against a set of templates generated for previously received messages. If it does not match, then the message is decoded from scratch, and a new template and optimized decoding sequence are generated. If a message does match against a template, the optimized decoding sequence is used, and the message is decoded very quickly.

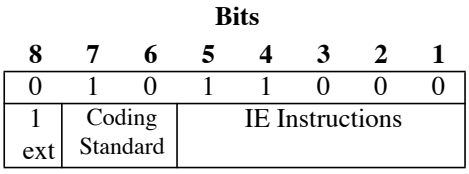
As will be seen in section 6, the decoding time is reduced by a factor of 4 to 5 over Lin’s results, and by a factor of 70 to 100 over some available decoding implementations.

4. Prototype Message Decoder

Each stage in the processing of messages is now described in detail. There are many ways in which each stage could possibly be done; however this discussion will focus on the design options chosen for our prototype. Section 8 describes some other organizations for the decoding system, which may be more convenient for particular applications.

4.1. Decoding from Scratch

When a message is received with an unfamiliar layout, it is parsed according to the rules of the encoding format, using a recursive descent parser as described in section 2. Methods of generating such parsers automatically from abstract specifications of message




```
parsef("ii:^01011000 1aabbbbb",
  &ie->ie_codingStd,
  &ie->ie_instruction);
```

FIGURE 1. Example of decoding part of a Q.2931 message. The top shows part of the header for a particular information element, as given in the ATM Forum UNI Spec (v 3.0). [] The C code at the bottom matches the header in the first octet, matches a 1 in the `ext` field in the second octet, and copies the two fields in the second octet into the `ie` structure.

formats are known, and toolkits are available [25]. The prototype provides a toolkit for parsing messages, which provides an API in the spirit of C's `scanf` [33]. A format string describes the meaning of each of a range of bits in the input.

An example will be helpful. Figure 1 shows a typical piece from which a parser is constructed. The top shows part of a message format in Q.2931 signalling defined in the UNI 3.0 standard [26]. The format string has two parts, separated by a colon. The first declares the types of fields that are being decoded. The "`ii`" indicates that there are two big-endian `int` types. The "`^`" is a failure point - if the match fails at some later point in the format string, the parser will back up to the most recent failure point. If no failure point is given, then it issues an error. Each "`1`" or "`0`" in the format string matches a one or zero bit in the message. Each "a" indicates a bit that is part of the first field; each "b" indicates a bit that is part of the second field, and so on.

While this programming style does not produce very fast code, it is very convenient. Our decoder for the complete Q.2931 message syntax is 300 lines and is nearly as easy to read as the standard itself.

While the parser is processing the message, it records the values of tag fields which were used to make a decision, and the value of all fields in the decoded representation as a function of bits in the received message. This information is sufficient to build both a template to recognize future messages with the same layout and an optimized decoding sequence.

4.2. Memory Allocation

The decoded representation is a hierarchical data structure, such that optional fields are accessed through a pointer which is nil if the field is not included. This requires dynamic memory allocation which is a substantial overhead for other decoding systems (this is quantified in section 6).

In the RTCG system, memory allocation need only be done when decoding a message from scratch. The prototype system allocates a single, extensible buffer for the entire decoded representation. Allocation is done simply by extending the buffer. Thus, the output of the optimized decoder is to a contiguous region of memory, and the optimized decoding sequence can refer to words in the decoded representation as an offset relative to the start of the buffer. Using a contiguous buffer may also make more efficient use of the processor's data cache than if each part of the data structure were allocated independently.

4.3. Code Generation

The intermediate representation of the decoding sequence is a set of expressions, one for each word in the output buffer. Expressions are constructed from a subset of the operators in the C language.

The expressions are built incrementally by the from-scratch message parsing API. The next step is to generate executable machine code from the expressions.

The prototype compiler generates executable code for the DEC Alpha. It is surprisingly compact: including the optimizer, it consists of about 2000 lines of C++, two orders of magnitude less than Gcc 2.7.2. It can be much simpler than a general-purpose compiler because it generates only straight-line code, and thus does not have to implement control flow analysis or any global optimizations. It also supports only a single integer data type. It does register allocation by graph colouring, and spills variables to the stack when necessary. Its simple instruction scheduling is effective, as large amounts of instruction-level parallelism are easily found in most decoding sequences. In addition to doing common subexpression elimination and constant propagation, the optimizer knows 25 specific optimizations, which were chosen ad-hoc by looking at the output code for several cases, and adding optimizations until all the obvious inefficiencies disappeared. Interestingly, the code generated by the prototype is about 35% faster than `gcc -O2` for the messages we tested, because:

- it knows that the input and output buffers to not overlap - thus it can eliminate multiple loads from the same address even if there is an intervening store, and it can move loads and stores past each other for improved scheduling. C compilers must make much more conservative assumptions about pointer aliasing.
- it knows the alignment of input and output regions, so it can merge multiple bitfield loads into a single

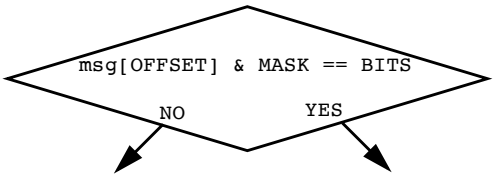
word load, and multiple shifts. Especially on the Alpha architecture, which has fast byte extraction instructions but no byte load instructions, this is a significant advantage. On a big-endian machine, it would frequently be able to extract a multi-byte integer with a single instruction.

4.4. Template Matching

Received messages must be quickly classified as to which (if any) message layout template they match. The prototype implementation builds a binary decision tree where each node matches some bits in an incoming message.

In the prototype, the decision tree is fully regenerated whenever a new template is added. Techniques for incrementally adding new paths to decision trees exist but seemed unnecessary, as the cost of regenerating a reasonably-sized tree was less than the cost of compiling a optimized decoder. Simple techniques are used to achieve a relatively balanced tree, although there are no guarantees on the quality of the balancing.

Each non-leaf node in the decision tree is a decision node of the following form:



where `OFFSET`, `MASK`, and `BITS` are constants defined in the node, and `msg` is the received encoded message, as an array of words. The leaf nodes of the tree are either an optimized decoding sequence or an indication that the message must be parsed from scratch.

Since every word in the message must be checked against the template, classifying a k word message from among a set of n takes $k + O(\log n)$ operations when the tree is well-balanced.

4.5. Integrated Layer Processing

Previous work [21] has pointed out several reasons why conventional protocol layering, as exemplified by the OSI reference model, does not lead directly to high performance implementations. However, manual integration of layer processing requires substantially increased development, debugging, and maintenance efforts.

A major function of layer processing that complicates integration of layers is reassembly. Decoding messages that are not contiguous in memory adds a significant extra overhead, as each reference must be checked for overflow into a new fragment. In the proposed system, it is feasible to generate an optimized decoding algorithm to take its source from multiple message fragments of known length. In systems where fragmentation tends to be into units of a few predictable sizes, including the fragmentation boundaries as part of the definition of the layout should not substantially increase the number of layouts that must be handled.

5. Related Work

Run-Time Code Generation (RTCG) has a long and checkered history - Keppel [2] gives an overview of some of it's many uses and provides many more references.

This particular application of RTCG described here is that of specialization for efficiency - creating a special purpose version of a general purpose algorithm that can do the job much faster. This has been used, for instance, by Pike [4] to optimize special cases of graphics block copy operations, and in the Synthesis kernel [5] to optimize common cases of data reads and writes to file descriptors. SELF, a dynamically typed object-oriented language, dynamically generates versions of methods optimized for particular receiver types [6]. Dean [9] showed how to extend SELF's specialization to multi-methods - methods whose implementation depends on the type of multiple receiver classes.

RTCG is used in other applications such as the Berkeley Packet Filter [3]. The BPF is not an example of specialization for efficiency, but rather a way of specifying a policy (which packets to capture) such that it can be efficiently implemented in the kernel.

The 4.3 Reno implementation of NFS [22] demonstrates the performance gains, as well as the difficult implementation issues, of presentation layer decoding directly from message buffers that have been fragmented by a network.

Although the prototype does not use any third-party toolkits to generate code, some are available [8] and might ease porting to other architectures.

6. Analysis of Decoding Time

Experimental performance results are presented for both Q.2931 and ASN.1 BER decoders. For the present section, we will assume that only a small number of message layouts (less than 100, say) are encountered; the following section will discuss the potential for large numbers of message layouts. Thus in this section we measure the decoding time for previously processed message layouts.

Two processing tasks are measured: decoding a Q.2931 `setup` message, as generated by the QCC API bundled with the SunATM network interface [29], and decoding a 100-element structure in ASN.1 BER coding.

The 119-byte Q.2931 message consists of the mandatory elements for a `setup` message and one optional element: adaptation layer parameters appropriate for AAL5. It reflects a typical message used to request connections in an IP networking environment. The 381-byte ASN.1 BER message contains a sequence of 50 integers, alternating with 50 booleans. The integers are chosen randomly to have a even distribution of 1, 2, and 3 byte codings, and the booleans are chosen uniformly at random.

Performance numbers are reported for the DECStation 3000/400 [30], based on the DEC Alpha 21064 CPU, which supports limited dual issue (memory operations can issue in parallel with integer ALU operations under certain conditions). All code is compiled with Gcc 2.7.0, using `-O2` optimization. Both instruction counts (obtained with Atom [32], or by single-stepping in a debugger) and cycle counts (measured with the processor's per-process cycle counter) are reported. The number of cycles per instruction (CPI) is also given. In order to measure timings with warm caches, each operation is run twice in succession using the same data, and timings are measured for the second iteration. To eliminate extraneous system activity, each experiment was repeated several times, and the lowest timing numbers are reported. Cycle counts for all decoders, especially the more complex ones, must be considered to have a significant error component due to the vagaries of instruction and data cache conflicts [23]. It should be noted that all numbers except Lin's include several overheads such as buffer management.

Table 1 compares the performance of the prototype decoder Lin's results, and with the ISODE decoder. ISODE decodes messages in two phases. First, it converts the BER-coded representation into a tree structure of elements, one for each type-length-data tuple in the binary representation. Then, code generated from an ASN.1 specification converts the tree structure into C data structures as defined by the application layer interface.

Estimates of the optimal performance are computed from Lin's formulas, taking into account the length distribution of the encoded fields. As Lin's formulas give instruction counts, cycle counts are estimated by multiplying by the average CPI (cycles per instruction) reported for the other decoders. Although Lin's figures give instruction counts for the SPARC architecture, these are at least as low as would be possible on the Alpha architecture. In fact, because Lin's code presumably contained many SPARC byte load/store instructions which require a sequence of two to six instructions on the Alpha, the optimal results given are somewhat optimistic.

Table 2 compares the performance of the prototype decoder with Vince [31], a publicly available UNI signalling implementation, decoding the sample Q.2931 message. Vince uses a table-driven approach to convert incoming messages into an association list of information elements used by higher software layers.

As can be seen in Tables 1 and 2, the RTCG prototype fast decoder ran 4 to 5 times faster than Lin's optimal results, and 70 to 100 times faster than the available implementations. It required between 3.6 and 6.3 instructions per byte of input.

The ISODE decoder for ASN.1 is about twenty times slower (by instruction count, ignoring SPARC/Alpha differences) than Lin's optimal results, requiring 266 instructions per byte of input. This does not reflect badly on the ISODE implementation; ISODE uses a fairly flexible decoding strategy that permits easy debugging and extensive error checking. Some valuable features of ISODE's decoder architecture require a separation of the ASN.1 BER from the data type specification, at the cost of some performance. It also includes (as does the RTCG prototype) functionality that Lin's decoder does not, such as error checking.

The Vince decoder took 279 instructions per byte of input, in the same range as the ISODE decoder. About half the instructions are executed as part of the `malloc` function, which required an average of 75 instructions for each call. (The `malloc` from the OSF1 3.0 library is used instead of Vince's own memory allocation function because it is significantly faster). Careful optimization could certainly make Vince faster.

TABLE 1. Performance of ASN.1 Decoders

ISODE ASN.1 BER Decoder	Estimates from Lin's Optimal Results	Prototype Fast Decoder
101500 inst.	4100 inst.	1307 inst.
174000 cycles	6264 cycles	1338 cycles
1.715 CPI	1.528 CPI est	1.02 CPI
1310 ∝S	47.0 ∝S	10.0 ∝S

TABLE 2. Performance of Q.2931 Decoders

Vince Q.2931 Decoder	Prototype Fast Decoder
33100 inst.	750 inst.
62100 cycles	792 cycles
1.876 CPI	1.171 CPI
470 ∝S	6.6 ∝S

7. Analysis of Message Layout Diversity

The cost of classifying messages using the tree structure algorithm in the prototype is inexpensive for any reasonable number of templates. The main limitation to the level of message layout diversity that the decoder can support is memory. Each known message layout requires a decoding sequence (the decoding sequences generated in the tests in the previous section consumed about 2 KB each) and several nodes in the decision tree. Thus commodity workstation technology in a signalling environment can support on the order of thousands of message layouts.

In our estimation of normal usage of the ATM Forum UNI 3.0 specification, analysis shows that for the SETUP message (the richest message type), on the order of 10^{43} message layouts are possible, considering elements passed transparently by network switches to be internally opaque so that two elements with the same length but different internal contents do not result in a different message layout. Use of some extra fields that are defined syntactically but which currently have no useful semantics (such as the codeset shift options) could push this number much higher.

Despite the enormous space of possible messages, Q.2931 signalling implementations seem to produce a small number of layouts in actual practice. For instance, the Sun QCC library (part of the SunATM network adapter software) defines an API to allow user processes to set up ATM connections. Normal usage of this API leads to only one message layout for each of the nine supported message types (it is possible for user programs to add or change information elements by directly manipulating the binary message contents). The IP over ATM system included with the SunATM adapter produces four more message layouts. In this environment, the RTCG system will quickly learn all the message layouts, and provide uninterrupted fast operation.

In the UNI environment, messages are sent hop-by-hop. This means that any messages received by a switch will have been generated by its directly connected neighbours (few ATM switches have more than 64 ports.) Thus, the potential number of distinct encoders that an decoder must deal with is small, and the likely number of distinct implementations is even smaller.

Designing an encoder to ensure that message layout diversity is kept to a minimum is not difficult. Certainly a developer using RTCG-based decoders would design his encoder in such a way. It might be reasonable to develop a set of guidelines for encoder developers to ensure maximum performance with RTCG-based decoders. The author thinks that adding optional guidelines to a standard to achieve maximum performance is a much better thing than developing a new and incompatible standard.

When new parsers must be compiled frequently, the cost of compiling is an important consideration. In the research prototype, compiling costs around 150 mS for a 300 instruction decoder (DEC 3000/400). This is not a slow compiler; compiling the same expressions with GCC 2.7.0 takes 7.4 seconds. However, Engler reports that his lightweight compiler costs on the order of 300 machine instructions for every instruction generated [8], which would require on the order of 1 mS on the same machine. (Engler's compiler does not performs many of the important optimizations performed in the prototype, however.) The use of lightweight compiler technology could reduce the potential performance penalty of an increase in message layout diversity.

8. Design Alternatives

In order to explore the upper limits of message layout diversity at which run-time specialization is worthwhile, our prototype uses a custom, fast code generator. Compiling a new fast decoder takes on the order of a hundred milliseconds. An alternative to using a custom code generator would be to write out source code, invoke a standard compiler, and dynamically link the resulting object into the executing code. There is adequate support for this on most UNIX platforms.

This strategy increases the cost of compilation by 20 to 50, both in terms of time and space overhead. Compilation times using a C compiler such as GCC are about fifty times longer. The various overheads in maintaining a dynamically linked code module are tens of kilobytes rather than the two kilobytes consumed by the prototype. It might be reasonable to use this strategy in systems where the total MLD is up to a hundred, whereas our prototype handles MLDs up to a few thousand. There may, in fact, be few systems with MLDs in this range - the majority of systems may not need an integrated code generator. In real time environments, however, occasional delays of a hundred milliseconds may be more tolerable than delays of multiple seconds.

Another possibility is to not require online management of the set of optimized decoders. A reasonable architecture might use a conventional decoder to handle messages for which optimized decoders do not exist. Based on some kind of traces, an off-line process could be run occasionally to generate new optimized decoders. In systems where most of the messages can be classified into a small number of layouts, but where there is a potentially large number of occasionally occurring layouts, this strategy may be superior to always generating an optimized decoder for unrecognized messages. A disadvantage is that it requires two separate decoding engines.

9. Conclusions

The proposed use of RTCG for message decoding has both positive and negative impacts on system performance, complexity, and robustness.

RTCG can lead to tremendous performance gains in environments with low to moderate message layout diversity. It achieves this without sacrificing the simplicity, extensibility, and functionality of the data format specification code. Performance is 70 to 100 times faster than some existing decoders, and 4 to 5 times faster than Lin's optimal results. This should not be taken as a criticism of Lin's results, as they represent an optimum for all message streams, while RTCG system performs well only for message streams without high message layout diversity.

Because the code implementing the encoding rules is rarely executed, it can be specified in a high-level, flexible manner, without spending programmer effort to increase efficiency. This allows the protocol implementation to closely follow standard layering and structuring conventions without performance penalty. It also allows extensive error checking to be done. Frills that would be impractical if the decoding sequence were in the critical path can be provided — for example, the prototype produces a Postscript document diagramming the message layout. In these respects, it preferable over a decoder implementation such as Lin's, which required the high-level rules specifying the expected data types to be implemented manually as highly optimized, bit-fiddling code. This adds a high cost to changing or adding to the type specification, and probably makes the type specification hard to decipher by reading the code.

Performance optimization of network code is frequently done by manually adding fast paths for common cases of protocol messages. The proposed system may have advantages over such systems in some environments. In particular, changes in conventions for what fields are commonly used may "break" a fast path implementation, in the sense that some small change in the implementation of a communicating party may cause a dramatic performance decrease.

The trade-off between manual and automatic generation of optimized decoders depends on the amount of freedom in the message encodings. The carefully designed formats of TCP and IP headers allow alignments of message fields to exploited manually, at least on 16- and 32-bit machines. TCP/IP has two degrees of freedom in the header format: the length of IP options, and the length of TCP options. The TCP implementation in 4.3BSD was implemented to be fast only for packets with no IP or TCP options. TCP/IP performance is, in this sense, fragile with respect to implementations which make use of these options. In 4.4BSD, use of the TCP timestamp option became desirable. Although many possible encodings of the necessary options are possible within the TCP specification, only one additional fast path was added for packets with a particular combination and formatting of timestamp options specified in RFC1323. Thus, there is some precedent for guidelines to reduce message layout diversity.

In richer message formats such as Q.2931, manual discovery of properties that might allow fast paths appears impractical. Although with sufficient effort one could write an optimized decoder that would work for a few selected message layouts, the system would be hopelessly fragile with respect to implementation changes or changes in traffic patterns. Also, the RTCG approach can handle thousands of message layouts, whereas manual coding of more than a few would be prohibitive.

The proposed system has a different kind of fragility which needs to be considered carefully in any application. In many protocols, it is possible to have an exponentially large number of message layouts occur in normal operation. A new extension to a protocol, or an increased use of optional fields may drastically increase the number of layouts encountered. This can decrease system performance by orders of magnitude, as a general parsing, compilation, and addition to the decision tree would occur for a large fraction of received messages.

An possible concern is the feasibility of denial-of-service attacks, performed by sending large numbers of messages with different layouts to a RTCG-based decoder. This can be caused by malicious users or by broken equipment. Such issues are not unique to this system; they are a concern in any system where processing unusual messages takes substantially longer than processing common messages. For the protocols discussed here, where compiling a new layout requires on the order of a tenth of a second, an attacker would have to send a fairly steady stream of messages to deny service to others. Techniques such as *babbler isolation* (detecting flurries of unusual activity from a particular source and rejecting further messages from the same source for a period of time) can be used to protect network elements against such disasters.

The RTCG approach will be used as part of a UNI signaling implementation being developed to demonstrate the feasibility of ATM connection setup at performance levels of 10000 connections per second within the established UNI signalling framework based on workstation technology readily available in 1995. Currently, a few UNI signalling implementations are available which can handle 50 to 200 connections per second, but we know of no implementations with much higher performance. Fast signalling performance will be important if ATM is to provide acceptable performance for the sort of data connections that dominate the Internet,

which tend to be short-lived. In the common case, a network switch must receive four Q.2931 messages for each setup/teardown. A reasonable total processing budget is 30 uS for the `setup` message, and 20 uS each for the other three messages. On a workstation twice as fast as the 3000/400 (such workstations have been affordable since mid 1995), the RTCG decoder would require about 10% of the processing budget. As decoding is only one of many steps involved in UNI signalling, a slower decoder would be unacceptable. Clearly, software similar to the Vince or ISODE decoder falls hopelessly short of our signalling performance goals. While a highly optimized static decoder such as described by Lin might meet our performance goal with somewhat faster processors or modest levels of parallel processing, the RTCG-based decoder has a clear advantage.

10. References

More references with abstracts, and online copies of some of the references below can be seen at http://www.eecs.harvard.edu/~tlb/references.html

1. Harry R. Lewis, Larry Denenberg. *Data Structures and their Algorithms*. 1991, Harper-Collins.
2. David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, University of Washington (EECS dept), 1991.
3. J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. of the Eleventh ACM Symposium on Operating System Principles*. 1987
4. R. Pike, B.N. Locanthi, and J.F. Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software - Practice and Experience*, 15(2):131-151, February 1985.
5. Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1):11-32, 1988.
6. C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. Proceedings of OOPSLA '89.
7. H.A. Lin. Estimation of the Optimal Performance of ASN.1/BER Transfer Syntax. *Computer Communication Review*. 23(3), July 1993.
8. D.R. Engler, T.A. Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generation System. In *Sixth Internation Conference on Architectural Support for Programming Languages and Operating Systems*. 1994.
9. J. Dean, C. Chambers, D. Grove. Selective Specialization for Object-Oriented Languages. In Proceedings of *SIGPLAN '95*.
10. M. Sample, G. Neufeld. Implementing Efficient Encoders and Decoders for Network Data Representations. *IEEE INFOCOM '93*, p 1144.
11. C. Huitema, A. Doghri. Defining faster transfer syntaxes for the OSI presentation protocol. *Computer Communication Review*, vol.19, no.5, p. 44-55. Oct. 1989.
12. A. Cardoso, E. Tovar. Defining more efficient transfer syntax for application layer PDUs in field bus applications. *Computer Communication Review*, vol.22, no.3, p. 98-105. July 1992.
13. M. Bassiouni, M. Loper. Performance tests and flexible decoding for transfer syntax in real-time applications. *IEEE 13th Annual International Phoenix Conference on Computers and Communications*, p. 512, 134-40. 1994.
14. M. Besson, A. Doghri, C. Huitema. High performance heterogeneous transmission using the OSI presentation protocol. *Proceedings of the Third International Symposium on Computer and Information Sciences*, p. xii+732, 1-12. 1989.
15. M. Bever, U. Schaffer. Coding rules for high speed networks. *IFIP Transactions C (Communication Systems)*, vol. C-7, p. 119-32. 1992
16. H. Horiuchi, S. Obana, K. Suzuki. Efficient packed encoding rules (EPER) for ASN.1 and its evaluation. *Transactions of the Information Processing Society of Japan*, vol.36, no.2, p. 492-500. Feb. 1995.
17. J.R. Pimentel. Efficient encoding of application layer PDU's for fieldbus networks. *Computer Communication Review*, vol.18, no.3, p. 14-44. May-June 1988.
18. N. Mitra. Efficient encoding rules for ASN.1-based protocols. AT&T Technical Journal, vol.73, no.3, p. 80-93. May-June 1994.
19. D. Ghosal, T.V. Lakshamn, Y. Huang. High-speed protocol processing using parallel architectures. Proceedings *IEEE INFOCOM '94*, p. 159.
20. M. Bilgic, B. Sarikaya. Performance comparison of ASN.1 encoder/decoders using FTAM. *Computer Communications*, vol.16, no.4, p. 229-40. April 1993.
21. D.D. Clark, D.L. Tennenhouse. Architectural considerations for a new generation of protocols. *Computer Communication Review*, vol.20, no.4, p. 200-8. Sept. 1990.
22. R. Macklem. Lessons learned tuning the 4.3 BSD Reno implementation of the NFS protocol. *Proceedings of the Winter 1991 USENIX Conference*, p. ix+363, 53-64. 1991.
23. H. Hennessy, D. Patterson. *Computer Architecture: A Quantitative Approach*. 1990, Morgan Kaufmann.
24. The Dragon Book.
25. M.T. Rose, J.P. Onions, C.J. Robbins. *The ISO Development Environment*.
26. ATM Forum. *ATM User-Network Interface Specification version 3.0*. 1993, Prentice Hall.
27. ISO standard 8824. Specification of Abstract Syntax Notation One (ASN.1), 1988.
28. ISO standard 8825. Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), 1988.
29. Sun Microsystems. *SunATM-155 SBus Cards Manual*. 1995.
30. Digital Equipment Corp. DEC 3000 Model 400/400S AXP Technical Summary. Order number EC-N0093-51.
31. Kaman Sciences Corporation. *Vince 1.0: Application Programmer's Interface (API) Manual*. 1995.
32. A. Srivastava, A. Eustace. ATOM: A system for building customized program analysis tools. *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 196-205, June 1994.
33. P. Hoschka, C. Huitema. Automatic Generation of Optimized Code for Marshalling Routines. *IFIP TC6/WG6.5 Internation Working Conference on Upper Layer Protocols, Architectures and Applications*, Barcelona. Amsterdam: North Holland.