Performance comparison of ASN.1 encoder/decoders using FTAM

Murat Bilgic* and Behcet Sarikaya†

Abstract Syntax Notation-One (ASN.1) is a standard external data representation language used to define messages of application layer protocols. Its encoding rules, the Basic Encoding Rules (BER), are also international standards that define the encoding/decoding of data values into/from a transfer syntax. Various approaches to automating BER encoding/decoding are examined; in particular, two widely used software packages (ISODE and CASN1) are studied. A hardware BER encoder/decoder called VASN1 is presented. Performance of software and hardware approaches are evaluated on real instances of file transfer using a standard FTAM protocol. Benchmarks obtained from running CASN1 on one of the fastest workstations and from running VHDL simulations of VASN1 indicate the superiority of the hardware approach.

Keywords: ASN.1, Basic Encoding Rules, ISODE, CASN1, VASN1, benchmarks, FTAM

Open systems interconnection (OSI) protocol standards have been developed to achieve the interconnection of systems from different vendors. However, there is growing concern as to whether the performance of implementations of these protocols can be good enough to meet end-to-end delay and throughput requirements. With advances in high-speed networks, a large difference between the performance at the application layer and the signalling rate of networks is becoming more apparent, since performance is mainly limited by the speed of computers which process the incoming and outgoing messages[1]. There have been different approaches to identify the possible locations of the bottleneck. Some researchers suggest that protocols are to be blamed, and new protocols that can be computed by hardware in special chips should be developed[2]. Others claim that the implementations of the protocols are to be blamed, and it is possible to obtain high throughput with more efficient implementation[3]. It should also be noted that much of the continuing research is located at the transport layer and below. However, especially for OSI application services, a large part of the execution time is spent in the layers above the transport layer[4]. One of the major contributions of this part is the data encoding/decoding, which is a natural requirement of heterogeneous networks.

The lower five layers of the OSI model deal with the movement of bits from source to destination, whereas the functionality of the sixth layer – the presentation layer – is to preserve the meaning of the information exchanged. Since the main motivation behind the OSI model is to achieve reliable communication in a heterogeneous environment, where different computers, different operating systems, etc., are involved, the model should provide some mechanism to convert the machine-dependent data structures into a bit stream suitable for transfer by the lower layers, and then to decode it to the required representation at the destination.

To solve the problem of representing, encoding, decoding and transferring complex data structures, a standard notation called Abstract Syntax Notation-One (ASN.1)[5] has been defined. Along with the notation, a set of encoding rules called Basic Encoding Rules (BER)[6], used to perform conversion between the data values and the transmitted bit stream, has been introduced. The format of the bit stream is called the *transfer syntax*.

In this paper, we compare the performance of our VLSI-based ASN.1 (VASN1) encoder/decoder with that of two different software-based implementations on Protocol Data Units (PDU) exchanged between two File Transfer Access Management (FTAM) entities.

*Concordia University, Department of Electrical and Computer Engineering, 1455 de Maisonneuve W. #915, Montreal, Quebec, Canada H3G 1M8
†Computer Engineering and Information Sciences Department, Bilkent University, Bilkent, Ankara 06533, Turkey (email: sarikaya @trbilun.bitnet)
Paper received: 30 September 1991; revised paper received: 4 December 1991

0140-3664/93/040229-12

ASN.1

Data structures

In the OSI stack, the nature of the data exchanged between two communicating entities substantially changes when the session layer–presentation layer boundary is crossed. In session and lower layers, PDUs are specified informally with the help of illustrations. These PDUs are considered as a flat sequence of octets. Presentation and application PDUs (PPDUs and APDUs) necessitate a more formal method for describing data structures. Because of this necessity, ASN.1 is proposed to describe the semantics of PDUs independent of the particular programming language, compiler or operating system being used on any node of the heterogeneous network. Since such networks include different types of machines, the representation of simple data (e.g. integer, boolean, etc.) should also be standardized to enable such machines to communicate. A common set of encoding rules understood by all the nodes of a heterogeneous network must be provided to convert the values of PDUs in their local format to/from a transmitted bit stream.

Application layer entities, unlike other OSI layer entities, are not self-contained entities; instead, they are collections of Application Service Elements (ASE), and Common ASEs (CASE). Each ASE cooperates with its peer using a specific protocol, and the set of ASEs within the application layer entity is determined by the application context. An application service is provided by a dynamic stack of ASEs and a presentation layer entity. One example is a FTAM Service Element, which uses the Association Control Service Element (ACSE) only for setting up and finishing an association. After association, FTAM ASE directly connects to the presentation layer entity.

Although ASN.1 encoding/decoding is conceptually associated with the presentation layer, an ASN.1 encoder/decoder can be implemented so that it provides an ASN.1 encoding/decoding service to ASEs, CASEs, and the presentation layer itself, as shown in *Figure 1b*. Such a *layerless* ASN.1 encoder/decoder can be implemented with special hardware support to provide a much faster encoding/decoding service, which is particularly required in high-speed networks.

Abstract syntax

Basic concepts of ASN.1 are *types* and *values*. A *subtype* is a type which is a subset of another type. ASN.1 provides a number of *built-in types* as well as a number of tools with which *constructed types* can be defined. There are two kinds of tool: *type constructors*, which are used to define types that include values of other types; and *subtype constructors*, which are used to define types that include only a subset of the values of another, parent type. The types and constructor tools of ASN.1 are listed in *Table 1*. ASN.1 is a case sensitive language with most of its keywords (BOOLEAN, ENUMERATED) written in capitals.

Simple built-in types are notationally integral to ASN.1, whereas *useful* built-in types are defined by means of type constructors. Some of the simple built-in types (e.g. BOOLEAN, INTEGER, REAL) are similar to those found in any programming language. BIT STRING and OCTET STRING are variable length strings of bits and bytes, respectively. The NULL type comprises a single value NULL; the OBJECT IDENTIFIER type is used to name standard and user-defined information object classes. There are several CHARACTER STRING types which comprise the ordered sequence of variable numbers of characters. The ANY type can be considered as the union of all types. GENERALIZED TIME and UNIVERSAL TIME types are used to identify time points. The EXTERNAL type constitutes the instances of information object classes, and the OBJECT DESCRIPTOR type is used to show the textual descriptions of those classes.

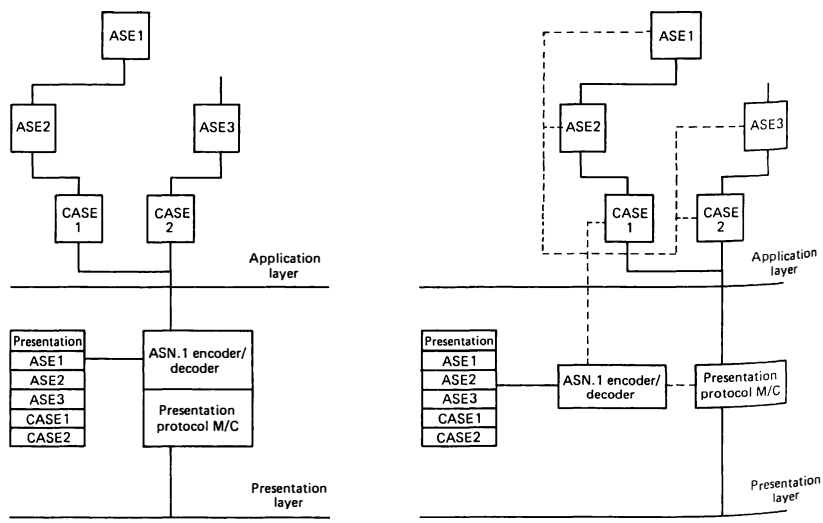

Figure 1 Structure of Application and Presentation layers. (a) ASN.1 encoder/decoder inside the Presentation Layer: (b) ASN.1 encoder/decoder outside the Presentation Layer

Table 1 ASN.1 types and constructor tools

Built-in types		Type constructors	Subtype constructors
Simple types	Useful types		
BOOLEAN	GENERALIZED TIME	ENUMERATED	SINGLE VALUE
INTEGER	UNIVERSAL TIME	SEQUENCE	CONTAINED SUBTYPE
BIT STRING	EXTERNAL	SEQUENCE OF	VALUE RANGE
OCTET STRING	OBJECT DESCRIPTOR	SET	SIZE CONSTRAINT
NULL		SET OF	PERMITTED ALPHABET
OBJECT IDENTIFIER		CHOICE	INNER SUBTYPING
REAL		TAGGED	
CHARACTER STRING			
ANY			

An example ASN.1 module is given in *Figure 2* to better describe the use of ASN.1.

The ENUMERATED type constructor is used to define named integers; the SEQUENCE type constructor is used to define a type such as Type A of *Figure 2,* whose values are ordered collections of values (e.g. first and second are similar to record structures in programming languages); the SET type constructor is used when the order of components of a record is not important. A component of SET or SEQUENCE, such as the component first of Type A, may be declared as OPTIONAL whose value need not be present in the record, or as DEFAULT, with a default value as in the case of component s2 of Type C, whose default value is 1. The SEQUENCE OF and SET OF type constructors are used to define types, e.g. Exp-PDU, Type B of *Figure 2,* whose values are collections of homogeneous values similar to arrays in programming languages. The CHOICE type constructor is used to define a type that is the union of one or more alternative types with distinct tags similar to variant records. The TAGGED type constructor is used to define a type that differs from a specified subject type only by its tag.

Each ASN.1 type is associated with a *tag.* The tag mechanism is used to provide a basis for distinguishing values of one type from those of others (as well as the TAGGED type constructor). A tag has two parts; its class and number. A tag's class specifies the domain of its number. UNIVERSAL tags are defined exclusively in the ASN.1 standard for types such as Type A, which is of the SEQUENCE type and has a tag UNIVERSAL 16. APPLICATION tags are defined for each ASN.1 module which is a named package for related definitions of types and values. PRIVATE tags are for user-defined data types, and CONTEXT-SPECIFIC tags vary from one context to another formed by the alternative CHOICE type, or the element type of a SEQUENCE or SET type. As an example, in *Figure 2* the components first and second constitute different contexts within Type A, and they differ with their tag numbers.

```
EXAMPLE DEFINITIONS ::=
BEGIN
  Exp-PDU ::= [APPLICATION 0] IMPLICIT SEQUENCE OF TypeA
  TypeA ::= SEQUENCE {first    [0] IMPLICIT TypeB OPTIONAL,
                      second [1] IMPLICIT TypeC}
  TypeB ::= SET SIZE (2) OF INTEGER
  TypeC ::= SEQUENCE {s1 IA5String,
                      s2 INTEGER DEFAULT 1}
END
```

Figure 2 Example ASN.1 module

Transfer syntax

Encoding rules define a transfer syntax for values, allowing them to be exchanged between open systems. Different transfer syntaxes (in turn, different encoding rules) may be needed at different times for the same abstract syntax, for different purposes.

BER are currently the only standard encoding rules. They provide a transfer syntax where encoding of every value has three parts: the identifier, length, and contents octets. If length octets use a specific format, called *indefinite,* then another part, called end-of-contents, succeeds the contents octets.

The *identifier* octets encode the value's tag and form. They take either the *short* form, comprising a single octet for types with tag numbers up to 30, or the *long* form, comprising two or more octets for types with tag numbers 31 and greater, as shown in *Figure 3.*

The *length* octets, which indicate how the final octet of the encoding is located, take one of three forms: short, long, or indefinite. The *short* form, comprising a single octet, is used to encode the number of contents octets, which is up to 127. The *long* form consists of two or more octets, where the first octet shows the number of succeeding length octets that encode the number of contents octets up to $2^{1008} - 1$. The *indefinite* form, using a single, fixed octet indicating the presence of end-of-contents-octets (two '00'H octets), may only be used for a constructed form of the contents octets (see *Figure 4*).

The *contents* octets take either primitive or constructed form. The *primitive* form consists of zero or more octets whose meaning depends on the type of the encoded value. The *constructed* form consists of the encodings of zero or more other values whose meaning depends on the type being encoded.

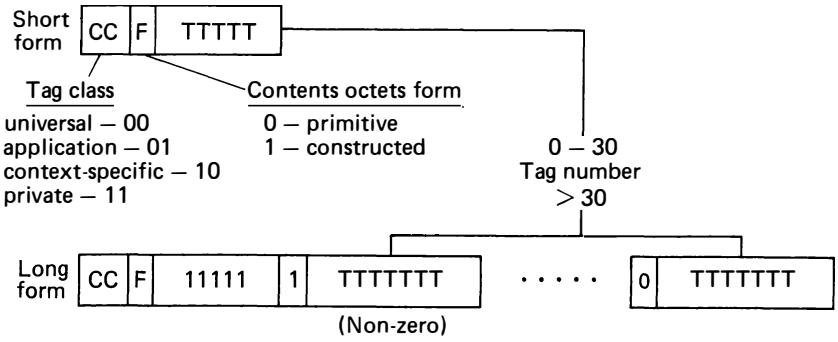

Figure 3 Identifier octets in BER

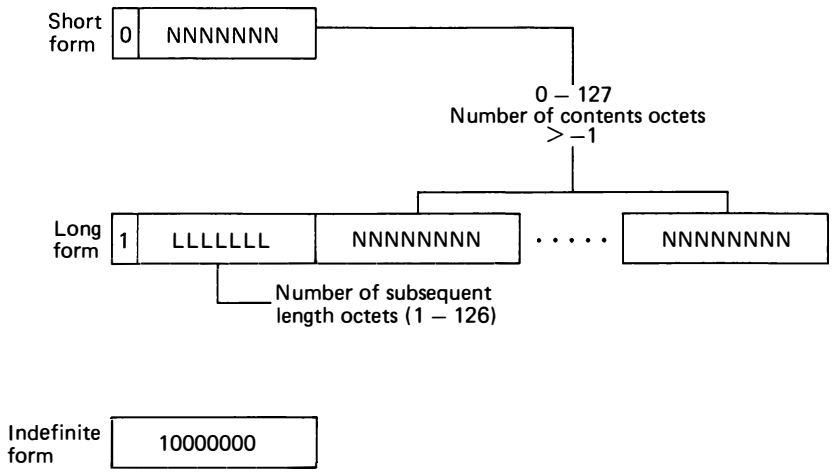

Figure 4 Length octets in BER

For the BOOLEAN, INTEGER, REAL, OBJECT IDENTIFIER, NULL and ENUMERATED types, BER define only a primitive form of the contents octets; whereas for the SEQUENCE, SEQUENCE OF, SET, SET OF and EXTERNAL types, only a constructed form of the contents octets is defined. BER define both primitive and constructed forms of the contents octets for BIT STRING, OCTET STRING, GENERALIZED TIME and UNIVERSAL TIME, as well as CHARACTER STRING and OBJECT DESCRIPTOR types. The form of the contents octets for TAGGED, CHOICE and ANY types depends upon the form of their subject types.

An example value for the type Exp-PDU given in *Figure 2,* and its transfer syntaxes according to BER, are given in *Figure 5.*

```
{{first    {1,2},
  second {s1 "string", s2 1}}
    Exp-PDU Length Contents
        60   17
          Sequence Length Contents
          30       15
                   first    Length Contents
                   A0       06
                            Integer Length Contents
                            02      01      01
                            Integer Length Contents
                            02      01      02
                   second   Length
                   A1       0B
                            IA5String Length Contents
                            16        06      "string"
                            Integer   Length  Contents
                            02        01      01
```

Figure 5 Value of Exp-PDU and its transfer syntax

SOFTWARE IMPLEMENTATIONS

Several software tools such as compilers[7], libraries inside communication software packages[8] and translators[9] have been developed to support ASN.1. In this section we discuss two of them.

CASN1

CASN1 is an ASN.1–C compiler, along with an implementation of BER, called the ED library, which has been developed at the University of British Columbia[7]. The approach of CASN1 is the most direct approach to encoding/decoding data with no type information embedded. It provides the application a comprehensive set of procedures to encode/decode every elementary item in the abstract syntax. The primary design goal in this project is to obtain efficient encoder/decoders by eliminating the use of an intermediate form, and by building a specialized memory management system.

The ED library contains four classes of routines. Primitive class routines are the encoding/decoding routines for ASN.1 types BOOLEAN, INTEGER, BIT STRING, OCTET STRING, NULL, OBJECT IDENTIFIER, UNIVERSAL TIME, GENERALIZED TIME and EXTERNAL. Constructor class routines are specific routines to encode/decode *struct_beg* and *struct_end,* which enclave the encode/decode routines for components of the constructed type. Utility class routines are used to encode/decode tags, end-of-contents, manipulate the tags, move value from files into main memory, and serialize an IDX link list, whose structure is given in *Figure 6,* and generate a flat octet sequence. IDX structure is used to store the output of each encoding routine. This serialization step usually takes most of the total encoding time when long PDUs are processed. Memory class routines are used to manage the ED library sub-memory system.

ISODE

The ISO Development Environment (ISODE) is a software package developed by the Wollongong Group and the Northrop Corporation for OSI-based applications[8]. As part of its structure, the ISODE contains a set of ASN.1 tools and libraries. The main design objective of this project is to provide BER encoding/decoding to different ASEs existing in the ISODE.

It has a library called 'libpsap' which implements presentation layer abstract-syntax for the machine independent exchange of data structures. It uses two objects, presentation-elements (PE) (whose structure is shown in *Figure 7*) and presentation-streams (PS). PE is an internal form used to represent ASN.1 objects in a machine-independent form. There are several library routines which convert a PE into machine-dependent types in the C language. PS is an object used to represent an I/O path of a PE, such as a communication port or a file pointer.

Encoding/decoding routines are produced by the *pepy* program from the augmented ASN.1 specifications, which are produced by another program called *posy* from the original ASN.1 specifications. The *posy* program also produces a set of C structures for the corresponding ASN.1 types.

The output octet sequence of each encoding routine is stored in a PE whose structure is suitable to embed PDUs from different layers and/or ASEs. Similar to

```
typedef unsigned char byte;
typedef struct IDX {
        byte        *buf;  /*pointer to an octet string*/
        long         len;  /*length of the octet string buf*/
        struct IDX  *next; /*pointer tonext IDX node*/
} IDX, *ptrIDX;
```

Figure 6 C structure IDX definition

```
typedef u_char     PElementClass;
typedef u_char     PElementForm;
typedef u_short    PElementID;
typedef int        PElementLen;
typedef u_char     byte, *PElementData;
typedef struct PElement {
  int              pe_errno;
  int              pe_context;
  PElementClass    pe_class;
  PElementForm     pe_form;
  PElementID       pe_id;
  PElementLen      pe_len;
  PElementLen      pe_ilen;
  union {
    PElementData   un_pe_prim;  /*PRIMitive value*/
    struct PElement *un_pe_cons; /*CONStructor head*/
  }                pe_un1;
  union {
    int            un_pe_cardinal; /*cardinality of list*/
    int            un_pe_nbits     /*number of bits in string*/
  }                pe_un2;
  int              pe_inline;  /*for "ultra-efficient" PElements*/
  char            *pe_realbase; /* . . */
  int              pe_offset;  /*offset of element in sequence*/
  struct PElement *pe_next;
  int              pe_refcnt;  /*hack for ANYs in pepy*/
} PElement, *PE;
```

Figure 7 C structure PE definition

CASN1, PE needs to be serialized into a continuous octet string and this step usually takes most of the time for encoding long PDUs.

VASN1

The basic idea behind the design of a VLSI-based ASN.1 (VASN1) encoder/decoder is to achieve fast encoding/decoding by mapping concurrent algorithms developed within the same project to a specialized hardware. Since an ASN.1 encoder/decoder is expected to function in a heterogeneous environment (e.g. different abstract and/or transfer syntaxes, different types of host machines, etc.), it should be flexible enough to accommodate changes. Since VLSI is selected as the implementation medium, the design should be regular enough such that it can be constructed using a number of basic building blocks.

Model

As shown in *Figure 8,* VASN1 is composed of several modules. Parser and assembler modules perform the conversion between the BER encoded octet string and the Value Descriptions in an intermediate format. The decoder module takes these Value Descriptions and the static Type Descriptions for the current abstract syntax and generates the generic format Value Descriptions. In this form, the incoming octet sequences are

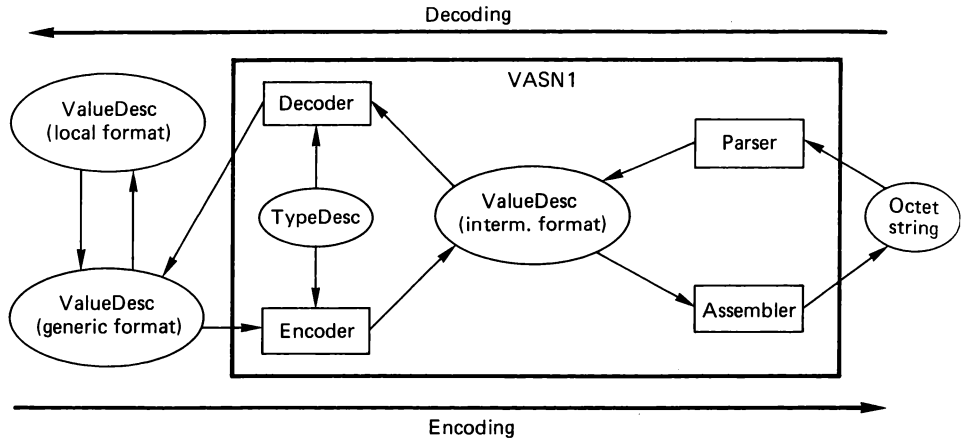

Figure 8 Conceptual model for ASN.1 encoding/decoding

replaced with the values of corresponding types. The generation of the Value Descriptions in the desired local format, (e.g. C, Pascal data structures) is left as a user option, depending on the environment. The encoding process is the dual of the decoding process. The static Type Descriptions for each abstract syntax are produced from their ASN.1 definitions by a software tool developed from the CASN1 compiler.

Architecture

VASN1 is designed to be implemented with VLSI technology. To reach a modular and simple design, different functional blocks are implemented on the same type of module. Basically, the parser and assembler pair and the encoder and decoder pair are implemented on two identical modules, as shown in *Figure 9*. The two modules and an intermediate buffer between them are connected through a local bus. The parser/assembler and encoder/decoder modules are also connected to the system bus to communicate with the master host, and to input from/output to the main memory.

To reach a regular, simple and efficient design, operating system functionalities are excluded from the execution units (EU) of the basic module. Instead, a processor is assigned with the task of distributing the load among the EUs. However, regularity is preserved, since both the central controller (CC) and EUs are implemented using the same RISC processor. There is no communication between the EUs, but the CC communicates with the EUs through the local module bus in terms of messages (see *Figure 10*). Another RISC called the interface controller (IC) is used to serve the execution units for memory read/write operations. The same module bus is used for communication between the EUs and the CC and IC.

The messages originated in the EUs are put into I/O registers, and the destination processor is signalled via an interrupt to complete the transfer by reading the I/O registers. Then the EU resumes its operation until a synchronization point comes in the program, in which case it simply polls the I/O register for the incoming message from the source processor. The communication in this direction is initialized by either the CC or the IC, which writes to the I/O registers of the destination EU. The CC and IC share the bus between them through an arbiter.

The RISC processor and other satellite units (e.g. I/O register, bus arbiters, local memory modules for RISCs) are developed using 1.2-micron CMOS technology. The RISC processor is a 16 bit machine whose instruction set covers 16 instructions. All the instructions

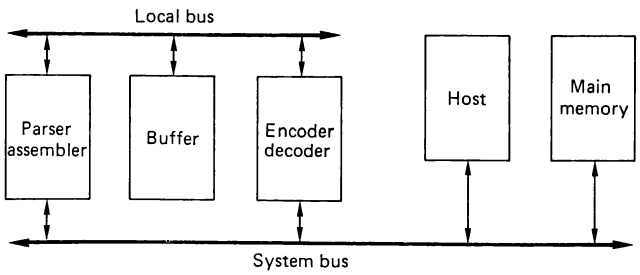

Figure 9 Organization of basic modules

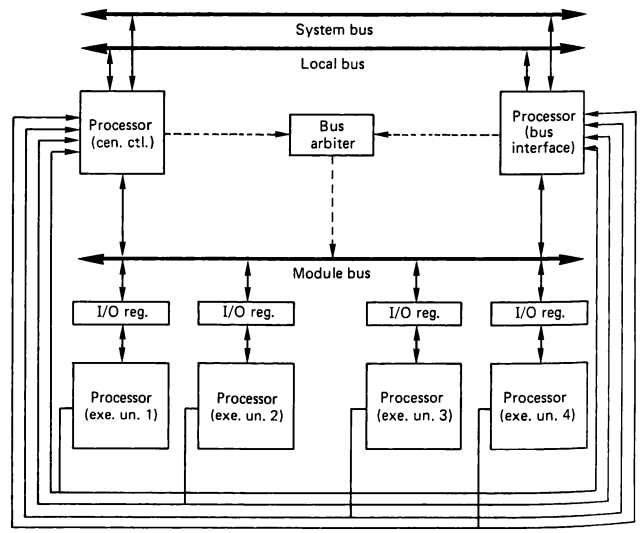

Figure 10 Module with four EUs

except those related to memory transfer take 1 cycle. The clock rate of the RISC is chosen to be 8 MHz (divided into four subcycles). All the instructions except store, load, jump, call and return are pipelined, increasing the effective clock speed to around 32 MHz. Each processor has local RAM and ROM of 2 K each, as well as 16 registers. The local ROM is used as the instruction memory for each processor.

VASN1 is designed with each module containing six RISCs running either parser/assembler or encoder/decoder software packaged into a single chip. The two modules are connected to standard memory units, which constitute the last component of the VASN1 system. The VASN1 system is designed to be embedded into any host system[10].

VASN1 software

The following sections briefly present the algorithms used in VASN1.

Parsing phase

The most important characteristic of BER encoded strings is that they can be separated into or constructed from the identifier, length and contents octets independently of the abstract syntax. In other words, a given BER encoded string w can be converted into a *value tree* T_v whose intermediate nodes contain only the identifier and length octets, whereas the leaf nodes also contain the contents octets for the primitive encoded types.

An example input to the parser is shown in *Figure 5*. The output of the parser is the value tree shown in *Figure 11*.

Now let us describe the possible degree of concurrency in the conversion of a BER encoded string into its value tree. According to BER, a given string w can be written as $w_1 w_2 \ldots w_n$, where:

$$w_i = \begin{cases} id_i.len_i & \text{if } id_i \text{ is constructed} \\ id_i.len_i.cnt_i & \text{otherwise} \end{cases}$$

Then parsing of w becomes such that $P(w) = P(w_1)\ P(w_2) \ldots P(w_n)$, where $P(w_i)$'s take place in the execution units of the parsing/assembling module. Parsing is initiated by the host, which sends a request message to the central controller of the parsing/assembling module. The central controller initiates the execution unit-1 to process the substring w_1. For the given example, execution unit-1 performs the parsing of w_1, w_2 and w_3; and when it finishes $P(w_3)$, it sends the address of w_6 to the central controller, since the length of $|w_4 w_5|$ is obtained during $P(w_3)$. Then, execution unit-1 resumes the parsing of w_4 and w_5, whereas execution unit-2, initiated by the central controller, performs the parsing of w_6, w_7 and w_8. Parsing finishes when all the execution units become idle again.

Decoding phase

Decoding of BER-encoded ASN.1 values is performed while each node of the Value Descriptions is linked to a corresponding node of the Type Descriptions for the current abstract syntax. Type Descriptions are directly produced from the given ASN.1 source text in a tree-like format. The nodes of the Type Descriptions are 4-tuples $\langle id, impl, flags, extinf \rangle$, where *id* constitutes the class, encoding form, and the tag number, *impl* constitutes the id of its implicit type for IMPLICIT defined types, *flags* are set for OPTIONAL or DEFAULT components, and CHOICE, and repetitive types (e.g. SEQUENCE OF, SET OF), and *extinf* stores a pointer to an extra-information table that stores information such as default and enumerated values, or constraints due to subtyping. To better describe the structure of the resulting Type Descriptions, that of Exp-PDU given in *Figure 2* is shown in *Figure 12*.

During the decoding operation, the value tree is traversed according to the structure of its Type Descriptions. For each node of the value tree, there must be a corresponding node in its Type Descriptions. ASN.1 includes provisions such as: CHOICE, ANY, which necessitate decisions to select alternatives; OPTIONAL, DEFAULT, which necessitate decisions to omit or include such a component; and SEQUENCE OF and SET OF type constructs, which necessitate decisions to determine the end of a repeating type. Therefore, traversal of a value tree is realized by finding the matching node of its Type Descriptions for each

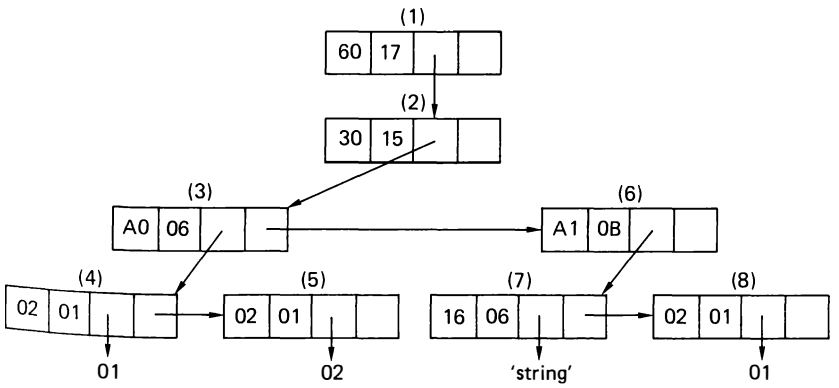

Figure 11 Value tree of Exp-PDU - parsing output

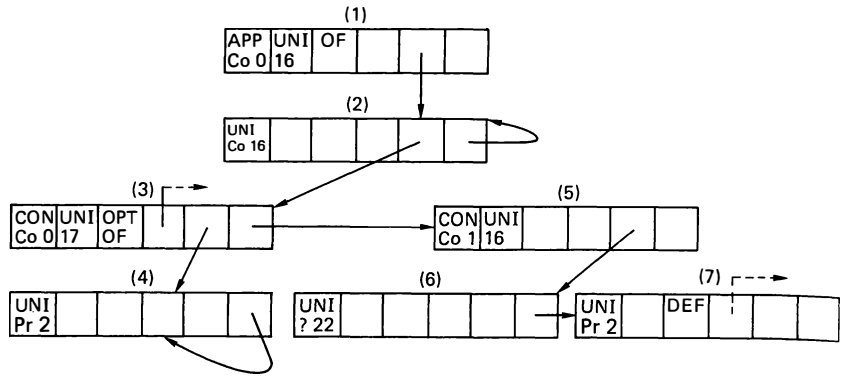

Figure 12 Type descriptions of Exp-PDU

node. This matching is performed by comparing the id parts of nodes of the value tree and Type Descriptions. The traversal starts with the root nodes of value tree T_v and Type Descriptions G_t; and it results in another tree T_r, whose nodes are produced from the fields of the matching nodes of the value tree and Type Descriptions. The resulting tree of the traversal on T_v given in *Figure 11* is shown in *Figure 13*. (To make the figure simpler, instead of their individual fields, node numbers for nodes of T_v and G_t are given.)

As can be seen from *Figure 13*, the nodes of T_r for primitive types such as BOOLEAN, INTEGER, OBJECT IDENTIFIER, etc., contain the decoded form of incoming octet strings, whereas the nodes for constructors such as SEQUENCE, SET, etc., have only fields of nodes of T_v and G_t.

Now, let us describe the possible degree of concurrency in the decoding process. Both T_v and G_t, and recursively $T_{v,i}$'s, $G_{t,j}$'s, can be written as $n_{v,\text{root}}\, T_{v,1}\, T_{v,2} \ldots T_{v,p}$ and $n_{t,\text{root}}\, G_{t,1}\, G_{t,2} \ldots G_{t,q}$. Then decoding of T_v becomes such that $D(T_v, G_t) = D(n_{v,\text{root}}, G_t) D(T_{v,1}, G_t) D(T_{v,2}, G_t) \ldots D(T_{v,p}, G_t)$, where $D(T_{v,i}, G_t)$'s take place in the execution units of the encoding/decoding module. Decoding is initiated by the central controller of the parser/assembler module, which sends a request message to the central controller of the encoding/decoding module. The central controller initiates the execution unit-1 to process the node $n_{v,\text{root}}$. For the given example, decoding of nodes 1 to 3 of T_v in *Figure 11* is performed by the execution unit-1. Once the decoding of $n_{v,3}$ is finished, $n_{v,6}$ and $n_{t,5}$ of *Figure 12* are sent to the central controller. Then execution unit-1 resumes the decoding of $n_{v,4}$ and $n_{v,5}$, whereas execution unit-2 initiated by the central controller performs the decoding of $n_{v,6}$, $n_{v,7}$ and $n_{v,8}$.

Encoding and assembling phases

The encoding operation is basically the inverse of decoding, where the inputs are T_r and G_t and the output becomes T_v.

Similar to the duality between decoding and encoding, there exists a duality between parsing and assembling in which the output flat octet sequence is generated from the value tree T_v. This step is the same as the serialization of IDX in CASN1, or the conversion of PE into PS in ISODE.

CC and IC software

CC units perform all the operating system functionalities of the system such as distribution of the processing load for each phase of encoding/decoding among EUs, message-based communication between modules and with the host machine, as well as management of the two-directional pipeline. The CC software enables the processing of multiple PDUs at the same module by appending a PDU identifier to messages transferred to and from the EUs.

The maximum number of PDUs which can be active in a module is equal to the number of EUs, which is four for the current model. To balance the load of the module's EUs, the CC distributes the processing load of each active PDU in a different order.

IC units facilitate the memory transfer between EUs and the memory modules. They segment the shared-

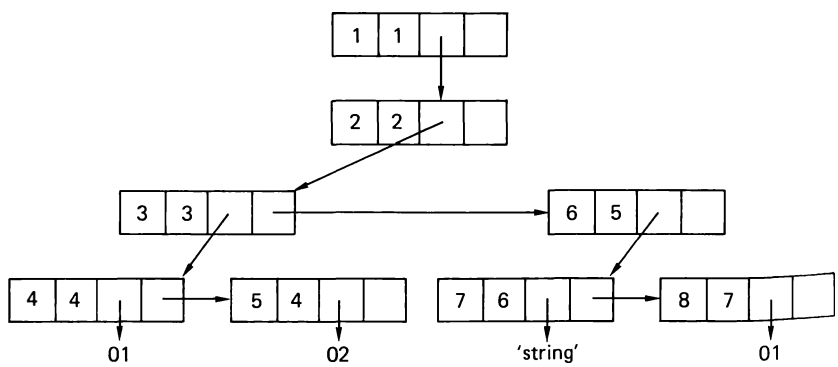

Figure 13 Value tree T_r of Exp-PDU - decoding output

memory into pages which are assigned to different PDUs in the pipeline. These PDUs are distinguished by their unique identifiers.

PERFORMANCE EVALUATION

The simplest approach to evaluate the performance of an ASN.1 encoder/decoder is to use different types and sizes of value trees as data, and measure the time taken for encoding/decoding. The number of levels of the value tree and the number of leaves at a level can be used as parameters of the measurements. Such an approach is used by Nakawaji *et al.*[11] to evaluate the performance of a software-based ASN.1 tool. However, a more realistic analysis can only be performed with real-time data.

To evaluate the performance of the encoders/decoders introduced, the PDUs transferred during the use of the FTAM service of ISODE are used. These PDUs include FTAM PDUs, ACSE PDUs used during the connection establishment and release, as well as Presentation PDUs used to carry these APDUs.

FTAM is an application layer protocol for transferring, accessing and managing files between open systems. FTAM is connection-oriented with a series of embedded regimes: FTAM Association, file selection, file open and data transfer. *Figure 14* shows the protocol stack formed during the FTAM association regime. In this figure, exchanged PDUs are given in parentheses under the used service primitives. Service primitives and PDUs written in italics are used during the association release, whereas the others are used during association establishment. In other regimes, the FTAM ASEs directly communicate with the underlying Presentation entity.

The performance of CASN1, ISODE and VASN1 are measured on FTAM, ACSE and presentation PDUs generated during the use of the FTAM service from the ISODE software distribution package (release 6.0). Two sets of measurements were made: on Sun 3/60 for ISODE and CASN1, and on Sun 4 Sparc 2 for CASN1. The test routines use the *getitimer* provided by the UNIX system to determine the processor time. All performance figures are the mean of 10 measurements; in each the encoding or decoding routine tested is repeated 1000 times.

The performance figures for VASN1 are derived from the simulation of a structural model built using the VHSIC Hardware Design Language (VHDL)[12]. The model includes two modules, each of which is built using six RISCs as a CC, an IC and four EUs, as shown in *Figure 10*. Other components such as queue, arbiter and memory modules also constitute entities of the overall model. Programs of different RISCs used in different phases of encoding/decoding are loaded into processor ROMs as static data. A RAM module is initialized with type table information, whereas PDUs to be encoded/decoded are dynamically loaded into RAM using another RISC which models the host.

Measurements are divided into groups according to the regimes in which the corresponding PDUs are exchanged. The first group involves FTAM, ACSE and Presentation PDUs used during the FTAM association regime. During the connection establishment, F-INITIALIZE-request, AARQ and CP-type PDUs are sent from initiator to responder, which responds with F-INITIALIZE-response, AARE and CPA-type PDUs if it accepts the connection request. Values used for F-INITIALIZE-request and F-INITIALIZE-response are shown in *Figure 15*.

In *Figure 16a,b,* the total execution times for encoding/decoding these PDUs are given in terms of individual encoding/decoding time figures for FTAM, ACSE and presentation PDUs. As can be seen, encoding/decoding of presentation PDUs takes much longer than for FTAM and ACSE PDUs, especially in the case of ISODE, since they include conversion between the PE and the PS. We can also see that the performance of ISODE and CASN1 (both measured on Sun 3/60 workstations) are comparable.

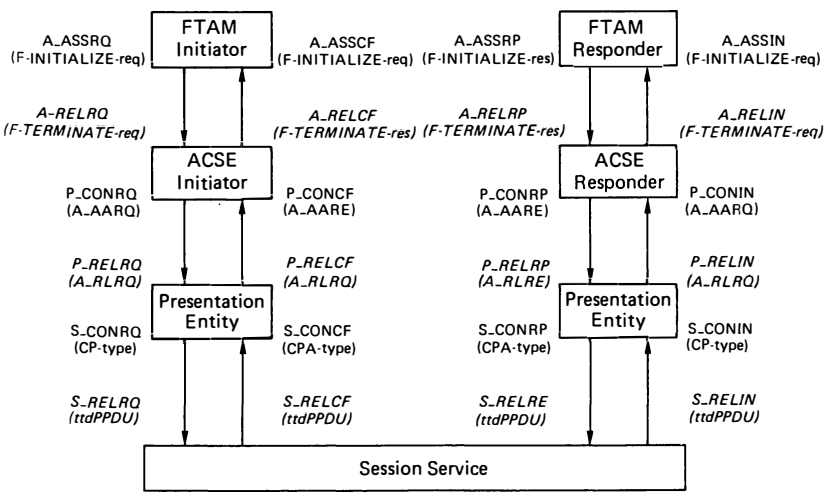

Figure 14 Service primitives and exchanged PDUs during FTAM association

```
{
 ftam-regime-pdu {
  f-initialize-request {
   service-class {management-class, transfer-class,
                  transfer-and-management-class},
   functional-units {read, write, limited-file-management,
                     enhanced-file-management, grouping},
   attribute-groups {storage},
   ftam-quality-of-service no-recovery,
   contents-type-list {1.0.8571.5.3,
                       1.0.8571.5.1,
                       1.3.9999.1.5.9},
   initiator-identity "bilgic",
   filestore-password {binary "mypass1"}
  }
 }
}

{
 ftam-regime-pdu {
  f-initialize-response {
   service-class {transfer-and-management-class},
   functional-units {read, write, limited-file-management,
                     enhanced-file-management, grouping},
   attribute-groups {storage},
   ftam-quality-of-service no-recovery,
   contents-type-list {1.0.8571.5.3,
                       1.0.8571.5.1,
                       1.3.9999.1.5.9}
  }
 }
}
```

Figure 15 Examples of PDU values

In the following benchmark tests (see *Figures 17* and *18*) we compare CASN1 with VASN1. CASN1 measurements were conducted on a Sun4 Sparc 2 workstation, and VASN1 performance figures were obtained from its VHDL model. These benchmark tests were conducted to measure the speed-up arising from concurrent PDU processing and parallelism in VASN1.

The first group of measurements are conducted for the FTAM disconnection phase. To perform the disconnection, the initiator sends F-TERMINATE-request, RLRQ and ttdPPDU to the responder, which replies with F-TERMINATE-response, RLRE and ttdPPDU. *Figure 17* depicts the execution time for encoding/decoding the F-TERMINATE-request, RLRQ and ttdPPDU. The results for reply PDUs are identical.

Figure 17a shows encoding times of 1, 2 and 4 PDU sets (F-TERMINATE-request, RLRQ and ttdPPDU), and *Figure 17b* shows corresponding decoding times. For CASN1, execution times for multiple PDU sets are obtained from those of a single PDU set by simple addition. Since VASN1 modules can process multiple PDUs concurrently, the overall execution times for multiple PDU sets show the exploitation of pipeline.

The other group of measurements are conducted in a data transfer regime. The ISODE FTAM service supports unstructured text, unstructured binary and filedirectory files. During the measurement of encoding/decoding for bulk data transfer, the unstructured text file option is used. The FTAM initiator for bulk data transfer-write and the responder for bulk data transfer-read separate the file into pieces, and then transfer them using the presentation service. *Figures 18a,b* show the performance of CASN1 and VASN1 for the transfer of unstructured text files whose size ranges from 2000–20000 bytes for encoding and decoding, respectively.

For encoding 2000 bytes CASN1 takes 6.91 ms, and VASN1 0.365 ms, a speed-up of 18.9. For encoding 20,000 bytes, while CASN1 takes 64.52 ms, VASN1 finishes the same work in 1.7 ms, increasing the speed-up to 37.95. The performance of VASN1 is far superior to that of CASN1 for decoding, since no data copying

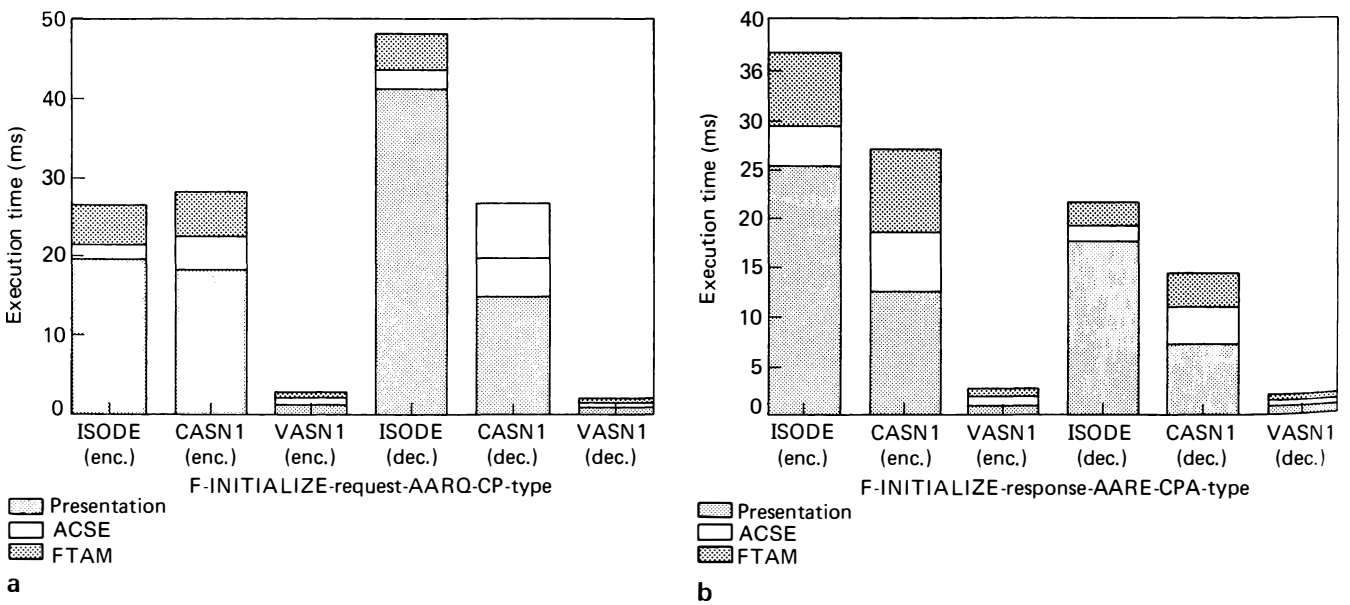

Figure 16 Encoding/decoding time for connection establishment PDUs. (a) Encoding; (b) decoding

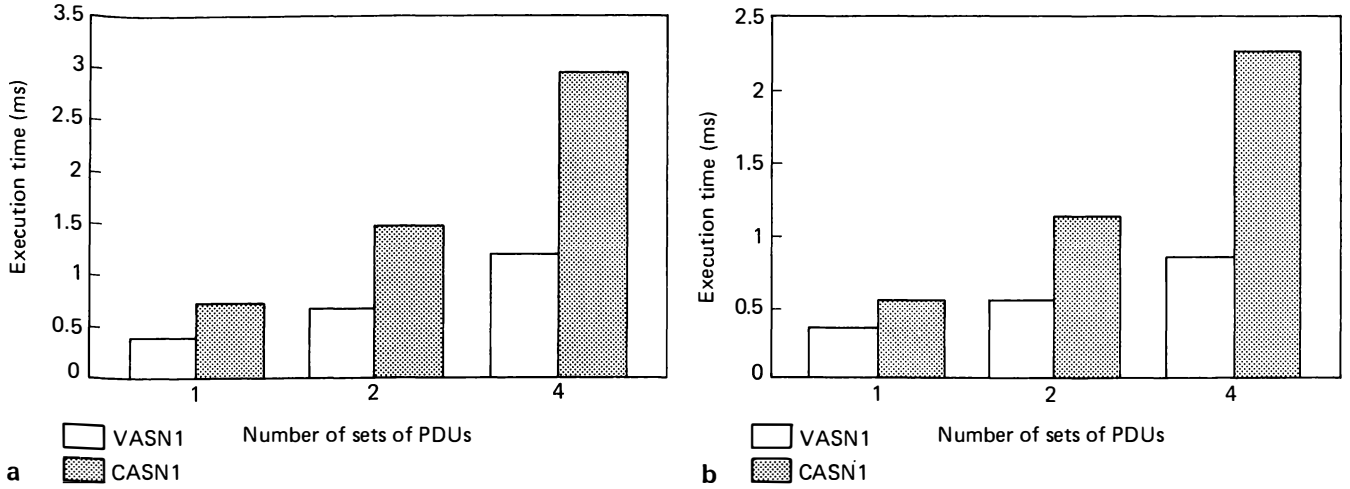

Figure 17 Encoding/decoding time for connection release PDUs. (a) Encoding: (b) decoding

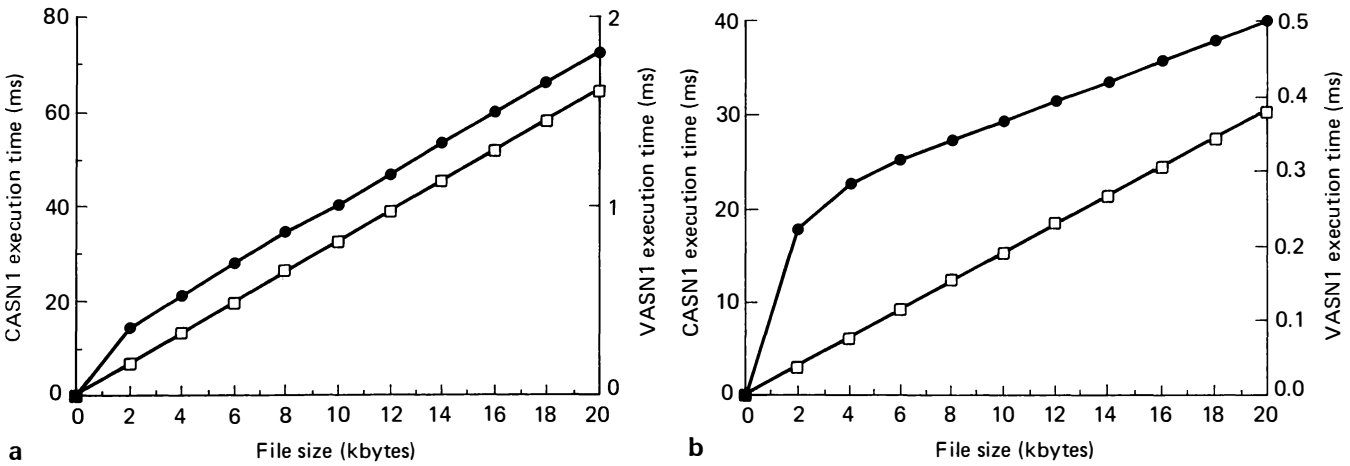

Figure 18 CASN1 and VASN1 encoding/decoding times for bulk data transfer. (a) Encoding: (b) decoding. ●: VASN1: □: CASN1

takes place during either the parsing or the decoding itself. As can be seen from *Figure 18*, VASN1 finishes decoding 2000 bytes in 0.224 ms and CASN1 takes 3.26 ms, therefore VASN1 is 14.55 times faster than CASN1. VASN1 decodes 20,000 bytes in 0.5005 ms and CASN1 decodes in 30.47 ms, a speed-up of 60.88. This dramatic increase in speed-up for decoding is a result of two factors: VASN1 performs no data copying, and its pipe is fully utilized.

CONCLUSION

In this paper, the OSI data representation standard ASN.1, along with the standard encoding rules set BER, are discussed. Two different software systems used to generate ASN.1 encoder/decoder routines, as well as a multiprocessor architecture for ASN.1 encoding/decoding, are discussed. We have presented the performance results for three different ASN.1 encoder/decoders on PDUs transferred during the service of FTAM ASE of an ISODE package. The following conclusions can be drawn:

- The performance of VASN1 is always superior to that of the software-based approaches. VASN1 becomes especially attractive when PDUs from FTAM, ACSE and presentation are embedded and a number of FTAM PDUs are sent as a group.
- The data copying amounts to a large portion of encoding/decoding of bulk data PDUs. The advantage of using pointers instead of copying the data itself in VASN1 becomes more apparent in the decoding.
- Due to its connection-oriented nature, FTAM necessitates a considerable number of PDUs besides the bulk data PDUs. Particularly when only a single file transfer is done for each association, the total encoding/decoding time of these PDUs becomes much larger than that of bulk data PDUs.

Although ASN.1 encoding/decoding contributes to a fair portion of the total execution time, the performance of other functional blocks of the protocol is also the major factor in determining the protocol's performance. It should also be noted that the implementation strategy is another important factor. The use of an encoder/decoder parallel to the host machine, which

includes the other protocol functionalities, is another dimension of concurrency, and in turn may be another source of further performance improvement providing an efficient interface.

Future study will include investigation of the effects of the ASN.1 encoding/decoding on overall protocol performance. This is especially interesting when parallel bulk transfers or parallel associations take place between two FTAM ASEs which will provide VASN1 to use its pipeline and to exploit the encoder/decoder, host parallelism.

The benchmarks reported in this paper were conducted for the file transfer application. It will be interesting to conduct similar benchmark tests for other OSI applications like e-mail, directory services, etc.

REFERENCES

1 **Strayer, W T and Weaver, A C** 'Performance measurement of data transfer services in MAP', *IEEE Network*, Vol 2 No 3 (May 1988) pp 75–81

2 **Chesson, G** 'XTP/PE overview', *Proc. 13th Conf. on Local Computer Networks*, Minneapolis, MN (October 1988) pp 292–296

3 **Clark, D D *et al.*** 'An analysis of TCP processing overhead', *IEEE Commun.*, Vol 27 No 6 (June 1989) pp 23–29

4 **Gunningberg, P *et al.*** 'Application protocols and performance benchmarks', *IEEE Commun.*, Vol 27 No 6 (June 1989) pp 30–36

5 CCITT Recommendation X.208, *Specification of Abstract Syntax Notation One (ASN.1)*, Geneva, Switzerland (1987)

6 CCITT Recommendation X.209, *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, Geneva, Switzerland (1987)

7 **Neufeld, G W and Yang, Y** 'The design and implementation of an ASN.1-C compiler', *IEEE Trans. Softw. Eng.*, Vol 16 No 10 (October 1990) pp 1209–1220

8 ISODE, *The ISO Development Environment: User Manual*, Wollongong Group, Palo Alto, CA

9 **Brady, F, Boshier, A G, Pitt, D and Szcygiel, B M** 'One2One – A tool for translating ASN.1 to ACT ONE', *Proc. FORTE'90*, Madrid, Spain (November 1990)

10 **Bilgic, M** *Concurrent Protocol Data Unit Encoding/Decoding: Algorithms, Architectures and Performance Evaluation*, PhD thesis, Concordia University, Canada (June 1992)

11 **Nakakawaji, T, Katsuyama, K, Miyaushi, N and Mizuno, T** 'Development and evaluation of APRICOT (Tools for Abstract Syntax Notation One)', *Proc. 2nd Int. Symposium on Interoperable Information Systems*, Tokyo, Japan (1988)

12 **Wu, W, Bilgic, M and Sarikaya, B** 'VHDL modelling and synthesis of an ASN1 encoder/decoder', *CCVLSI 90*, Ottawa, Ontario (October 1990) pp 1.5.1–1.5.8