Published in the proceedings of 44th ACM Southeast Conference (ACM SE'06), 2006, pages 726-731

TDX: a High-Performance Table-Driven XML Parser *

Wei Zhang
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530
wzhang@cs.fsu.edu

Robert A. van Engelen
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530
engelen@cs.fsu.edu

ABSTRACT

This paper presents TDX, a table-driven XML parser. TDX combines parsing and validation into one pass to increase the performance of XML-based applications, such as Web services. The TDX approach is based on the observation that context-free grammars can be automatically derived from XML schema. We developed a parser construction tool to automatically construct TDX grammar productions from a schema. Grammar tokens are defined by the specific schema element names, attribute names, and text. Because most of the structural constraints in XML schema are cast as grammar rules, parsing and validation of XML instances are efficiently implemented. The results show that TDX is several times faster than DOM or SAX parsing with validation enabled.

Categories and Subject Descriptors

J.m [**Computer Applications**]: Miscellaneous

General Terms

Performance

Keywords

XML, Web services, high-performance computing, schema-specific parsing, LL(1) grammars

1. INTRODUCTION

Most XML parsers use XML schemas [21], or equivalent Relax NG schemas [5], to validate XML document instances. Validation is typically performed in a separate phase that sits on top of the generic XML parser module. Validation is important, because applications that use the XML data typically require the data to be modeled according to a schema. Applications should also not be burdened by having to verify assertions on the data content that can otherwise be done by a validating parser. However, the current separation of parsing and validation in XML parsers incurs significant overhead and requires frequent access to the schema. This overhead can be eliminated by integrating parsing and structural validation into a schema-specific parser [2, 3, 11, 16, 18].

We integrated parsing and validation using a new table-driven XML parsing approach based on linguistic principles by encoding XML parsing and validation inseparably in context-free grammar rules. We named this approach TDX parsing: table-driven XML parsing. We developed a TDX parse engine, a tokenizer, and a tool to construct TDX parse tables from XML schemas. These parse tables are compact representations of schema component definitions and effectively serve as LL(1) parse tables to drive the TDX parse engine. The tokenizer scans XML element tag names, attribute names, PCDATA text, and namespaces. The engine takes tokenized XML input and verifies the XML structure against the grammar stored in a compact table format.

TDX provides a flexible framework for combining structural, syntactic and semantic constraints in an efficient manner. Efficiency of parsing is increased by almost an order of magnitude compared to popular XML parsers with validation enabled. TDX also offers a high level of modularity, because modular parse tables can be generated for new or updated schemas. Application actions can be embedded as semantic rules in the table to fire actions and/or instructions for translating XML to application data. This increases the robustness for developing high-performance XML Web service applications, because the XML translation logic is integrated.

The remainder of this paper is organized as follows. We first review the related work in Section 2. Then we describe our table-driven XML parsing approach in Section 3. Section 4 describes the automatic construction of a TDX-based parser with our code generator. Performance results are presented in Section 5. Finally, we summarize the paper with conclusions in Section 6.

2. RELATED WORK

The validation of XML instances against a schema is traditionally processed in two stages by checking well-formedness and validity. The first stage is purely syntactic to scan and parse XML documents. The second stage determines whether or not the the structure is a valid instance of a given schema. The separation of stages for parsing, validation, and application processing of an XML instance introduces significant overhead [3, 15].

*Supported in part by NSF grant BDI-0446224 and DOE grant DEFG02-02ER25543

In [16], Van Engelen proposes a method to integrate parsing and validation into a single stage with parsing actions encoded by a finite state automaton (DFA), where the DFA is directly constructed from a schema. The DFA actions perform parsing and validity checks on an XML instance. Because of the limitations of the language described by DFAs, i.e., regular languages, a DFA-based approach does not permit cyclic schemas, i.e., the approach can only by used to process the regular subset of context-free schema languages.

Chiu et al. [3] also suggests an approach to merge all aspects of low-level parsing and validation by constructing a single push-down automaton. However, their approach does not support XML namespaces, which is essential for SOAP compliance. Furthermore, conversion from their non-deterministic automaton (NFA) to a DFA may result in exponentially growing space requirements [1].

In earlier work by Van Engelen on the gSOAP toolkit [17, 18, 19] a schema-specific parsing (SPP) approach was implemented and a compiler tool was developed to generate recursive descent parsers to efficiently parse XML documents defined by a schema. The approach is essentially a form of push-down parsing. However, this approach has the disadvantage of recursive descent parsing, which are code size and function calling overhead.

Also the Packrat parser [7] implements recursive-descent parsing with backtracking. It guarantees linear time parsing, but the main disadvantage is its space requirement, which is directly proportional to the input size. This rules the Packrat parser out as a practical approach for XML parsing, or any other where the document length may in ordinary practice be quite large.

Tree grammars were developed to represent the structure of XML instances [13, 14]. Tree grammars provide an efficient way to encode schema validation constraints, but they cannot be utilized to combine parsing and validation into one parsing stage.

3. TDX PARSING

Our table-driven parsing approach takes advantage of LL(1) parsing for both XML syntax and validation constraints. Most of the structural constraints imposed by schemas can be implemented as grammar rules in an LL(1) grammar. The remaining non-structural semantic validation rules are implemented with semantic actions in the productions.

The architecture of TDX is shown in Figure 1. A TDX-based parser consists of a scanner, a parser, an LL(1) parsing table, a set of LL(1) grammar productions along with associate semantic actions, and a set of tokens. The scanner takes XML instance as input and produces token stream and/or text to the parser. The parser performs verification and validation to the XML instance using the LL(1) parsing table and grammar productions with semantic actions. Therefore, only valid data is passed to the application. Syntax errors indicate schema constraint violations.

3.1 Scanner

The scanner takes XML message as input and breaks it into tokens, where each token is either a tag name, such as element names and attribute names, or character data (CDATA). Tokens of element names are further classified as token of the beginning of an element and token of the ending of an element. Figure 2 lists tokens from an example XML schema.

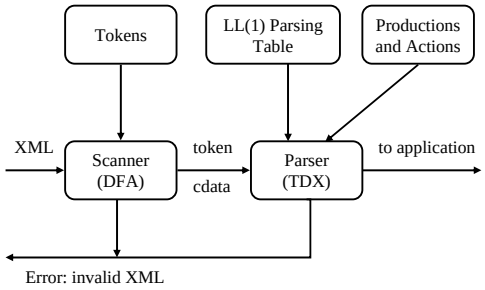

Figure 1: TDX Parsing Architecture

Token	XML Schema
	<schema >
bTITLE	<complexType name="bookType">
eTITLE	<sequence>
bAUTHOR	<element name="title" type="xsd:string"/>
eAUTHOR	<element name="author" type="xsd:string"/>
aISBN	</sequence>
bBOOK	<attribute name="isbn" type="xsd:string">
eBOOK	</complexType>
	<element name="book" type="bookType"/>
	</schema>

Figure 2: Tokens from an Example XML Schema

In this paper, all beginning or opening element tags `<NAME>` are represented by bNAME, ending element names `</NAME>` are represented by eNAME, and attribute tag names are represented by aNAME, Namespace bindings are supported by internal normalization to a standard namespace prefix. Namespace qualified elements and attributes are translated to the corresponding element and attribute tokens. An identical tag name defined under two different namespace domains is in fact represented by two different tokens.

3.2 Parser

The TDX parser consists of an engine that takes the scanner's output and analyzes the XML structure and validation based on the LL(1) parsing table. It also executes embedded semantic actions for further semantic validation and to trigger application events. Semantic actions are implemented with function pointers to action functions. For example, the schema shown in Figure 2 has an equivalent augmented LL(1) grammar shown in Table 1. The TDX engine only uses the table constructed for the LL(1) grammar to parse instances of the schema. The token `DATA` indicates the content of an element or an attribute. It is also used as an indicator to perform semantic validation on text content, i.e., to verify XSD types such as integers and floats.

TDX implements a predictive top-down parsing with one token lookahead for LL(1) parsing. It guarantees a linear

Table 1: LL(1) Grammar for parsing XML instances of Schema in Figure 2

#	Production	Action
1	$t \rightarrow$ bBOOK a b eBOOK	
2	$a \rightarrow$ aISBN DATA	imp_template(s.val)
3	b $\rightarrow b_1$ $b_1{}'$	
4	$b_1 \rightarrow$ bTITLE DATA eTITLE	imp_template(s.val)
5	$b_1{}' \rightarrow$ bAUTHOR DATA eAUTHOR	imp_template(s.val)

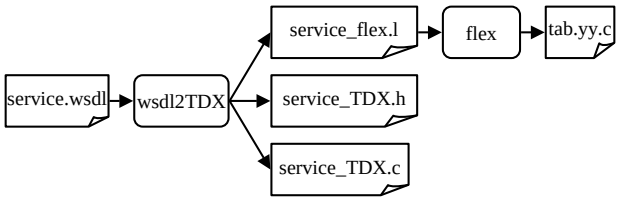

Figure 3: Table-driven XML Parser Generator

time parsing and the space only grows to the occurrence depth. The parsing table, grammar productions and token are automatically constructed from a given set of schemas using a code generator. Section 4 discribes the construction in detail.

4. CONSTRUCTING TDX PARSERS

This section introduces the table-driven parser generator. An overview of the system is given and technical details pertaining to the implementation of the parsing method are discussed.

4.1 Overview

The architecture of the TDX table generator is shown in Figure 3. The generator takes a Web service description (WSDL) or a set of schemas `service.wsdl` and produces a Flex (lex) description `service_flex.l` for the XML scanner. It also generates the TDX source codes `TDX.c` and `TDX.h` for XML parsing and validation. Flex is a frequently used automatic generator tool by compiler developers for high-performance scanners [1, 12]. The `service_flex.l` generated by the generator `wsdl2TDX` is fed into Flex to produce the source code for the XML scanner. The scanner provides token stream for the parser at run time as shown in Figure 1. The generated scanner is specialized to the XML schemas to improve the efficiency.

4.2 Constructing the LL(1) Grammar

The generator maps schema components to LL(1) grammar productions to generate parsing tables. The mapping preserves the structural constraints imposed on XML instances of the schemas. In addition, many types of validation constraints are automatically incorporated in the resulting grammar, such as occurrence constraints, order constraints.

The mapping $\Gamma[\![X]\!]N$ takes a schema component X and a designated non-terminal N and returns a set of LL(1) grammar productions starting with non-terminal N for parsing instances of X. Table 2 lists part of mapping rules for the most common schema components.

This example mapping defines translations for some of most common schema components. All other components such as `choice`, `all`, `group`, and `extention` are implemented in a similar way.

4.3 Generating Tokens for the Scanner

The TDX code generator generates tokens for schema element names and attribute names by assigning a unique number to each token. Note that each token is to be defined by a tag name that is valid in a particular XML namespace, and the default namespace "" is assigned for a tag name without namespace.

$\Gamma[\![$ < simpleType > X < /simpleType> $]\!]$N =$\Gamma[\![$X$]\!]$N
$\Gamma[\![$ < simpleType name='T'> X < /simpleType > $]\!]$N =$\{$T $\rightarrow \epsilon$ $\} \cup \Gamma[\![$X$]\!]$T
$\Gamma[\![$ < element name='E' type='T' / > $]\!]$N =$\{$N $\rightarrow$ bE T eE$\}$
$\Gamma[\![$ < element name='E' type='T' minOccurs='0' /> $]\!]$N =$\{$N $\rightarrow$ bE T eE, N $\rightarrow \epsilon\}$
$\Gamma[\![$ < element name='E' type='T' minOccurs='0' maxOccurs='unbounded' / > $]\!]$N =$\{$N $\rightarrow$ bE T eE N, N $\rightarrow \epsilon\}$
$\Gamma[\![$ < attribute name='A' type='T' use='optional' /> $]\!]$N =$\{$N $\rightarrow$ aA T, N $\rightarrow \epsilon\}$
$\Gamma[\![$ < attribute name='A' type='T' use='required' /> $]\!]$N =$\{$N $\rightarrow$ aA T$\}$
$\Gamma[\![$ < sequence > $X_1 X_2 \ldots X_n$ < /sequence > $]\!]$N =$\{$N $\rightarrow N_1 N_1'$, $N_1' \rightarrow N_2 N_2'$, ... , $N_{n-1}' \rightarrow N_n\} \cup \bigcup_{i=1}^{n} \Gamma [\![X_i]\!] N_i$
$\Gamma[\![$ <complexType name='T'> X </complexType > $]\!]$N = $\Gamma [\![$ X $]\!]$T
$\Gamma[\![$ <complexType> X$A_1 A_2 \ldots A_n$ </complexType>$]\!]$N =$\{$N $\rightarrow N_1 N$, N $\rightarrow N_2 N$, ..., N $\rightarrow N_n N\}$ $\cup \Gamma [\![$ X $]\!]$T $\cup \bigcup_{i=1}^{n} \Gamma [\![A_i]\!] N_i$

Table 2: Example Rules of Mapping Schema Components to LL(1) Grammar

Namespace URIs and tokens are stored in a 1-D array `ns[MAX_SIZE]` and in a 2-D array `token[MAX_SIZE][MAX_SIZE]`, indexed by namespace and name respectively. Both arrays are used to generate tokens for the scanner and the grammar productions in the parsing table. The generated scanner also consults both arrays to break input stream into tokens for the parser. Each tag name is defined as macro. The code generator generates ANSI C codes for the tag names and the namespaces from the schemas. For example, the fragment of source code generated from the schema shown in Figure 2 is listed below.

```
#define TITLE    0
#define AUTHOR   1
#define ISBN     2
#define BOOK     3
  ...

token[0][TITLE]  = 100
token[0][AUTHOR] = 102
token[0][ISBN]   = 104
token[0][BOOK]   = 106
```

Here we assume the default namespace is stored in `ns[0]`. Tokens are stored only once to reduce storage. A token with an even number denotes the token of a beginning element tag. The token of an ending element is represented as the next number (odd) of its corresponding starting token. For example, `bBOOK = 106` and `eBOOK = 107`.

4.4 Constructing the XML Scanner

The code generator constructs a Flex description for scanner based on WSDL or schema documents. This description reads an input stream and breaks it into strings which match the given expressions. The recognition of the expressions is performed by a DFA generated by Flex (For more details on Flex, see [12]). The generated Flex description has the following structure.

```
 1  %{ ... %}
 2  whsp    [ \t\v\n\f\r]
 3  name    [^>/:= \t\v\n\f\r]+
 4  qual    {name}:
 5  open    <
 6  stop    >
 7  skip    [^/>]*
 8  data    [^<]*
 9  xmln    xmlns(:{name}|"")=(\"[^"]*\"|\'[^']*\')
10  attr    =(\"[^"]*\"|\'[^']*\')
11   %x OPEN_ELEM
12   %x CLOSE_ELEM
13   %x ATTS
14   %%
15   {whsp}                    // ignore white space
16   {open}"?"{skip}{stop}     // ignore declaration
17   {open}"!"{skip}{stop}     // ignore comment
18   {open}"/"                 {RESETNS; BEGIN(CLOSE_ELEM);}
19   {open}                    {RESETNS; BEGIN(OPEN_ELEM);}
20
21   <OPEN_ELEM>{whsp}         //ignore white space
22   <OPEN_ELEM>{qual}         {ns = get_ns(yytext);}
23   // ... definitions of XML element names and actions
24   <OPEN_ELEM>{data}         {return DATA;BEGIN(INITIAL);}
25
26   <CLOSE_ELEM>{whsp}         // ignore white space
27   <CLOSE_ELEM>{qual}         {ns = get_ns(yytext);}
28   // ... definitions of XML element names and actions
29   <CLOSE_ELEM>{stop}         {BEGIN(INITIAL);}
30
31   <ATTS>{whsp}              // ignore white space
32   <ATTS>{qual}              ns = get_ns(yytext);
33   // ... definitions of XML attribute name and actions
34   <ATTS>attr                {return DATA; RESETNS;}
35   <ATTS>{stop}              BEGIN(OPEN_ELEM);
36   <*><<EOF>>                return EOF;
37   %%
```

This flex specification extracts all the element names and attributes, and return a stream of tokens. The lines 15-17 ignores white space, declaration and comment.The `qual` regular expression is used to extract the name space for each element name and attribute name (line 22, line 27 and line 32). Names may or may not be qualified. Therefore the variable `ns` must be reset before starting to match a name each time. The function `get_ns(char *)` returns an integer representing the namespace. The function `ins_ns(char *,char *)` takes a namespace name and its content to map the namespace to an integer by accessing the global array `ns[MAX_NS_SIZE]`. The unique root element contains namespaces. The namespace regulation expression and installation are added to the specification by the code generator when the root element is determined.

Schema-specific regular expressions and actions are added to the description. This includes all the element and attribute names found in the WSDL or schemas. These element definitions are collected from all parts of a set of related schemas, including top-level element schema components and local elements.

For example, suppose a schema contains following components:

```
<complexType name="bookType">
  <sequence>
     <element name="title" type="string"/>
     <element name="author" type="string"/>
     <attribute name="isbn" type="string"/>
  </sequence>
</complexType>
```

Then the following actions are added to Flex specification:

```
<OPEN_ELEM>"title"   {return token[ns][TITLE];BEGIN(ATTS);}
<OPEN_ELEM>"author"  {return token[ns][AUTHOR];BEGIN(ATTS);}
<CLOSE_ELEM>"title"  {return token[ns][TITLE]+1;BEGIN(ATTS);}
<CLOSE_ELEM>"author" {return token[ns][AUTHOR]+1;BEGIN(ATTS);}
<ATTS>"isbn"         {return token[ns][ISBN];BEGIN(OPEN_ELEM);}
```

Table 3: Grammar productions containing semantic actions

#	Production	Action
2	$x \rightarrow$ DATA	imp_X(s.*val*)
6	$y \rightarrow$ DATA	imp_Y(s.*val*)

4.5 Constructing the TDX Parser

The driver program of the TDX-based parsing engine is independent of the schema-specific requirement. This enables us to write a high-efficient driver in ANSI C. The generated scanner provides the TDX-based parser a stream of tokens for parsing. The parser maintains an LL(1) parsing table, a set of productions and actions, and a local stack. The stack contains a sequence of grammar symbols with `$`, a symbol used as an endmarker, on the bottom, indicating the bottom of the stack. The parsing table is a 2-D array `T[A][a]`, where `A` is nonterminal, and `a` is a terminal or endmarker `$`. Terminals and non-terminals are represented as integers.

The parsing table contains the production indexes. All the entry is initialized with `ERROR_ENTRY`, a negative integer indicating error entries. The parsing table is implemented based on the generated LL(1) grammar using algorithm4.4 in [1].

The generated grammar productions are stored in a 2-D array `P[I][S]`, where `I` is the production index and `S` indicates the grammar symbols.

The code generator `wsdl2TDX` generates a set of functions performing the semantic validation. The function pointers are stored in an array `pt2Func[I]`, where `I` is the index of the production. This offers a way to invoke a function associated with a production. For example, suppose a grammar contains the following productions associated with semantic actions in Table 3.

The commented sample below illustrates the generated code for performing semantic validation.

```
// type-definition: define a function pointer type
typedef int (*pt2Function)(char *);
...
// define a function pointer array holding function pointers
// and initialize with NULL
pt2Function pt2Func[NUM_Production] = {NULL};
...
// assign the function's address
pt2Func[2] = &imp_X;
pt2Func[6] = &imp_Y;
...
// calling a function using an index
pt2Fun[2](yytex);  // calling imp_X(char *)
pt2Fun[6](yytex);  // calling imp_Y(char *)
```

The generated parser driver implements a modified version of `algorithm 4.3` published in [1]. The token on top of the stack, `X` and the current input token, `a`, determines the behavior of the driver:

a. If `X=a=$`, the parser announces success.

b. If `X=a=DATA`$\neq$`$`, the parser pops of `X`, invokes the corresponding function to perform semantic validation, the reads next input token on successful validation, halts and announces an error otherwise.

c. If `X=a`$\neq$`DATA`$\neq$`$`, the parser pops off `X`, then the reads the next input token.

d. If `X` is a non-terminal, the parser consults entry `T[X][a]` of the parsing table. The entry will be either an index of an `X`-production or an error entry. If it is an error entry, the parser halts and announces a syntax error. Otherwise get the production using the index, replace the `X` on top of the stack by its right side token in reverse order, i.e., the rightmost token is on top of the stack.

The code generator `wsdl2TDX` implements the modified algorithm in a straightforward manner. The following code will illustrate the generated code for the driver program(slightly edited for clarity, variable `a` is the current input token).

```
// get the token on the top of stack
X = stack(top);
// if X==a==$
if(X==a && X==endmarker )
{
    //success of parsing
    printf("parsing successfully");
    return;
}
// if X==a==DATA!=$
else if(X==a && X==DATA)
{
    //pop of X
    top--;
    // perform semantic validation
    // by calling associated function
    if (pt2Func[index](yytext))
    {
        // valid message, get next input token at next loop
    }
    // invalid message
    else
    {
        printf("invalid message %s", yytext);
        return;
    }
}
// if X==a!=DATA!=$
else if(X==a && X!=DATA && X!=endmarker)
{
    // pop of X, get next input token at next loop
}
// X is non-terminal
else
{
    // get the production index
    index = T[-X -1][a-1];from an XML schema.
    //push right side of the production with
    // the rightmost token on top of the stack
    ...
    //read next input token at next loop
}
```

5. PERFORMANCE RESULTS

We compared our implementation to gSOAP 2.7, eXpat 1.2, Apache Xerces for C++ 2.7.0 and a DFA-based XML parser [16]. All code was compiled using gcc 3.2.2 (Xerces compiled with g++ 3.2.3) with option -O2 on a 2.4GHz P4 CPU and 512MB of main memory machine running Red Hat Linux 3.2.2-5.

The raw XML parsing performance was measured on memory-resident messages. Therefore network bandwidth and I/O latency are not measured to compare raw parsing speeds. The first run is discarded (warm-up) and timings of multiple runs were measured with `gettimeofday()`.

The XML schema used for testing is shown in Figure 4. This schema defines an `echoString` message element containing a child element `input` of type XSD string.

Figure 5 shows the performance (elapsed time in microseconds) of the TDX-based parser produced for the schema, the performance of the DFA-based parser, the performance of eXpat, the performance of gSOAP, and the performance of Xerces. The size of the input message is 1024. The performance of TDX-based parser with a scanner produced with Flex -Cfa, and the performance of DFA-based parser with scanner produced with Flex option -Cfa were also compared. The Flex -Cfa options are used to generate a faster scanner but in larger size. This optimization improves performance in terms of speed.

```
<schema targetNamespace="urn:echoString"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="echoString">
    <complexType>
      <sequence>
        <element name="input" type="xsd:string"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Figure 4: The echoString Message Schema

The TDX-based parser combines parsing and validation in one stage. It was compiled with validation and namespace processing. Apache Xerces for C++ is a validating XML parser written in a portable subset of C++ [20]. It was compiled with option validation and namespaces support. All other options are turned off to achieve better speedups (no schema support and no constraints checking). The result shows that the optimized TDX-based parser is 17 to 18 times faster than Xerces, and the non-optimized TDX-based parser is 13 times faster than Xerces, in terms of the total time for parsing and validation.

The eXpat is a non-validating streaming XML parser [4]. It is considered one of the fastest steaming XML parser. It was used without XML namespace support to speed it up. The performance of the optimized validating TDX-based parser is approximately three times faster than non-validating eXpat parser.

The performance of gSOAP shown here has two parts, because gSOAP's parser was not designed to be used as a stand-alone parser. The validation/decoding part indicates the validation time and the time spent on deserializing the message for application. Thus the total time includes parsing time, validation time and deserialization time. The performance of the gSOAP is seven times slower than the TDX-based parser with Flex -Cfa. Note that performance of TDX-based parser is lower than the DFA-based parser. But the difference is very small.

The Figure 6 shows the performance to parse message of a given size in log scale. The x-axis shows the number of elements in the message and the y-axis shows the total time in microseconds. The Xerces has a larger initial time than others. The TDX-based parser is about three times faster than eXpat, and six times faster than gSOAP, seven to eight times faster than Xerces. The result also shows that the TDX-based parser is comparable to the DFA-based parser in terms of performance.

6. CONCLUSION

This paper presented the TDX parsing approach, which is a table-driven, LL(1)-based technique to implement schema-specific XML parsing and validation. TDX parsing achieves

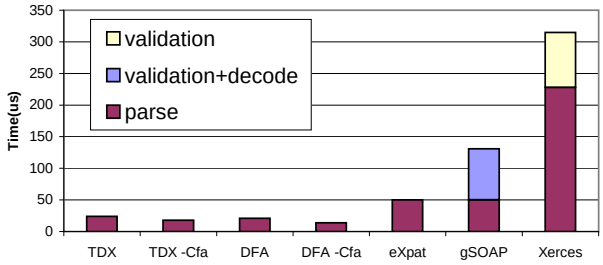

Figure 5: Performance of echoString Parsing for n=1024 (2.40GHz P4)

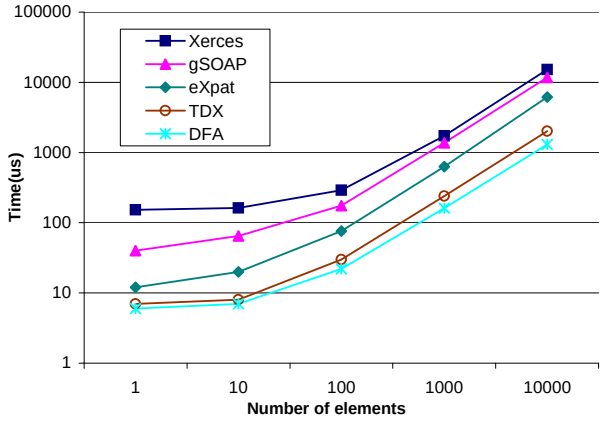

Figure 6: Performance of echoString Parsing

a very high level of performance by combining parsing and validation into one stage and by utilizing an efficient tokenization algorithm and parser engine. The results show that the speed of the parser is several times faster than industry-strength high-performance validating XML parsers. The TDX approach also achieves a high level of modularity and adaptiveness for developing XML Web service applications, because TDX tables are interchangeable and can be easily replaced when a schema is updated, while other compiled approaches require the application to be recompiled.

7. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[2] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *proceedings of the 11th IEEE International Symposium on High-Performance Distributed Computing*, 2002.

[3] K. Chiu and W. Lu. A Compiler-based approach to schema-specific XML parsing. In *First International Workshop on High Performance XML Processing*, May 2004.

[4] J. Clark. eXpat XML parser. http://expat.sourceforge.net.

[5] J. Clark and M. Makoto, editors. RELAX NG Specification. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[6] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid*, 2002.

[7] B. Ford. Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking. Master's Thesis, MIT, 2002.

[8] T.J. Green, A. Gupta, G. Miklau, M. Onizuka and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Transactions on Database Systems (TODS)*, 29(4):752-788, December 2004

[9] C. Kohlhof and R. Steele. Evaluating SOAP for high-performance business applications: Real-time trading systems. In *proceedings of the 2003 International WWW Conference*, Budapest, Hungary.

[10] D. Lee, M. Mani and M. Murata. Reasoning about XML Schema Languages using Formal Language Theory. Technical Report, IBM Almaden Research Center, RJ# 95071, November 2000.

[11] W. Lowe, M.L. Noga and T. Gaul. Foundations of Fast Communication via XML. *Annals of Software Engineering*. 13(1-4):357-379, January 2002

[12] T. Mason and D. Brown. *Lex & Yacc*. O'Reilly and Associates, Inc., 632 Petaluma Ave, CA, 1990.

[13] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

[14] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.

[15] H. S. Thompson and R. Tobin. Using finite state automata to implement W3C XML schema content model validation and restriction checking. In *Proceedings of XML Europe*, 2003.

[16] R. van Engelen. Constructing Finite State Automata for High Performance XML Web Services In *proceedings of the International Symposium on Web Services (ICWS)*, 2004.

[17] R. van Engelen. Code generation techniques for developing light-weight efficient XML Web services for embedded devices. In *proceedings of 9th ACM Symposium on Applied Computing SAC 2004*, Nicosia, Cyprus, 2004.

[18] R. van Engelen and K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid*, 2002.

[19] R. van Engelen, K. Gallivan and S. Pant. developing Web Services for C and C++. *IEEE International Computing*, pages 53-61, march 2003.

[20] Xerces-C++. http://xml.apache.org/xerces-c.

[21] W3C. XML Specfication. http://www.w3.org/TR/2004/REC-xml-20040204.