High-Performance XML Parsing and Validation with Permutation Phrase Grammar Parsers

Wei Zhang and Robert A. van Engelen
Department of Computer Science, Florida State University, Tallahassee, FL 32306
{wzhang,engelen}@cs.fsu.edu

Abstract

The extensibility, flexibility, expressiveness, and platform-neutrality of XML delivers key advantages for interoperability. The interoperability of XML Web services often comes at the price of reduced efficiency of message composition, transfer, and parsing compared to simple binary protocols. This paper presents a high-performance XML parsing and validation technique that is time and space optimal. A schema-specific parsing method is developed that uses a two-stack push-down automaton (PDA) for single-pass parsing and validation without backtracking. The schema validity constraints are packed in a compact parsing table derived from a permutation phrase grammar. This approach reduces both the space and time requirements of XML parsing and validation. By contrast, other XML schema-specific parsing methods trade efficiency for space (larger code and/or data size) or trade space for efficiency (backtracking). Performance results show that the method is significantly faster than traditional validating and non-validating XML parsers.

1. Introduction

The Extensible Markup Language (XML) format is widely adopted as a standard for exchanging structured information, particularly via Web services, to deliver rich content to users and business data processing systems. Attractive properties of XML include its extensibility, flexibility, expressiveness, and platform-neutrality. These properties deliver key advantages for interoperability across a wide spectrum of data-centric business processing systems. Web services technologies and applications have built on the success of XML by providing standardized delivery of structurally and semantically rich content over the Web, as defined by the Simple Object Access Protocol (SOAP) and Web Service Definition Language (WSDL) W3C standards.

The interoperability of XML Web services often comes at the price of reduced efficiency of message composition, transfer, and parsing compared to simple binary protocols. Parsing and validation of XML against a schema is expensive [10,16], as well as the cost of deserialization into usable objects [5, 7]. Several efforts have been made to address the parsing and validation performance through the use of schema-specific (grammar-based) parsers [6, 9, 12, 13, 17, 18, 20]. These parsers encode parsing states and validation rules at compile time by exploiting schema structures and validation rules. XML schema-specific parsing techniques typically exhibit the following properties:

- **Compile-time versus run-time parsers:** all compile-time parsing and validation approaches use specialized compilation techniques to generate customized parsers from a set of schemas. Most of these approaches generate recursive descent parsers [9, 12, 17]. Run-time approaches use generic parsers (drivers or engines) and a grammar-like representation of a schema. This approach is used in Deterministic Finite Automaton (DFA) based parsing [18] and Table-Driven XML (TDX) parsing [22, 23].
- **Blocking versus non-blocking parsers:** a blocking parser acquires the main thread of control and may suspend the entire program until sufficient XML content arrives over the network to be operated on, typically assisted using a timeout policy in a Web services context. In a non-blocking parser the main program is always in control. Buffered data can be incrementally supplied, rather than having the parser fetch all data at once. The main program can invoke the parser at any time to continue the parsing process. Parsing methods based on recursive descent are blocking, because the chain of recursive calls cannot be arbitrarily broken to (temporarily) return control to the main program.
- **Time-efficient versus space-efficient parsers:** a time-efficient parser limits the executed instruction sequence between each octet read from the network port and the instantiated object. The overhead could be as low as a hundred CPU cycles per octet, which has a similar cycle penalty as a memory access. Time efficiency may require a hardcoding of many parsing states to avoid backtracking, thereby increasing the spatial resource requirements for parsing. On the other

hand, space-efficient parsers use backtracking to limit the hardcoding of parsing states. There is a clear similarity of this property to the well-known theoretical time-space differences of scanners based on DFA and Nondeterministic Finite Automaton (NFA).

Current schema-specific XML parsers are either time efficient, but encode many states as a result, or space efficient with backtracking. Some also limit the application of schema validity constraints as a tradeoff to avoid excessive backtracking, or mix time and space efficiency by using backtracking when parsing states cannot be effectively represented at compile-time.

This paper presents a high-performance XML parsing and validation technique that is both time and space optimal. This is achieved through the use of a table-driven XML parsing and validation with permutation phrase grammar. Our *Permutation grammar Table-Driven XML parsing* (pTDX) method utilizes a compact tabular representation of schemas and a push-down automaton (PDA) for single-pass parsing and validation without backtracking. This work builds on our previous work [22, 23], yet is the first successful attempt to achieve both space and time requirements. To avoid backtracking on XML elements and attributes defined by permutable XML schema constructs such as unordered sequence of elements (`xs:all`) and attributes (`xs:attribute`), we extend Backus Naur Form (BNF) with support of *permutation phrase grammar* representation of a schema. The permutation phrase grammar is a compact representation of common XML element and attribute permutations that have specific occurrence constraints. Thus, space optimality is ensured by encoding permutations compactly. The permutation phrase grammar requires a specialized recognizer, which is implemented by a two-stack push-down automaton.

Our pTDX tool implements a run-time, asynchronous, and predictive parsing engine. In contrast to gSOAP [17] and XML Screamer [9], pTDX is not based on recursive descent parsing, thus pTDX implements non-blocking parsing operations. Furthermore, pTDX implements a schema-directed scanner which significantly improves input scanning performance. It scans tags and certain XML Schema Definition (XSD) primitive type values and character data (CDATA), and breaks these up into tokens for the parsing engine. The parsing engine performs operations on tokens. This is more efficient than operations at the string level that most XML parsers do. Buffering of tokens is needed but is only limited to scan ahead of namespace bindings, which may be applicable to attributes prior to the occurrence of the namespace binding.

In addition, unlike some validation parsers [6,13], pTDX implements namespaces support in an efficient way. At the lexical level, pTDX compares element and attribute tags as well as other data content directly at the UTF-8 level, with namespace normalization when elements and attributes are namespace qualified. This method has similarities with gSOAP and XML Screamer. XML Screamer optimizes scanning by minimizing the redundant scanning of input data. However, for some cases, the ideal of visiting each character just once can not always be achieved and a backtrack scan is needed. By contrast to XML Screamer, pTDX implements a non-backtracking scanning of input data that guarantees each input character is visited at most once.

Similar to our initial work on TDX [22,23], pTDX is implemented independent of XML schemas; the parsing table and tokens are provided as hot-swappable modules. Content data validation routines can be pushed as semantic actions onto stacks and will be invoked by the parsing engine at runtime. Thus, when a schema is updated or extended, a new parsing table can be updated and loaded at run time.

The remainder of this paper is organized as follows. We first give a brief description of pTDX architecture in Section 2. In Section 3, we introduce permutation phrase grammar and give mapping rules from XML schema components to permutation phrase grammar. Permutation phrase parsing is described in Section 4. Section 5 gives table-driven permutation phrase grammar parsing using a two-stack PDA. In Section 6, we describe how to optimize schema-directed scanner. Performance evaluation is given in section 7 and related work is discussed in Section 8. Conclusions are drawn in Section 9.

2. Overview of pTDX

The pTDX approach consists of three stages: schema specification processing, parsing table generation, and runtime message processing, where the pTDX engine uses a two-stack PDA to support permutation phrase parsing. A set of non-permutation mapping rules for translating WSDL or XML schema description into LL(1) grammar is defined in [22,23]. Unordered sequence elements `xs:all` and attributes `xs:attribute` as well as `xs:any` are mapped onto permutation phrase grammar rules (see Section 3). The grammar rules preserve structural and semantic validity constraints imposed by XML schemas. Application-specific events can be inserted, e.g. manually, into the productions of generated grammar for automatic invocation at running time before parser generator is called. The parser generator generates parsing table, scanner, and content data validation routines. A schema-directed Flex [15] description of the scanner is fed to Flex to generate DFA-based XML token scanner in C. The scanner scans tags and CDATA and breaks these up into tokens for the parsing engine. The parsing engine then performs the well-formedness checks, structural and content data validation, and generates events to invoke application-specific actions if applicable, and feeds data to the application's back-end by consulting the generated parsing table.

3. Permutation Phrase Mapping Rules

3.1. Permutation Phrase Grammar

A permutation phrase is a grammatical phrase that specifies a syntactic construct as any sequence of constituent elements in which each element occurs exactly once and the order is irrelevant [3, 4]. A permutation phrase grammar is a context-free grammar (CFG) which contains production $v \to p$, where p is a permutation phrase. A permutation phrase grammar offers an efficient way for presenting and parsing free-ordered elements such as `xs:all`, `xs:any`, and `xs:attribute`. Consider, for example, the following schema fragment:

```
<xs:complexType>
  <xs:all>
    <xs:element name="a" type="xsd:integer"/>
    <xs:element name="b" type="xsd:boolean"/>
    <xs:element name="c" type="xsd:string"/>
  </xs:all>
</xs:complexType>
```

This schema grammar can be represented using Cameron's permutation phrase notation [4] as

$$S \to \langle\!\langle a \parallel b \parallel c \rangle\!\rangle$$

It can be argued that permutation phrase grammar definitions can be expanded to EBNF grammar definitions by summing up all possible permutations. However, each permutation phrase composed of n constituents yields $n!$ alternative phrases in a CFG. For example, an element with five attributes, not uncommon in the Web services, would generate 120 different productions in the CFG. Furthermore, such grammar violates the LL(1) properties. Applying `left-factoring` to these productions to eliminate such ambiguity introduces more productions.

3.2. Mapping `xs:all` and `xs:attribute` to Permutation Phrase Rules

XML schema `xs:all` component specifies that its child elements can appear in any order and that each child element can occur zero or one time. Similarly, `xs:complexType` may define an element that contains a set of attributes specified by `xs:attribute`. Attributes of an element can occur in any order. These schema components are mapped to permutation phrase grammar.

The mapping $\Gamma[\![X]\!]N$ takes a schema component X and a designated non-terminal N and returns a set of grammar productions starting with non-terminal N for parsing instances of X (see [22, 23] for the non-permutation phrase mapping rules). We use A_i to represent an attribute component. Table 1 lists parts of mapping rules using permutation phrase grammar for `xs:all` and `xs:attribute`. Mapping rules for `xs:any` are defined in a similar way.

4. Permutation Phrase Parsing

Permutation phrases can be parsed efficiently by a two-stack PDA. We first introduce an algorithm that parses a permutation phrase without optional elements, i.e. no productions that produce empty strings, and then we relax such constraints to extend the solution to cover optional elements.

Assume a permutation phrase holds the following two constraints:

- **Nonemptiness:** No constituent of any permutation derives empty string, i.e. no optional elements.
- **Uniqueness:** No constituent of any permutations shares the same first symbol, i.e., $FIRST(X_1) \cap FIRST(X_2) \cap \cdots \cap FIRST(X_n) = \emptyset$ for the permutation phrase $\langle\!\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\!\rangle$.

A two-stack PDA can efficiently parse the permutation phrase as follows: the constituent elements are first pushed on to the "main stack". The element on top of the main stack is checked to see if it derives the current input token. If it does, then the parser pops it off from the main stack and pushes all the elements from an "auxiliary stack" onto the main stack, and empties the auxiliary stack. The parser then reads the next token from the input stream and checks again. If the element on top of the main-stack does not derive the current input token, it is popped off and saved by pushing it onto the auxiliary stack. If no permutation phrase element on the main stack derives the symbol, the parser reports an error. Therefore the parser can detect errors as early as no permutation element deriving the input symbol without consuming all permutation elements.

4.1. Handling Optional Elements

Technically, there is no mechanism to transform a grammar with permutation phrases containing optional constituent elements into an unambiguous context-free grammar, because there is no way to determine where an empty constituent is derived; i.e. whether it occurs before the first nonempty element, between two nonempty elements, or after the last one. However, the order of the derivation is of no importance to a permutation phrase. As proposed by Cameron in [4], the strategy to parse such permutation phrase is: "parse nonempty constituents as they are seen until a symbol is encountered which is not a first symbol of any of the constituents remaining to be parsed." Fortunately, the two-stack machine adopts this strategy intuitively: all the optional elements are left on the auxiliary stack once all the nonempty elements have been parsed. The permutation parsing algorithm with optional elements support is given in Algorithm 1.

4.2. Preserving the LL(1) Properties

Permutation mapping rules for `xs:all` and `xs:attribute` do not introduce productions that violate LL(1) properties of the grammar. The component `xs:all` can only occur as the sole child of `xs:complexType` and

Rule#	Translation		
1	$\Gamma[\![$<all> $X_1 X_2 \ldots X_n$ </all>$]\!]N$	$=$	$\{N \rightarrow \langle\!\langle N_1 \parallel N_2 \parallel \cdots \parallel N_n \rangle\!\rangle\} \cup \bigcup_{i=1}^{n} \Gamma[\![X_i]\!]N_i$
2	$\Gamma[\![$<complexType name='T'> $X A_1 A_2 \ldots A_n$ </complexType>$]\!]N$	$=$	$\{T \rightarrow T_1 T_2\} \cup \Gamma[\![X]\!]T_1 \cup \{T_2 \rightarrow \langle\!\langle T_{21} \parallel T_{22} \parallel \cdots \parallel T_{2n} \rangle\!\rangle\} \cup \bigcup_{i=1}^{n} \Gamma[\![A_i]\!]T_{2i}$

Table 1. Mapping `xs:all` and `xs:complexType` attributes to permutation phrase productions.

Algorithm 1: Parsing a permutation phrase with optional elements.

```
Input: $p \rightarrow \langle\!\langle Y_1 \parallel Y_2 \parallel \cdots \parallel Y_n \rangle\!\rangle$: permutation phrase production, $ip$:
       input stream pointer)
Output: A sequence of constituent elements representing the left most
        derivation for parsing the input if the input is in $L(p)$; otherwise a
        parsing error.
begin
    for $i \leftarrow n$ to $1$ do
        $push(S_m, Y_i)$;
    end
    $c = readnext(ip)$;
    repeat
        $X \leftarrow top(S_m)$;
        if $c \in FIRST(X)$ then
            $output(X); pop(S_m); c = readnext(ip)$;
            while $S_a$ not empty do
                $t \leftarrow top(S_a); pop(S_a); push(S_m, t)$;
            end
        else
            $push(S_a, X); pop(S_m)$;
        end
    until $X$ is not a permutation phrase element or main stack is empty;
    while $S_a$ is not empty do
        $t \leftarrow top(S_a)$;
        if $t$ is an optional element then
            $pop(S_a)$;
        else
            $error()$;
        end
    end
end
```

can only have `xs:element` children. Thus no elements in `xs:all` component within the same `xs:complexType` shares the same tag names when reading qualified tag names. Moreover, `xs:all` element can not occur within `xs:choice` component. Therefore mapping `xs:all` to permutation phrase grammar productions also preserves the LL(1) properties. Similarly, mapping `xs:attribute` to permutation phrase grammar preserves LL(1) properties.

5. Constructing Table-Driven XML Parsers

5.1. Parsing Table

Constructing the parsing table is similar to the method used by TDX through the use of FIRST and FOLLOW sets [1]. The FIRST and FOLLOW set definitions are extended to support the permutation grammar. The permutation grammar composition symbol is commutative and associative and the FIRST and FOLLOW sets are computed as union of all elements. Consider a permutation production, $A \rightarrow \langle\!\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\!\rangle$, to compute the FIRST set, apply the following rules:

1. $FIRST(A) = FIRST(X_1) \cup FIRST(X_2) \cup \cdots \cup FIRST(X_n)$.
2. Add ε to $FIRST(A)$ if for all i, $X_i \overset{*}{\Rightarrow} \varepsilon$, where $1 \leq i \leq n$.
3. $FIRST(\langle\!\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\!\rangle) = FIRST(X_1) \cup FIRST(X_2) \cup \cdots \cup FIRST(X_n)$.
4. Add ε to $FIRST(\langle\!\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\!\rangle)$ if for all i, $X_i \overset{*}{\Rightarrow} \varepsilon$, where $1 \leq i \leq n$.

To compute the FOLLOW set, we apply the following rules:

1. $FOLLOW(X_1) = FIRST(X_2) \cup FIRST(X_3) \cup \cdots \cup FIRST(X_n)$.
2. $FOLLOW(X_n) = FIRST(X_1) \cup FIRST(X_2) \cup \cdots \cup FIRST(X_{n-1})$.
3. $FOLLOW(X_i) = FIRST(X_1) \cup \cdots \cup FIRST(X_{i-1}) \cup FIRST(X_{i+1}) \cup \cdots \cup FIRST(X_n)$, where $1 < i < n$.

5.2. Parsing Engine

The pTDX table-driven predictive permutation parser uses an input buffer, two stacks, a predefined parsing table with grammar definitions, and an output stream. The main stack is initialized with \$, the endmarker, and S, the start symbol on top. The current symbol X, which is the symbol on top of the main stack, and c, the current input symbol, together with the state of the auxiliary stack, i.e. empty or not, determine the parsing action.

- **X is a terminal**. If $X = c \neq \$$, the parsing engine pops X from the main stack and read the next input symbol. If $X = \$$, the parser halts and announces success. Otherwise the engine announces an error.
- **X is a nonterminal**. The engine first checks the empty status of the auxiliary stack. If it is not empty, the action depends on whether X is a permutation nonterminal or a regular one. If X is a permutation symbol, the engine moves all symbols of the auxiliary stack into the main stack. If X is a regular symbol, the engine check to see if all symbols of the auxiliary stack can generate empty string. An error is announced when a symbol that cannot generate empty string is present. The engine pops all symbols that can generate empty string. If the auxiliary stack is empty, the engine consults the entry $M[X, c]$ of the parsing table M, and replaces X by the right hand side of the production, with the left most symbol on top of the stack[1].

6. Schema-Directed Scanner

6.1. Optimization by Leveraging Schema Information

Input scanning performance is significantly improved by efficiently leveraging schema information. When exactly

[1]We use $\overline{X}$ to represent a permutation constituent element symbol to distinguish it from a regular symbol, e.g $\langle\!\langle X \| Y \rangle\!\rangle$ are represented as $\overline{X}, \overline{Y}$.

Test Case	Schema Filename	Schema Size (Bytes)	No. of Elts. `<xs:all>`	Instance Filename	Instance Size (Bytes)	Throughput (MB/Sec)				
						Validating Parsers			Non-Validating	
						pTDX	gSOAP	Xerces	DFA	Expat
G21_2k	g.xsd	4021	21	g.xml	2341	41	11	5	34	23
A50_64k	a.xsd	3155	50	a 64k.xml	68060	33	10	3	38	26
A50_3k	a.xsd	3155	50	a 3k	3016	31	8	3	38	21
B5_0.2k	b.xsd	814	5	b.xml	291	24	3	3	26	13
B5_8k	b.xsd	814	5	b 8k.xml	8232	40	19	7	44	37
A50_16k	a.xsd	4021	50	a50 16k	17156	32	11	10	39	26
A2_0.3k	a2.xsd	569	2	a2 0.3k	341	28	3	2	31	12
A4_0.4k	a4.xsd	668	4	a4 0.4k	452	32	4	2	35	14
A8_0.6k	a8.xsd	881	8	a8 0.6k	678	34	5	4	38	16
A16_1k	a16.xsd	1314	16	a16 1k	1124	35	6	2	39	18
A32_2k	a32.xsd	2190	32	a32 2k	2036	36	8	3	39	20
A32_4k	a32.xsd	2190	32	a32 4k	3886	34	10	3	42	22
A32_8k	a32.xsd	2190	32	a32 8k	7584	35	10	3	42	25
A32_16k	a32.xsd	2190	32	a32 16k	16826	35	17	2	38	25
A32_32k	a32.xsd	2190	32	a32 32k	33462	36	11	4	42	26

Table 2. Test Cases and measurements.

Algorithm 2: Predictive Permutation Parsing

```
Input: A string $w$ and a parsing table $M$ for permutation grammar $G^p$
Output: If $w$ is in $L(G^p)$, a leftmost derivation of $w$; otherwise, an error
        indication
Initially, the parser is in a configuration in which it has $S on the main stack
$S_m$ with S, the start symbol of the grammar $G^p$ on top. The auxiliary stack
$S_a$ is initialized empty. The current input c points to the first symbol of $w$$.;
repeat
    $X \leftarrow top(S_m)$;
    if $X$ is a terminal or $ then
        if $X = c$ then
            $pop(S_m)$;
            $c \leftarrow read\_next(input)$
        else
            $error()$;
        end
    else
        /*$X$ is either a permutation nonterminal or a regular terminal.    */
        $pop(S_m)$;
        if $S_a$ is empty then
            if $M[X, c] = X \rightarrow Y_1 \dots Y_n$ or
            $M[X, c] = X \rightarrow \langle\langle Y_1 \parallel \cdots \parallel Y_k \rangle\rangle$ then
                for $i \leftarrow k$ to 1 do
                    if $M[X, c] = X \rightarrow \langle\langle Y_1 \parallel \cdots \parallel Y_k \rangle\rangle$ then
                        $push(S_m, \overline{Y_i})$
                    else
                        $push(S_m, Y_i)$;
                    end
                end
                $output(M[X, c])$;
            else
                $error()$;
            end
        else
            /*Auxiliary stack has permutation nonterminals left.    */
            if $X$ is a permutation nonterminal then
                while $S_{aux}$ is not empty do
                    $t = top(S_a); push(S_m, t); pop(S_a)$;
                end
            else
                /*$X$ is a regular nonterminal; If the nonterminals left in
                  auxiliary stack can generate empty string.    */
                while $S_{aux}$ is not empty and no error is found do
                    $t \leftarrow top(S_{aux})$;
                    if $t \overset{*}{\Rightarrow} \varepsilon$ then
                        $output(``t \rightarrow \varepsilon")$;
                    else
                        $error()$;
                    end
                end
            end
        end
    end
until $X = \$$;
```

one element is expected, then scanning for that particular tag, rather than the generic element production, is significantly more effective since it only amounts to matching the tag name. Similarly, when a specific `xs:type` is expected for an element of simple type, scanning that specific type is preferable. For example, when an `xs:boolean` of an element content data is expected, scanning the specific string "true" or "false" turns out to be more efficient than scanning the generic string first and then comparing the string to see if it is either "true" or "false". By using carefully designed Flex regular expression and efficient use of Flex [15] start conditions, the Flex description of the pTDX scanner can be significantly optimized to the schema. Consider for example the following schema fragment:

```
<xs:element name="c">
  <xs:complexType>
    <xs:all>
      <xs:element name="a" type="xs:integer"/>
      <xs:element name="b" type="xs:boolean"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

The above schema fragment specifies an element `"c"` that has two children elements `"a"` of type `xs:integer`, and `"b"` of type `xs:boolean`. The generated Flex description of the specialized scanner is shown in Figure 1.

This generated Flex specification exactly drives the scanner to initially scan the start or end tag of element `"c"` because it is in the Flex start condition `INITIAL` by default. When the start tag of the element `"c"` is seen, the scanner goes to start condition `C_CONTENT`, in which start tag of either element `"a"` or element `"b"` can be exactly expected. When the start tag of element `"a"` is seen, the scanner goes to the start condition `A_CONTENT`, where digital numbers are expected to be met (refer [15] for more details of Flex).

6.2. Tokenization

The overall performance can be improved by tokenization because matching tokens once is more efficient than repeatedly comparing strings. The scanner of pTDX breaks

```
int-type [0-9]+
%x A_CONTENT
%x B_CONTENT
%x C_CONTENT
%x A_CLOSE
%x B_CLOSE
%%
"<c>" {BEGIN(C_CONTENT);}
<C_CONTENT>"<a>" {BEGIN(A_CONTENT)};
<C_CONTENT>"<b>" {BEGIN(B_CONTENT)};
<C_CONTENT>"</c>" {BEGIN(INITIAL)};
<C_CONTENT>.  {error()};
<A_CONTENT>{int-type} {BEGIN(A_CLOSE)};
<A_CONTENT>.  {error_val()};
<B_CONTENT>"true" {BEGIN(B_CLOSE)};
<B_CONTENT>"false" {BEGIN(B_CLOSE)};
<B_CONTENT>.  {error_val()};
<A_CLOSE>"</a>" {BEGIN(C_CONTENT)};
<A_CLOSE,B_CLOSE>".  {error()};
<B_CLOSE>".  {error()};
```

Figure 1. Generated Flex description of the specialized scanner for element `"c"` consisting of two children elements `"a"` and `"b"` of type `xs:integer` and `xs:boolean` respectively (simplified for readability).

input data into normalized tokens for the parsing engine. All tag names and all content data that can be represented as integers such as `xs:boolean` are tokenized and normalized.

6.3. Elimination of Backtracking

Scanning performance can be greatly improved by eliminating redundant scanning over the input. Through the use of carefully designed Flex start conditions and rules, pTDX scanner avoids backtracking, i.e. it guarantees that visiting of each character of input data is just once. Consider the same example used in [9], the following two XML elements within an `xs:integer`:

```
<e>123456</e>
<e>123<!-- comment -->456</e>
```

Both two elements represent the same integer although the second form is very uncommon. At this level, most XML parsers scan the integer in the second form with backtracking and retrying with a different deserializer. However, pTDX scanner can avoid backtracking using three rules. One rule scans and discards the comment. Another rule scans the integer and pushes the integer onto a cached stack. The third one scans the end tag of the element. Upon the end tag is met, the scanner assemblies the integer for the parsing engine.

6.4. Start Tag Processing and Namespaces Optimization

XML Namespaces [21] pose some challenges for a high performance XML parser or validator. Within the start tag, element tag name and attribute name(s) can not be parsed until the namespace binding is resolved, whether qualified or not. However, there is no way to determine if a QName will be redefined or the default namespace will be changed until the last attribute within the tag has been seen. The strategy is to scan and cache all the tags into a queue and namespaces onto a stack (or using hash). When the closing delimiter `'>'` is met, the scanner emits the tokens by consulting the namespaces stack.

6.5. Early Error Detecting

The pTDX engine is capable of detecting early parsing errors during scanning stage in some cases. An XML Schema provides the scanner with information about whether an element or an attribute is required or optional, and what is specific type of the content data. A well-formedness error is announces when a required tag name is expected and not present. A validation error is issued when a specific data type is expected and no such data of that type present. For example, Figure 1 indicates that a validation error is issued when neither “true” nor “false” is present when the scanner is the `<B_CONTENT>` condition.

7. Performance Evaluation

Benchmarks were chosen to measure the performance of the parsers for different schema structures. To this end, we choose industry-quality XML-based Web services for our tests. Each test instance consists of a string of UTF-8 XML content stored in continuous memory buffer. File system I/O or network overhead is not measured. Elements of `xs:all` are randomly arranged in the message instance for accurate measurements of free-ordered property. For the same reason, attributes in `xs:attribute` are also placed in a random way. Multiple instances are parsed from separate buffers to avoid any effect possibly caused by high cache hit rates. The first run is intended to warm up the system and is discarded. Average parsing time of a hundred runs is reported. Time is measured as Wall Clock real time elapsed using system call *gettimeofday()*. Memory usage is measured using valgrind –tool=massif. All tests reported here were conducted on a Dell Optiplex GX620 with a 3.0 GHz Intel Pentium D processor, and 2 GB of main memory, running Linux 2.6.20-1.2320. All parsers reported here were compiled with GCC version 4.1.1 option -O2.

7.1. Parser Performance Compared

For the performance measurements, we compared our pTDX with two widely used runtime-based parsers, Xerces [2] and expat [14]. Xerces is an industry widely used, popular high-performance parser. It supports both validating and non-validating parsing mode with capability of schema caching. We measured performance with validation in SAX mode. We chose Xerces with version of 2.7.0 for Linux.

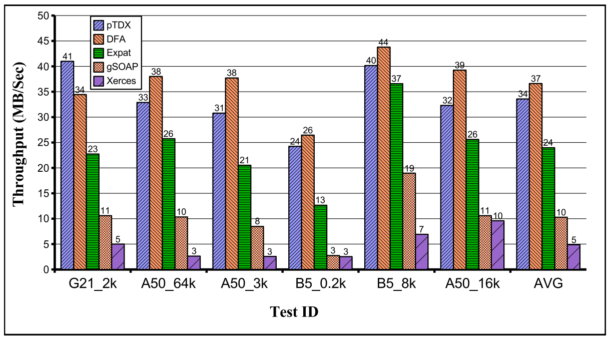

Figure 2. Performance comparison of validating and non-validating parsers

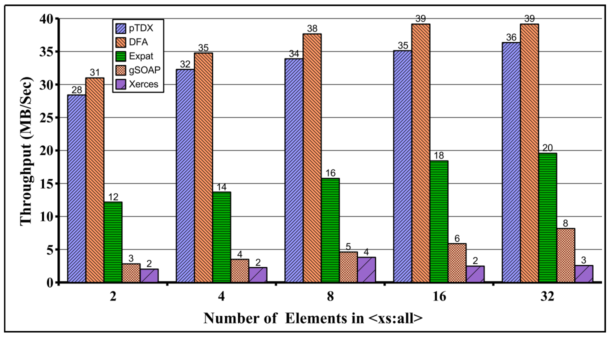

Figure 3. Effect of number of elements in `xs:all` of pTDX Parser.

Expat is a non-validating streaming XML parser that only checks well-formedness of the input XML message. It is considered one of the fastest non-validating parsers. The latest version of 2.0.1 for Linux was chosen for the performance comparison.

We also compared two compile-based parsers, DFA [18] and gSOAP [17]. The gSOAP toolkit generates highly optimized and C-based XML validation parsers. Performance comparisons [8, 18, 19] have shown that gSOAP has very fast parsers and deserializers. The presented comparisons are not completely fair for gSOAP whose timings include parsing, validation, and deserialization while other parsers do not deserialize data.

A DFA-based parser can significantly improve the XML parsing by encoding parsing states with a DFA. However, the states of DFA increases exponentially when the number of permutation elements increases. Thus the DFA-based parser measured in this paper was implemented only to perform well-formdedness parsing. Because it has a similar scanner as pTDX, it provides a base line for the pTDX validation cost. It also provides a comparison with compile-based parsing and runtime-based parsing.

No application-specific events were triggered in the measurements, although pTDX offers the capability to trigger events.

7.2. Test Cases

To test whether performance is related to some specific schemas or to the size of the input message, we use a set of schemas and combination of schema instances. Schema `g.xsd` is directly taken from the `GoogleSearch.wsdl`, containing 21 elements in `xs:all`. Schema `a.xsd` is taken from `AmazonSearch.wsdl` and customized to have 50 elements in `xs:all`. Schemas ai`.xsd`, where $i = 2, 4, 8, 16, 32$ are derived from `a.xsd` to have $2, 4, 8, 16, 32$ elements in `xs:all` respectively. Schema `b.xsd` is a very simple one used to test attributes in `xs:attribute`. It has five attributes which is not uncommon in the real Web services. The first part (before the symbol '_') of a test case name denotes the schema name and the number of elements in `xs:all` or the number of attributes in `xs:attribute`. The second part (after '_') represents the approximate instances size in KB. Table 2 lists the test cases.

7.3. Performance Discussion

Performance measurements were taken for each of the chosen parsers on each of the test case. Performance was measured in throughput MB/Sec. Test cases and the measured results are listed in Table 2. The throughput of pTDX ranges from 24 MB/Sec to 40 MB/Sec for various message sizes and different numbers of elements in `xs:all` or `xs:attribute`. Compared to runtime parsers, pTDX is on average 10 times faster than Xerces parser with validation, and can be up to 14 times faster than validation Xerces. pTDX is also 2 times faster that fast non-validating Expat.

Compared to schema-specific (or compiler-based) parsers, pTDX is still 4 times faster on average, and can be up to 9 times faster than gSOAP (though gSOAP also deserialized data in these performance statistics, while pTDX does not). TDX-based non-validating parser is on average 1.2 times faster than validating pTDX, and 1.7 times faster than non-validating Expat. These results indicate that pTDX offers efficient technique for XML validation. It is also indicated that the scanner takes significant portion of the processing time. This is because both the TDX and pTDX scanners also perform tokenization.

7.3.1 Scalability to Number of Elements in `xs:all`

The results in Figure 3 demonstrate that the number of `xs:all` elements does not play a significant performance penalty for all the parsers tested. The number of the elements in `xs:all` makes not much difference to throughput. The number of elements in `xs:all` varies from 2 to 32, while the throughput only varies between 28.42 MB/Sec and 36.26 MB/Sec for pTDX validating parser. This indicates that pTDX parser has very good scalability to the number of elements in `xs:all` and `xs:attribute`.

Message Size (KB)	pTDX		gSOAP		Expat	Xerces
	Run time (KB)	Compile time (KB)	Run time (KB)	Compile time (KB)	Run time (KB)	Run time (KB)
2	16	10	2	45	4	280
4	16	10	3	45	8	280
8	16	10	4	45	12	280
16	16	10	10	45	32	280
32	16	10	20	45	64	280
64	16	10	40	45	128	280

Table 3. Runtime and compile time memory usage comparison (32 `xs:all` elements).

7.4. Memory Usage Analysis

Table 3 shows the runtime and compile time memory usage of different parsers specialized on a schema containing 32 `xs:all` elements. The results indicate that pTDX requires a constant of 16KB rum time memory during the parsing time. These memory consumption is used by the scanner, constructed by the Flex. Flex generated scanner requires buffering input string that it is 16KB by default. pTDX requires 10KB to store productions, parsing tables, and parsing stacks. gSOAP runtime memory usage increases as the size of message document increases, partly due to deserialization of objects. However, it still requires 45KB to decode parsing states at compile time. This is four times larger than pTDX memory consumption. Thus, pTDX implements a space optimal parser. The validating Xerces parser and nonvalidating Expat parser are runtime XML parsers, thus no compile-time overhead exists. The table shows that Xerces consumes a constant of 280KB memory for parsing while Expat memory usage increases linearly as the size of XML document increases. Though not shown in the table, the DFA-based parser requires 10KB at compile time because it has similar scanner as pTDX. However, the runtime memory measurement of the DFA-based parser is not available because the measured DFA-based parsers were only implemented as a well-formedness parsers due to the fact that the number of DFA states has factorial increase as the number of `xs:all` elements increases. In general, the number of DFA states for recognizing the `xs:all` elements, $N(n)$, is determined by:

$$N(n) = \sum_{j=0}^{n-1} \prod_{i=0}^{j} (n-i), \qquad (1)$$

where n is the number of `xs:all` elements. Clearly it is not practical without state reduction.

8. Related Work

There have been many efforts aimed to improve performance of XML parsing and validation in the past recent years. A promising technique is schema-specific XML parsing [6, 9, 11–13, 16–18, 20, 22, 23]. Schema-specific XML parsing achieves performance gains by exploiting schema information to compose a parser at compile time and utilizing the parsing states at runtime to verify schema validation constraints.

Our previous work on the gSOAP toolkit [17] is the earliest work on a schema-specific LL(1) recursive descent parser for XML with namespace support and validation. To our knowledge, this was also the first published work in the literature to suggest an integrated approach to schema-specific parsing by collapsing scanning, parsing, validation, and deserialization into one phase. However, gSOAP implements a recursive descent the parser that involves function calling overhead and blocking property.

In [18] Van Engelen presents a method that integrates parsing and validation into a single stage by using a two-level schema in which a lower-level Flex scanner drives a DFA validation. The DFA is directly constructed from a schema based on a set of mapping rules. However, this approach can only process a non-cyclic subset of XML schema due the limitations of regular languages described by DFAs. Furthermore, this approach is not applicable in practice for permutation phrase that consists of even not a large number of elements due to the fact that the number of DFA states increases exponentially.

Chiu et al. [6] also suggest an approach to merge all aspects of low-level parsing and validation by extending DFAs to nondeterministic generalized automata. They also provide a technique for translating these into deterministic generalized automata. However, translating from an NFA to a DFA may blow up the number of states, thus limiting these parsers to small occurrence constraints. Furthermore, their approach does not support namespace, which is an essential requirement for SOAP compliance.

Cardinality-constraint automata (CCA) [13] offers an efficient schema-aware XML parsing technique by extending deterministic finite automata with cardinality constraints on state transitions. These automata can easily take care of occurrences constraints imposed by schema. Unfortunately, CCA does not provide mechanism for well-formedness checking.

TDX [22, 23] provides an integrated approach that combines well-formedness checking, content-model validation and application-specific event by pre-encoding parsing states in a tabular form at compile time and by utilizing an efficient a push-down automaton at runtime. However, TDX relies on exponential enumerations of permutation phrases and is therefore not space optimal.

XML Screamer [9] presents an efficient parser generator that translates XML schema into a parser either in C or Java code. Similar to gSOAP and the work by Chiu et al., XML screamer also integrates deserialization with scanning, parsing, and validation. It demonstrates that high-performance can be obtained by careful design of APIs. The tool uses recursive descent with backtracking, and covers a large schema space. As with all recursive descent parsers, XML Screamer is a blocking parser. More recent work that builds on XML Screamer is iScreamer [11]. iScreamer is a schema-directed interpretive XML parser and achieves high-performance gains by using a carefully tuned set of special-purpose bytecodes. iScreamer, does not support full schema features. Also, its reliance on specialized bytecodes may hinder its acceptance.

9. Conclusion

In this paper we presented a table-driven permutation phrase grammar parsing technique that is time and space optimal for schema-specific XML parsing and validation. The experimental pTDX parser implementation demonstrates that high-performance parsing and validation of XML is achieved both in terms of time and space. Free-ordered constraints such as `xs:all`, `xs:attribute`, and `xs:any` are efficiently parsed and validated using a two-stack PDA permutation phrase grammar parsing engine. Because the method does not rely on recursive descent, the parser is nonblocking and can be used when asynchronous message passing is required.

References

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

[2] Apache Foundation. Xerces XML Parser. Ghttp://xerces.apache.org/.

[3] A. I. Baars, A. Löh, and S. D. Swierstra. Functional pearl parsing permutation phrases. *Journal of Functional Programming*, 14(6):635–646, 2004.

[4] R. D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Program Languages and Systems*, 2(1-4):85–94, 1993.

[5] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.

[6] K. Chiu and W. Lu. A compiler-based approach to schema-specific XML parsing. In *In proceedings of The First International Workshop on High Performance XML Processing*, 2004.

[7] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and evaluation of RMI protocols for scientific computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, pages 61–86, Washington, DC, USA, 2000. IEEE Computer Society.

[8] M. R. Head, M. Govindaraju, R. van Engelen, and W. Zhang. Benchmarking XML processors for applications in Grid Web services. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 121–133, New York, NY, USA, 2006. ACM.

[9] M. G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, and M. Mercaldi. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 93–102, New York, NY, USA, 2006. ACM.

[10] W. Lowe, M. Noga, and T. Gaul. Foundations of fast communication via XML. *Annals of Software Engineering*, 13:357–379, 2002.

[11] M. Matsa, E. Perkins, A. Heifets, M. G. Kostoulas, D. Silva, N. Mendelsohn, and M. Leger. A high-performance interpretive approach to schema-directed parsing. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1093–1102, New York, NY, USA, 2007. ACM.

[12] E. Perkins, M. Matsa, M. G. Kostoulas, A. Heifets, and N. Mendelsohn. Generation of efficient parsers through direct compilation of XML schema grammars. *IBM Syst. J.*, 45(2):225–244, 2006.

[13] F. Reuter. Cardinality automata: A core technology for efficient schema-aware parsers, 2003. http://www.swarms.de/publications/cca.pdf.

[14] SourceForge.net. http://expat.sourceforge.net.

[15] sourceforge.net. Flex: The fast lexical analyzer. http://flex.sourceforge.net/.

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

[17] R. van Engelen. The gSOAP toolkit 2.1, 2001. http://gsoap2.sourceforge.net.

[18] R. van Engelen. Constructing finite state automata for high performance XML Web services. In *proceedings of the International Symposium on Web Services (ISWS)*, 2004.

[19] R. van Engelen. A framework for service-oriented computing with reusable C and C++ Web service components. *accepted for publication in ACM Transactions on Internet Technologies*, 2008.

[20] 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*, pages 128–135, Berlin, Germany, May 2002.

[21] W3C. XML schema specification, October 2004. Part1:http://www/w3c.org/TR/xmlschema-1/.

[22] W. Zhang and R. van Engelen. A table-driven streaming XML parsing methodology for high-performance Web services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 197–204, Washington, DC, USA, 2006. IEEE Computer Society.

[23] W. Zhang and R. A. van Engelen. TDX: a high-performance table-driven XML parser. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 726–731, New York, NY, USA, 2006. ACM.