An Adaptive XML Parser for Developing High-Performance Web Services

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

Abstract

This paper presents an adaptive XML parser that is based on table-driven XML (TDX) parsing technology. This technique can be used for developing extensible high-performance Web services for large complex systems that typically require extensible schemas. The parser integrates scanning, parsing, and validation into a single-pass without backtracking by utilizing compact tabular representations of schemas and a push-down automaton (PDA) at runtime. The tabular forms are constructed from a set of schemas or WSDL descriptions through the use of permutation grammar. The engine is implemented as a PDA-based, table-driven driver, as a result, it is independent of XML schemas. When XML schemas are updated or extended, the tabular forms can be regenerated and populated to the generic engine without requirement of redeployment of the parser. This adaptive approach balances the need for performance against the requirements of reconstruction and redeployment of the Web services. Our experiments show the adaptive parser usually demonstrates performance of 5 times faster than traditional validating parsers and performance drop within 20% of the fastest fully compiled traditional validating parsers.

1. Introduction

The Extensible Markup Language (XML) format delivers key advantages in interoperability and is widely adopted as a standard for exchanging structured information by Web services. 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. However, 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. Several studies have evaluated the performance of SOAP and concluded that SOAP and XML incur a substantial performance penalty compared to binary protocols [4, 9, 10]. Parsing and validation of XML against a schema is expensive [12, 19], as well as the cost of deserialization into usable in-memory objects for applications [6, 10].

Several efforts have been made to address the parsing and validation performance through the use of grammar-based parser generation by leveraging XML schema languages such as DTD [23], XML schema [14], and Relax NG [8] at compile time. Compiled schema-specific parsers [7, 11, 15, 16, 20–22, 24, 25] have shown significant performance improvement. Schema-specific parsers encode parsing states and validation rules at compile time by exploiting schema structures and validation rules to increase processing efficiency at runtime.

However, each generated schema-specific parser must be appropriate to the operating system, compiler, supporting libraries, and hardware on which applications will be run on. The parser must be regenerated and deployed when an XML schema is updated. This is a significant challenge for developing extensible Web services. A Web service is usually a long term agreement that allows consumers to interact with a web service. To address schema updates, service designers typically add new elements to their schema by changing the source code, adding the required business logic and rebuilding the service. However, this approach only works for simple services. Consider for example a large business application requires customizations to fit specific industries, countries, and customers. Exposing such business applications as web services is difficult because they have to be able to be customized over time and these customizations must work for all consumers, even consumers that have made changes to the application. This issue requires that the services have to be designed to be extensible, i.e. Extensible Web services that typically require extensible XML schemas.

Our previous works [26] presents a table-driven XML (TDX) parsing and validation for high-performance Web services. Our TDX technique utilizes a compact tabular representation of schemas and a push-down automaton (PDA) for a single-pass parsing and validation with-

out backtracking. To avoid backtracking on XML elements and attributes defined by XML schema constructs such as unordered sequence of elements (`xs:all`) and attributes (`xs:attribute`), in our later work [25], we extend Backus Naur Form (BNF) with support of *permutation phrase grammar* representation of a schema (thus we call a TDX parser with permutation phrase support a pTDX parser). The permutation phrase grammar is a compact representation of common XML element and attribute permutations that have specific occurrence constraints. The permutation phrase grammar requires a specialized recognizer, which is implemented by a two-stack push-down automaton. In pTDX parser the parsing engine is implemented as a generic table-driven driver that is independent of XML schema at runtime. However, the DFA-based scanner is built from a schema-directed Flex [18] description[1]. The Flex description of the scanner is fed to Flex to generate DFA-based scanner source code in C. As a result, rebuilding of the pTDX parser is a requirement for schema updates. We refer this pTDX parser to Flex-based pTDX parser, or pTDX-fle in short thereafter.

This paper presents an adaptive XML parsing and validating technique that can be used to develop high-performance extensible Web services. Unlike pTDX-flex parser separating a scanner and a parsing engine, this approach implements a table-driven engine that integrates scanning, parsing and validation. This is based on the observation that the parsing table not only drives the parsing, but also can it direct scanning. We call this approach a table-directed pTDX parser, or pTDX-table in short.

The remainder of this paper is organized as follows. We first give a brief description of an adaptive pTDX parser in Section 2. In Section 3, we introduce mapping rules from XML schema components to augmented LL(1) grammar. Construction of modular tables is described in Section 4. Section 5 gives table-driven 2-stack PDA based engine that integrates scanning, parsing and validation. Performance evaluation is given in section 6 and related work is discussed in Section 7. Conclusions are drawn in Section 8.

2. Overview of Adaptive pTDX-based Parser

The architecture of a pTDX-based Web service with swappable modules is shown in Figure 1. The front-end consists of a generic parsing engine, and several modules containing an LL(1) parsing table, a token table, an LL(1) production rule table, tag name table and an action table.

The generic engine, implemented by a push-down automaton (PDA), scans the XML messages for tags and character data (CDATA) , convert each recognized tag into a token, and performs well-formedness checking and validity of the XML content by consulting the parsing table combined with the production rules with semantic actions. Well-formedness and most structural and some XML content types imposed by XML schema are automatically incorporated into the parsing table and production rules, thus they are verified automatically by the parsing engine. Some schema built-in types or derived types from `<xs:restriction>` can not be easily incorporated in to grammar productions. Such types are checked by semantic actions associated with grammar productions. A semantic action function is invoked by the engine to validate if the content of an XML element conforms its constraints. The engine also invokes functions at the back-end through the application action table, which contains function pointers (callbacks) to the application logic for performing application tasks. Both type-checking and application actions are encoded as indices to the entries in the action tables. By using indices, the semantic actions associated with grammar productions do not need to be pre-compiled. This ensures that the modular tables are swappable. In addition, the scanner and parsing engine are implemented as table-driven generic scanner and parser, thus are independent of XML schemas or WSDL descriptions. Therefore, this approach offers a flexible and adaptive mechanism to deal with XML schema updates. When an XML schema updates, the modular tables can be regenerated and populated with no requirement of recompilation of the scanner and the engine.

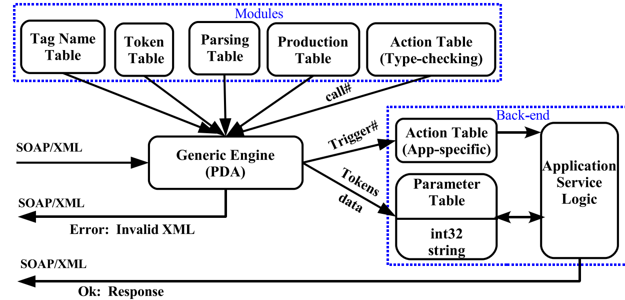

Figure 1. Architecture of a pTDX-based extensible Web service with swappable modules.

The back-end that consists of an application-specific action table and a shared parameter table serves for application service logic. Application logic tasks can be achieved by well-defined APIs that are indexed and stored in an application-specific action table. These application functions are also triggered by semantic actions associated with production rules. Like type-checking function, the use of index tokens to refer application functions ensures the independence of the scanner and the parsing engine on the application logic in the back-end. The parameter data table temporarily holds primitive data passed from the engine that can be directly used by application functions without

[1] Flex is a frequently used automatic generator tool by compiler developers for high-performance scanners.

requirement of deserialization (although the TDX-based approach is capable of deserialization).

3. Mapping XML Schema to Augmented LL(1) Grammar

In this section we briefly describe the mapping rules that define the translations from XML schema components to LL(1) grammar productions (See our previous papers [24–26]). The mapping preserves the structural and semantic constraints imposed by XML schemas. Many types of validation constraints are incorporated in the resulting grammar, for example, the `xs:sequence` order constraint.

We use X to represent any arbitrary Numbers of schema components or particles. The symbol N denotes a non-terminal. The term $N\{m..n\}$, where $2 \leq m \leq n$, represents a string of N's with at least m but no more than n number of occurrences of N's. This is used to represent the extended Context Free Grammar (CFG) for productions generated for occurrence constraints and some facets such as `xs:length`, `xs:minLength`, and `xs:maxLength`.

The mapping operator $\Gamma[\![X]\!]N$ takes a schema component and a designated non-terminal N, and returns a set of LL(1) grammar productions and/or the mapping operator with reduced schema component. A set of grammar productions are generated from the root element by applying the mapping rules until nor more schema component is left in the mapping operator. The symbols N', N'', and N_i represent new nonterminals derived from nonterminal N.

Table 1 lists part of the mapping rules that translate commonly used schema component to augmented LL(1) grammar productions. The first rule maps the most commonly basic format of `xs:element`. The starting tag and closing tag are represented as tokens. Note that this mapping rule does not apply to global elements. We notice that top-level elements and attributes must be mapped using their global name, such that the non-terminal used in the grammar productions are available at global scope in the grammar. That is, for the global elements, the following mapping rules have the highest priority.

$$\Gamma[\![\text{<element name='E' } X\text{/>}]\!]N \quad = \quad \Gamma[\![\text{<element name='E' } X\text{/>}]\!]E$$
$$\Gamma[\![\text{<attribute name='A' } X\text{/>}]\!]N \quad = \quad \Gamma[\![\text{<attribute name='A' } X\text{/>}]\!]A$$

Similarly, `xs:complexType` and `xs:simpleType` must also be mapped using their names (see rules (2-3)). Such mapping rules have the highest priority.

3.1. Mapping Occurrence Constraints

Occurrence constraints `xs:minOccurs` and `xs:maxOccurs` determine the minimum and maximum occurrences of an element or other components. It is straight forward to map an optional occurrence (`minOccurs='0'`) to `epsilon` production, i.e. empty production, and to map unbounded occurrences (`maxOccurs='unbounded'`) to right recursive productions. To reduce the size of the generated grammar, we extended CFG to support arbitrary occurrence between m and n, where where $2 \leq m \leq n$. Rule (6) defines such mapping for occurrence constraints to preserve that the component `<x/>` can appear m to n times. Note that mapping rules for schema facets `xs:minLength` and `xs:maxLength` are defined in a similar way. The occurrence mapping rules significantly reduce the number of productions. For example, an element with `xs:minOccurs` of value 10 and `xs:maxOccurs` value of 1100 may require a thousand LL(1) production rules without using occurrence production.

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. One way to map these unordered elements or attributes is to sum up all possible permutations. However, this may cause the grammar size to combinatorially increase. 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 for eliminating ambiguity introduces more productions. 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, 5]. Thus we extended the LL(1) grammar to support permutation phrase grammar [25]. Rules (7-8) defines permutation phrase mappings. We use $\langle\!\langle a \parallel b \parallel c \rangle\!\rangle$ to represent a permutation phrase.

3.3. Preserving the LL(1) Properties

It is critical to preserve the LL(1) property for constructing the LL(1) parsing table. The mapping rules map element tag names and attribute tag names to unique tokens and makes use of namespace bonded names as nonterminals, and thus ensuring most mapping rules including permutation phrase mapping rules to preserve LL(1) properties.

However, rules (10-11) map `xs:choice` and `xs:union` to the productions with the same nonterminal at right hand side. These mapping rules may generate grammar productions that violate LL(1) property for some

Rule#	Translation		
1	$\Gamma[\![\text{<element name='E' Type='T'>}\ X\ \text{</element>}]\!]N$	=	$\{N \rightarrow \text{bE}\ \tilde{T}\ \text{eE}\}$
2	$\Gamma[\![\text{<complexType name='C'>}\ X\ \text{</complexType>}]\!]N$	=	$\Gamma[\![\text{<complexType name='C'>}\ X\ \text{</complexType>}]\!]C$
3	$\Gamma[\![\text{<simpleType name='S'>}\ X\ \text{</simpleType>}]\!]N$	=	$\Gamma[\![\text{<simpleType name='S'>}\ X\ \text{</simpleType>}]\!]S$
4	$\Gamma[\![\text{<X minOccurs='0'/>}]\!]N$	=	$\{N \rightarrow \epsilon\} \cup \Gamma[\![\text{<X />}]\!]N$
5	$\Gamma[\![\text{<X maxOccurs='unbounded'/>}]\!]N$	=	$\{N \rightarrow N'N, N \rightarrow \epsilon\} \cup \bigcup \Gamma[\![\text{<X />}]\!]N'$
6	$\Gamma[\![\text{<X minOccurs='}m\text{' maxOccurs='}n\text{'/>}]\!]N$	=	$\{N \rightarrow [N']\{m..n\} \cup \Gamma[\![\text{<X />}]\!]N'$
7	$\Gamma[\![\text{<all>}\ X_1 X_2 \ldots X_n\ \text{</all>}]\!]N$	=	$\{N \rightarrow \langle\langle N_1 \parallel N_2 \parallel \cdots \parallel N_n \rangle\rangle\} \cup \bigcup_{i=1}^{n} \Gamma[\![X_i]\!]N_i$
8	$\Gamma[\![\text{<attribute name='A}_1\text{'}\ X\text{/>} \cdots \text{<attribute name='A}_n\text{'}X\text{/>}]\!]N$	=	$\{N \rightarrow \langle\langle N_1 \parallel \cdots \parallel N_n \rangle\rangle\} \cup \bigcup_{i=1}^{n} \Gamma[\![\text{<attribute name='A}_i\text{'}\ X\text{/>}]\!]N_i$
9	$\Gamma[\![\text{<sequence>}\ X_1 X_2 \ldots X_n\ \text{</sequence>}]\!]N$	=	$\{N \rightarrow N_1 N_2 \ldots N_n\} \cup \bigcup_{i=1}^{n} \Gamma[\![N_i]\!]N_i$
10	$\Gamma[\![\text{<choice>}\ X_1 X_2 \ldots X_n\ \text{</choice>}]\!]N$	=	$\bigcup_{i=1}^{n} (N \rightarrow N_i) \cup \bigcup_{i=1}^{n} \Gamma[\![X_i]\!]N_i$
11	$\Gamma[\![\text{<union memberTypes='T}_1\ \text{T}_n \ldots\ \text{T}_n\text{'}\ X\text{/>}]\!]N$	=	$\bigcup_{i=1}^{n} (N \rightarrow N_i) \cup \Gamma[\![X]\!]N$
12	$\Gamma[\![\text{<list itemType='T}\ X\text{/>}]\!]N$	=	$\{N \rightarrow TN', N' \rightarrow TN', N' \rightarrow \epsilon\} \cup \Gamma[\![X]\!]N$

Table 1. Examples rules mapping Schema component to augmented LL(1) grammar productions.

cases (ref. [26] for examples). Such violation can be eliminated by applying *left-factoring* [1]. We perform *left-factoring*for the generated grammar to ensure the LL(1) properties preserved.

4. Constructing Modular Tables

Modular tables play a key role in Table-Driven XML parsing. Modularity offers a flexible and adaptive mechanism for dealing with schema updates. In this section, we describes construction of these modular tables from a set of schemas or WSDL descriptions.

4.1. Token Table

Not only are tokenization of string once and matching on tokens more efficient than repeatedly comparing strings, but also tokenization simplifies process of parsing table and grammar production rules. Tokens are defined by schema element tag names, attribute tag names, schema built-in types such as `xs:boolean` and some facets such as `xs:enumeration`. Element tag names are further classified as starting element tag and closing element tag. Through this paper, we use `bNAME` and `eNAME` to denote the starting element tag `<NAME>` and closing tag `</NAME>` respectively. An attribute tag name is represented by `aNAME`. Similarly, an enumeration value `value='V` is represented by `cV`. Namespace bindings are supported by internal normalization of the token stream to simplify the construction of LL(1) parse table and avoid naming conflicts. A full tag name is a pair of `<ns,name>`. Namespace qualified elements and attributes are translated into normalized tokens according to a namespace mapping table. Thus, identical tag names defined under two different namespace domains are in fact separate tokens.

4.2. Action Table

Some schema built-in types or derived types from `<xs:restriction>` can not be easily incorporated in to grammar productions. Such types are checked by semantic actions associated with grammar productions. Schema built-in types are implemented as libraries. Derived types from `<xs:restriction>` are constructed from schemas as routines for invocation by the engine to perform elements' or attributes' content type checking.

4.3. Parsing Table

The parsing table is a two dimensional array $M[A, a]$, where A is a nonterminal, and a is a terminal. Each entry of the table is either an index that refers to a production rule or an token indicating an error entry[2]. The parsing table are constructed through the use of FIRST and FOLLOW sets [1]. The differences between our augmented LL(1) grammar and the LL(1) grammar in [1] exist in that ours supports occurrence production rules and permutation phrase production rules. The former imposes no affect to calculation of FIRST and FOLLOW sets while the latter does. All of the constituent elements should be treated as the first element when constructing the FIRST and FOLLOW sets because of the unordered property of the permutation phrase production. Thus, the permutation grammar composition symbol is commutative and associative and the FIRST and FOLLOW sets are computed as union of all elements (ref. [25]).

5. Constructing Generic Engine

The engine behaves in two modes: *scanning mode* and *parsiong mode*. In scanning mode, the engine works as a scanner to scan tags and converts recognized tags into tokens. In paring mode, the engine consumes tokens and performs parsing and validation. When the top of the stack is a nonterminal and there is no current token to parse, the engine enters scanning mode. Once a tag name or CDATA is recognized, the engine converts the recognized tag name or CDATA into a token, and enters into parsing mode.

5.1. Scanning Mode

In scanning mode, the engine scans the input string each time to match a specific tag name. Tag names are classified

[2] We say the parsing table entry is either a production or an error entry to simplify the description thereafter.

as *starting element name*, *closing element name*, *attribute name*, and *chracter data*. Once a match is found, the engine converts the tag into a token, and enters into parsing mode.

The parsing table provides information of the specific tag name. From the point view of scanning, the parsing table restricts the possible strings that can be next input string. Each row of the parsing table is indexed by a nonterminal and each column is indexed by a terminal, i.e. a token representing a tag name. To this end, the expected tag name must be among the ones that the nonterminal can generate, and there is exactly one tag name is expected to meet. The engine checks the entry indexed by the nonterminal and the token. If it is a production entry, the engine picks up the token's corresponding tag name and starts to scan. Otherwise, the engine try next token. If no matck is found for all the tokens that correspond to a production entry with the nonterminal, its behavior depends on the type of nonterminal. If it is a regular nonterminal, it indicates an error. If it is a permutation nonterminal, put the nonterminal into auxiliary stack. Typically each row contains few production entries unless for a grammar that consisting large portion of permutation phrases.

- **Scanning Starting Element**

  XML namespaces and attrtibute impose a significant challenge for scanning starting element. A starting element tag, either qualified or not, can not be determined until all the namespaces have been resolved, because a namespace or default name space may be redclared. Unfortunately the namesapces can not be resolved until the starting element tag is closed (the character `'>'` is seen). The strategy is to scan the entire starting element until it is closed. During the scanning, the engine resolves the namespaces, moves all the attributes into another buffer, then converts the starting element to token and enters parsing mode. Because namspaces may be redeclared in a starting element, thus default namespace (if applicable) and namespaces are pushed in a stack.

- **Scanning Closing Element**

  Scanning closing element is much easier than scanning starting element because there is no need to scan namespaces. But it still needs to pop up the namespaces that is possibly pushed by its corresponding starting element.

- **Scanning Attribute**

  Scanning attributes is also easy because all the namespaces have been resolved when its corresponding starting element tag name is scanned. Note that attributes have been moved into a buffer, no longer in the input string.

- **Scanning Character Data**

  Scanning character data is trivial. It continues to scan each character until the dilimiter is met. It sets the current token to a special token that represents character data.

5.2. Parsing Mode

In parsing mode, the engine behaves similarly as a permutation parsing engine. From the point view of parsing, the parsing table encodes a topdown parsing tree for each instance of the XML schema from which the parsing table is constructed. A predictive parsing engine maintains a local stack to track the parser's states. The nonterminal on top of the stack and the current token determines a unique production in the parsing table that needs to be expanded. To be able to parse permutation phrase pgrammar, an auxiliary stack is required to temporialy hold permutation nonterminals that can not be expanded at this point. This indicates that this permutation non terminal does not generate the current symbol. Two flags are also needed for parsing occurrences constraints. 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 token generated in the scanning mode, 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 an occurrence nonterminal**. The engine checks flag F_{max} to see if it exceeds the maximum number of occurrences. It decreases the values of the flags F_{min} and F_{max} by one if not; and it puts the right side hand of the production with the occurrence nonterminal in the parsing table with the nonterminal remaining in the stack. Otherwise it declares an error.

- X **is a permutation nonterminal**. By checking flag F_{max}, the engine verifies if the minimum occurrence constraint is met. The engine moves all symbols of the auxiliary stack (if not empty) into the main stack and consults the entry $M[X, c]$ of the parsing table M. The engine replaces the nonterminal with the right side hand of the production with the left most symbol on top of the stack. If it is an error entry, the symbol is moved into the auxiliary stack from the main stack.

- X **is a regular nonterminal**. The engine also first checks minimum number of occurrences requirement is met. It then checks status of the auxiliary stack.The engine consults the entry $M[X, c]$ of the parsing table

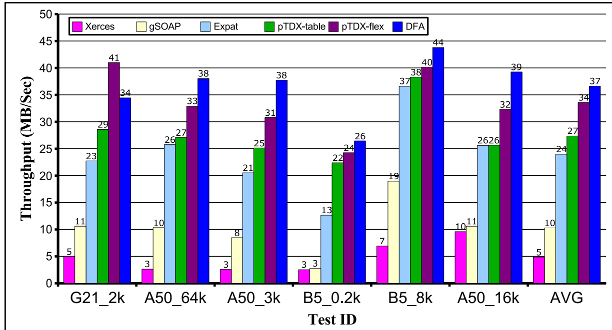

Figure 2. Performance comparison of validating and non-validating parsers

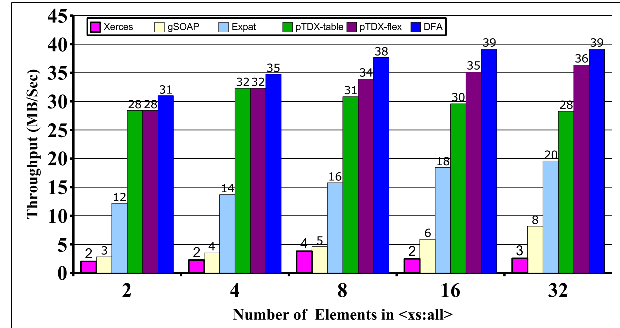

Figure 3. Effect of number of elements in `xs:all` of pTDX Parser.

M, and replaces X by the right hand side of the production, with the left most symbol on top of the stack. If $M[X, c]$ is an error entry, it declares an error. If auxiliary stack is not empty, the engine checks to see if all symbols in 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.

6. Performance Evaluation

We use the same bench mark and test cases as in [25] to measure the performance. File system I/O or network overhead is not measured. Free-ordered elements are placed in a random way. Multiple instances are parsed from separate buffers to avoid any effect possibly caused by high cache hit rates. Time is measured as Wall Clock real time elapsed using system call *gettimeofday()*. 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.

We firstly compared our table-directed adaptive pTDX (pTDX-table) with the Flex-based pTDX (pTDX-flex). We compared two widely used runtime-based parsers, Xerces [2] and expat [17]. We chose Xerces with version of 2.7.0 and Expat with version of 2.0.1 for Linux. We also compared two compile-based parsers, DFA [21] and gSOAP [20] with version of 2.7.3. No application-specific events were triggered in the measurements although TDX offers the capability to trigger events.

Test cases and the measured results are listed in Table 2. From the Figure 2, we can see that the throughput of pTDX-table ranges from 22 MB/Sec to 38 MB/Sec for various message sizes and different numbers of elements in `xs:all` or `xs:attribute`. pTDX-table demonstrates throughput of 27MB/Sec on average of various number fo `xs:all` elements and instance sizes.

Xerces is one of the most widely used toolkits for developing runtime XML parsing and validation by developers. Compared to runtime parsers, pTDX-table is on average 5.4x faster than Xerces parser with validation in SAX mode.

Expat is a non-validating streaming XML parser. It is considered one of the fastest steaming XML parser. It was used without XML namespace support to speed it up. The performance of our pTDX-table with validation is very comparable to non-validating Expat.

gSOAP is a widely used Web services development toolkit for developing high-performance Web services in C/C++. It generates very fast codes for XML parsing, validation and deserialization. pTDX is still 2.7x faster than gSOAP though gSOAP's strength is deserialization.

Compared to pTDX-flex, pTDX-table demontrates a 20% performance drop. This is because pTDX-table uses a table-directed scanner that interpretively scans the input messages. pTDX-flex uses a compiled DFA-based scanner constructed by Flex description. pTDX-table balances the need for adaptive mechanism against performance.

DFA-based parser is the fatest parser measured here. It uses the Flex-based description to constructed a DFA-based XML parser. Like Expat it performs well-formedness check without validation. Like pTDX-flex, It is a compile time DFA-based non-validating parser.

The results in Figure 3 demonstrate that the number of `xs:all` elements does not play a signicant 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 MB/Sec and 36 MB/Sec for pTDX-table validating parser. This indicates that pTDX-table parser has good scalability to the number of elements in `xs:all`.

7. 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

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-flex	pTDX-table	gSOAP	Xerces	DFA	Expat
G21_2k	g.xsd	4021	21	g.xml	2341	41	29	11	5	34	23
A50_64k	a.xsd	3155	50	a 64k.xml	68060	33	27	10	3	38	26
A50_3k	a.xsd	3155	50	a 3k	3016	31	25	8	3	38	21
B5_0.2k	b.xsd	814	5	b.xml	291	24	22	3	3	26	13
B5_8k	b.xsd	814	5	b 8k.xml	8232	40	38	19	7	44	37
A50_16k	a.xsd	4021	50	a50 16k	17156	32	25	11	10	39	26
A2_0.3k	a2.xsd	569	2	a2 0.3k	341	28	28	3	2	31	12
A4_0.4k	a4.xsd	668	4	a4 0.4k	452	32	28	4	2	35	14
A8_0.6k	a8.xsd	881	8	a8 0.6k	678	34	32	5	4	38	16
A16_1k	a16.xsd	1314	16	a16 1k	1124	35	31	6	2	39	18
A32_2k	a32.xsd	2190	32	a32 2k	2036	36	30	8	3	39	20
A32_4k	a32.xsd	2190	32	a32 4k	3886	34	27	10	3	42	22
A32_8k	a32.xsd	2190	32	a32 8k	7584	35	28	10	3	42	25
A32_16k	a32.xsd	2190	32	a32 16k	16826	35	28	17	2	38	25
A32_32k	a32.xsd	2190	32	a32 32k	33462	36	28	11	4	42	26

Table 2. Test Cases and measurements.

parsing [7, 11, 13, 15, 16, 19–22, 24, 26]. 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 [20] 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 [21] 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. [7] 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 namespaces, which is an essential requirement for SOAP compliance.

Cardinality-constraint automata (CCA) [16] 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.

XML Screamer [11] 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 [13]. 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.

TDX [24, 26] provides an integrated approach that combines well-formedness checking, content-type validation and application-specific event by pre-encoding parsing states in a tabular form at compile time and by utilizing an efficient push-down automaton at runtime. However, TDX relies on exponential enumerations of permutation phrases and is therefore not space optimal. pTDX-flex [25] proposes a TDX-based approach that achieves both time and memory space efficientcy by extending 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. The permutation phrase grammar requires a specialized recognizer, which is implemented by a two-stack push-down automaton. However, this Flex-based TDX parser lacks capability of addresing schema updates.

8. Conclusion

In this paper we presented an adaptive table-driven XML parsing and validation technique that can be used to develop extensible high-performance Web services. The adaptive TDX encodes XML parsing states in compact tabular forms by support of permutation phrase grammar. As a result it ensures a memory space efficiency. This adaptive approach uses interpretive scanning at run time by leveraging these tabular forms to improve scanning performance.

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] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient wire formats for high performance computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

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

[6] 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.

[7] 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.

[8] J. Clark and M. Makoto. Relax NG specification, November 2001. http://relaxng.org/spec-20011203.html.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] OMG. XML metadata interchange (XMI) specifications. Available from http://www.omg.org/.

[15] 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.

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

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

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

[19] 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.

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

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

[22] 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.

[23] W3C XML Specification DTD. XML metadata interchange (XMI) specifications. Available from http://www.omg.org/.

[24] 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.

[25] W. Zhang and R. van Engelen. High-performance XML parsing and validation with permutation phrase grammar parsers. In *ICWS '08: Proceedings of the IEEE International Conference on Web Services (ICWS'08)*, pages 286–294, Beijing,China, 2008. IEEE Computer Society.

[26] 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.