Recognizing Matching Patterns for XML Data Using Grammar-based Data Compression Algorithm

Hiroyasu Nishiyama, Tomoya Ohta,
Systems Development Lab.
1099, Ohzenji, Asao,
Kawasaki, Japan
hiroyasu.nishiyama.uq@hitachi.com,
tomoya.ohta.py@hitachi.com

Seiro Tamura, Hideo Munechika
Software Division.
5030, Totsuka,
Yokohama, Japan
seiro.tamura.yk@hitachi.com,
hideo.munechika.gv@hitachi.com

ABSTRACT

XML is a standard format for data exchange and it is well suited to represent internet applications because of its text-based format. However, this flexibility means that it incurs higher data processing overhead than ordinary data formats. In this paper, we propose a high-performance XML processing method using a novel pattern recognition algorithm based on a grammar compression algorithm. In the method, training XML documents are pre-analyzed in order to detect frequently appearing constructs in the document. The extended XML parser uses the results of the pre-analysis to make its parsing faster with speculative input matching. The results of experiments show that the proposed method improves the performance of XML parsing by up to 182% (146% on average) compared with an ordinary SAX parser with namespace processing under the condition that the target XML documents are similar to the pre-analyzed XML documents.

1. INTRODUCTION

Fixed format data was commonly used for data exchange in the past, but the inflexibility of such format sometimes causes problems with data/program updates and data exchanges over a network. The Extensible Markup Language (XML) [5] is widely used as a standard data exchange format to cope with these problems.

XML adopts semi-structured data by representing data with tags and literal texts. Because XML data makes it easier to identify the contents from tags, its data representation is more flexible than those of traditional data formats. A hierarchical data structure can be represented with nested tags. Because of these features, XML has been widely adopted in many areas such as a means of complex software configuration, SOAP [27] and other internet data formats. Its popularity has led to a large demand for faster XML data processing. Because XML data is represented in text format, separation of tags and literal texts and analysis of its hierarchical structure are needed. This processing is performed by an XML parser. Document Object Model (DOM) [26] based methods and Simple API for XML (SAX) [16] based methods are standard tools for parsing XML data.

DOM employs a model that operates on a tree-like structure converted from XML data. In contrast, SAX employs an event-based model that associates user-defined event handler with the elements of input XML data. The event handler is invoked in response to the inputs of the corresponding elements. The time and space efficiencies of the DOM parser are lower than those of the SAX parser because of the overhead of translating an input text into an internal data structure. Programming the SAX parser, in contrast, is more difficult than programming the DOM parser because it does not directly recognize the hierarchical structure of XML data.

In this paper, we describe a fast parsing method of XML data for a SAX-style event-based parser. In the method, training XML documents are pre-analyzed in order to detect frequently appearing constructs in the documents. The XML parser then uses the results of the pre-analysis to make its parsing faster with predictive input matching.

A simple example of XML data is shown in Figure 1. A text between '<' and '>' represents a start tag, and a text between '</' and '>' represents an end tag. A text between a start tag and an end tag represents a literal data. The XML parser must recognize each syntactic element in the document.

The parsing model used by the SAX parser defines events for each syntactic element in an XML document. An event handler is defined to deal with events caused by the recognition of the corresponding syntactic elements. Thus, the parsing model is defined by the event sequences corresponding to the input XML string. A typical SAX parsing process is as follows:

1. Read XML document string from a file or other input device.
2. Break the input string into tokens; validations such as checking input character set are performed in this step.

```
<items>
  <item>
    <name>item0</name><price>100</price>
  </item>
  <item>
    <name>item1</name><price>150</price>
  </item>
  <item>
    <name>item2</name><price>50</price>
  </item>
  <item>
    <name>item3</name><price>800</price>
  </item>
</items>
```

Figure 1: Example XML document

3. Construct argument data for the event handler.
4. Invoke the event handler.

In the ordinary SAX parser architecture, an XML input string is divided into tokens and the event handler is called for each input event. The ordinary parser has runtime overheads such as validating input character set or constructing the parameters of the event handlers.

We can replace the token generation and the validation activities in step 2 with simple matching of the input character sequence if the next input token can be predicted. We can also replace the event handler arguments constructed on-demand with pre-allocated data. These optimizations can lead to faster XML document analysis. Because input XML data are usually similar in each application area, we can expect performance improvements as a result of pre-analyzing typical XML data and recognizing frequently occurring patterns in the documents.

In this paper, we propose a novel algorithm for recognizing frequent patterns that uses a grammar-based data compression method. The algorithm converts training XML documents into hierarchical grammar representation then extracts frequent patterns. For example, we can see from the XML document of Figure 1 that the following repetition pattern

```
<item>
  <name> ... </name><price> ... </price>
</item>
```

follows the grammar rule:

$$A \rightarrow \texttt{\{ <item>...<name>...</name>...</item>... \}*}$$

By using the grammar-based extraction of frequent patterns, we can capture the structure of the original XML data and apply optimizations based on grammar transformations. Since it does not require well-formed XML schema, it is more easy to use than syntax-directed parsing methods based on XML schema.

The architecture of the XML parser is shown in Figure 2. The parser consists of an execution phase and a learning phase. The execution phase contains an ordinary parser and a custom parser. The learning phase analyzes training XML documents to recognize frequent patterns in XML documents. The custom parser is specialized for each pattern recognized as frequent in the learning phase. Initially, the execution phase loads the custom parser before parsing and starts the ordinary parser. If the input character is a start character of the matching pattern, the control is transferred from the ordinary parser to the custom parser. Because the custom parser speculatively analyzes the input XML documents, we can expect performance improvements by using simple automaton-based input matching and some kind of partial evaluation [21]. For example, tokenization of the input string can be replaced by simple character matching against predicted input, and the overhead for allocating the internal data structure for processing SAX events can be avoided. If the input symbol does not match the pattern, the custom parser abort the processing of the symbol and the ordinary parser resumes the execution at the symbol. Our experimental implementation converts the custom parser into a Java program, then converts it into a byte code program.

2. RELATED WORK

The structure of XML documents can be defined in a schema language such as DTD [5], RELAX NG [12] or XML schema [28]. XML schema is currently the most widely used among these schema languages.

There are many studies on schema-directed parsing of XML documents [7, 9, 20, 14]. The methods proposed by [7, 9] generate a customized parser for a specified schema. The method generates a matching DFA from the XML schema and uses it to accelerate the XML document analysis. The method proposed by [20] directly produces analysis code from XML schema. These methods speed up XML document processing by restricting the acceptable document syntax to the grammar defined by XML schema and pre-allocating data that are defined from XML schema. Because these methods generate an analyzer directly from XML schema, they can perform XML schema validation in parallel with document parsing. Unfortunately, XML schema specification is complicated and error prone. For example, Bex et al. reported that 2/3rds of XML schema descriptions they examined did not conform to the XML schema specification[4]. Our method is similar in principle to these methods because it generates a DFA to accelerate XML processing. However, our method does not require a XML schema description that is not always guaranteed to be accessible and correct. This means that our proposed method can create more compact matching pattern using frequently occurring fragment data as learning data. The compact matching pattern leads to smaller matching overhead.

There are studies on automatically inferring XML schema descriptions from XML document instances [10, 1, 3]. The automatically generated descriptions can be fed as input to the schema-directed parsing method. However, producing correct XML schema definitions from example XML documents is difficult in general and requires correct and complete example documents. In contrast, our method can

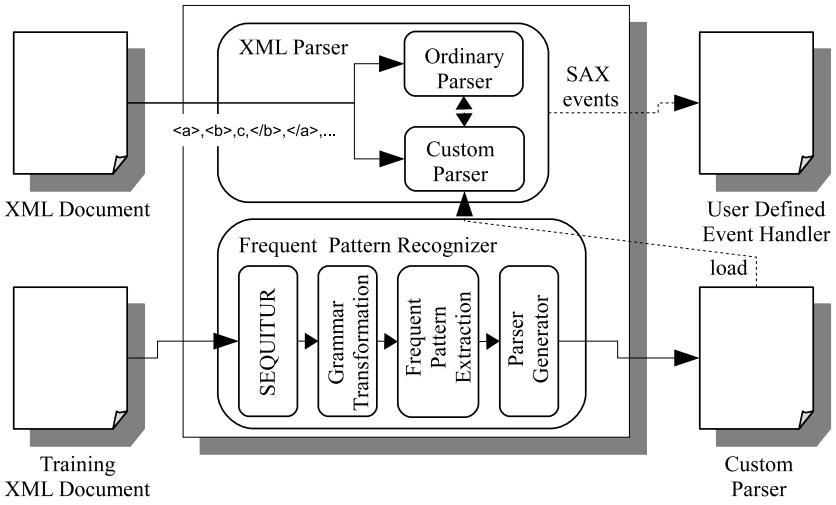


Figure 2: Proposed XML parser architecture

accept XML documents that have no correlation with the learning XML documents because it can switch back to the ordinary XML parser; thus, it does not require completely correct XML schema or fully complying learning XML documents.

SOAP [27] is a protocol for exchanging messages between clients and servers of Web services. It uses XML as its base message format. Messages for the same Web service are usually similar except in their value parts. Thus, there are studies that separate SOAP messages into a fixed part and a variable part. The fixed part is used as a template to accelerate message processing [8]. This idea is suitable for mostly fixed and simple input data but not variable and complex format. There are also studies on dynamically generating a DFA from input SOAP messages [11, 25, 24]. These methods compare the input SOAP message to the DFA created from the previous input in order to reuse the previous message deserialization results. They are similar to our method but incorporate dynamic DFA construction in parallel with input processing. They have an advantage that the DFA can be updated dynamically according to the input data. On the other hand, our method statically constructs the DFA from training XML data to generate a custom parser. Thus, our method can select the globally optimal pattern by statically selecting and optimizing production rules generated from fine-tuned learning XML documents.

There are many other studies on XML acceleration methods. For example, Fast Infoset [19] extends XML to include binary format in order to decrease the overhead for the sake of serializing/deserializing performance. A hardware-based acceleration method that offloads part of XML processing has also been proposed [17].

The grammar-based compression algorithm [13, 18] we base is a lossless data compression algorithm that uses context-free grammar as a means for compressed data representation. The compressed data naturally represents the characteristics of the original input data such as its structure and phrases; it can be used for extracting the features of DNA sequences, musical scores, and other data that have a hierarchical structure. There are many studies on compressing algorithm of XML documents for efficient archiving and querying of large XML data [15, 6]. Compared with these existing work on XML compression algorithm, our method apply transformations on the compressed grammar because our objective is extracting the matching pattern for efficient XML parsing.

3. RECOGNIZING FREQUENT PATTERNS IN XML DOCUMENTS

In order to generate a compact recognition pattern from learning XML documents, the frequently occurring patterns in the input data have to be found. Our frequent pattern recognition algorithm is based on a grammar-based compression algorithm to generate matching automaton. The algorithm converts the input data into a context-free grammar from which the frequent patterns for the matching XML documents are extracted.

3.1 Conversion of input into grammar-based representation

The grammar-based compression algorithm creates a context-free grammar from input data [13]. The input data is encoded in the form of a context-free grammar that is uniquely derived from the input. For this purpose, we use the SEQUITUR algorithm proposed by Nevill-Manning [18]. The SEQUITUR algorithm constructs grammar rules that have the following two properties:

property 1 No pair of adjacent symbols appears more than once in the generated grammar.

property 2 Every production rule except the start rule is used more than once.

The first property ensures that patterns that appear more than once in the input data are merged together into a single grammar rule; and the second property ensures that each

rule is useful. From the point of view of matching pattern recognition, we can consider frequently referenced rules as frequent pattern. The grammar generated by the SEQUITUR algorithm represents the hierarchical structure of the original input symbol sequence. The algorithm operates on a set of symbols Σ and obtains a context-free grammar $G = (V, \Sigma, R, S)$, where V is a finite set of non-terminal symbols, R is a finite set of production rules, and S is the start symbol of G. We represent the data part that varies with each occurrence (such as literal contents of the XML document) with the wildcard matching symbol "#". The set of symbols Σ is defined as $T \cup \{\#\}$, where T is a set of XML tags.

For example, the XML string "<a> *data* " is converted into the following seven symbols:

"<a>", "#", "", "#", "", "#", ""

Note that space and carriage return characters are represented with a "#" symbol.

Step	Input	Generated Rules
1	<items>	N0 →
2	#	N0 → <items>
3	<item>	N0 → <items> #
4	#	N0 → <items> # <item>
...	...	...
i	<item>	N0 → <items> # <item> # <name> ... <item> #
i+1	#	N0 → <items> N1 # <name> ... </item> N1 N1 → # <item>
i+2	<name>	N0 → <items> N2 <name> ... </item> N2 N1 → # <item> N2 → N1 #
i+3	<name>	N0 → <items> N2 <name> ... </item> N2 N2 → # <item> #
...	...	...
j	$	N0 → <items> N8 N8 # <items> N8 → N7 N7 N7 → # <item> # <name> # </name> <price> # </price> # </item>

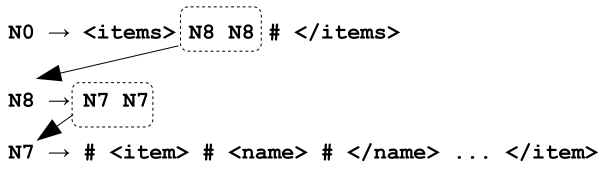
Figure 3: Using the SEQUITUR algorithm to generate production rules

Figure 3 shows the results of applying the SEQUITUR algorithm to the example XML data shown in Figure 1. The "*Input*" column shows the next input symbol, and the "*Generated Rules*" column shows the grammar rules generated before consumption of the next input symbol. First, the empty production rule "N0 →" and the next input symbol "<items>" are provided as the initial state. Since the grammar has the two properties listed above, the symbol "<items>" is appended to the rule N0 generating "N0 → <items>". Then, the input symbols are appended to the production rule "N0" until the second "<item>" is input in step i. In step i, if we add a symbol "<item>" to the production rule N0, the result does not satisfy the first property because the digram "{N1, #}" appears twice on the right-hand side of rule "N0". Thus, the new production rule "N1 → # <item>" is created, and the digrams are replaced with N1 in step i+1. Similarly, in processing the next input symbol "#", the digram "{N1, #}" appears twice in the rules. These digrams are replaced with the new production rule "N2 → N1 #" in step i+2. The grammar rule after replacement does not satisfy the second property because the number of references of the production rule N1 becomes one. Thus, the reference of the production rule N1 on the right-hand side of the rule N2 is expanded to obtain the new production rule "N2 → # <item> #". Repeating this process, we finally obtain the grammar rule shown in step j[1].

We can observe from the grammar rule obtained by the SEQUITUR algorithm that the instance of "#, <item>, ..., </item>" are replaced with one non-terminal symbol N7 and the input data is compressed.

3.2 Recognizing repetitive patterns

The above grammar is equivalent to the input XML document except that the contents of the document is represented with a wildcard symbol "#". Because the SEQUITUR algorithm is a lossless compression, repeated input symbols are represented with a fixed-length symbol sequence with a hierarchical structure. However, a fixed repetitive pattern is not good for predicting inputs; a non-fixed repetitive pattern can match a larger range of input documents with a small matching pattern. Thus, we transform the generated SEQUITUR grammar into one that contains a repetitive grammar in order to enlarge its applicability.



(a) SEQUITUR generated grammar

N0 → <items> N8 # </items>
N8 → N7*
N7 → # <item> # <name> # </name> ... </item>

(b) Grammar after repetitive pattern recognition

Figure 4: Recognition of repetitive patterns from SEQUITUR grammar

The grammars obtained by the method described in the section 3.1 represent the repetitive patterns in the input data by making hierarchical references to a production rule that has a digram of the same symbol on the right-hand side (see Figure 4(a)). We can use the following procedure to recognize grammar patterns that occur more than once in a document.

[1]"$" represents the end of the input.

Recognition of repetitive patterns:

1. For all production rules $r \in R$, apply the following transformations.

   1.1. If the right-hand side of the production rule r is a digram of the same symbol represented as "$A \rightarrow \alpha\ \alpha$", replace the rule with "$A \rightarrow \alpha*$", where $A \in V$ and $\alpha \in V \cup \Sigma$.

   1.2. If the right-hand side of the production rule r contains a digram of the same symbol represented as "$A \rightarrow \ldots B\ B \ldots$" and $B \in V$ is a rule of the form "$B \rightarrow C*$", replace the digram with one B; i.e., replace A with "$A \rightarrow \ldots B \ldots$".

   1.3. If the right-hand side of the production rule r is repetitive represented as "$A \rightarrow B*$" and B is also a repetitive rule of the form "$B \rightarrow \alpha*$", replace the production rule of A by "$A \rightarrow \alpha*$", where $A, B \in V$ and $\alpha \in V \cup \Sigma$.

Figure 4(b) shows the grammar for the example XML document of Figure 1 after recognizing the repetition pattern. Accordingly, the right-hand side of the production rule of `N8` is translated into a repetitive pattern.

If the repetitive structure is nested and the number of inner repetitions is different for each occurrence of the inner repetition, they are recognized as different production rules. Thus, the repetition within such a structure cannot share the same pattern. To cope with this situation, we add a "property" to terminate recognition of consecutive series of the identical symbol; namely, we recognize up to 4 consecutive series of the identical symbol and ignore the symbols after that[2]. Although the resulting grammar does not have the same XML structure as the input data, the pattern recognition step treats the input as if it had the same repetitive structure.

property 3 A consecutive series of identical symbols does not repeat for more than four times in the generated grammar.

Figure 5 shows an example of a nested repetitive structure. The grammar shown in Figure 5(b) is generated from the input data shown in Figure 5(a) if the termination property is not used. In this case, the production rule `N2` represents repetition of the sequence "`<item> # </item>`". By adding the termination property, we can obtain the grammar shown in Figure 5(c). This property recognizes repetitions of the sequence "`<list>` ... `</list>`" that includes repetitions of the sequence "`<item>` ... `</item>`".

3.3 Transforming production rules

The matching pattern obtained in the learning phase is used to find a sub-string that matches the predicted symbol sequences. If we allow any sequence of symbols to be used as the start symbol for matching, the opportunities for successful matching will increase. However, in doing so, the input string has to be checked at many points in an ordinary XML parser, and this increases the parsing overhead. We restrict the start of the matching pattern to tokens corresponding to the start tags. Thus, we only need to change processing of the start tags in the ordinary parser. This does not cause a problem because the start and the end tags always appear in pairs in XML documents and the wildcard matching character can't be used as the start symbol of the matching pattern.

[2]This threshold is chosen because the SEQUITUR algorithm creates hierarchical reference of production rules after 4 consecutive series of an identical symbol.

```
<list>
  <item> # </item> <item> # </item>
  <item> # </item> <item> # </item> // repeat 4 times
</list>
<list>
  <item> # </item> <item> # </item> <item> # </item>
  <item> # </item> <item> # </item> // repeat 5 times
</list>
```

(a) Training XML document

```
S → N3 N4 N3 N1 N4
N1 → <item> # </item>
N2 → N1 N1
N3 → <list> # N2 N2
N4 → # </list> #
```

(b) Generated rule without termination property 3

```
S → N3 N3
N1 → <item> # </item>
N2 → N1 N1
N3 → <list> # N2 N2 # </list>
```

(c) Generated rule with termination property 3

Figure 5: Effect of adding termination property 3 to SEQUITUR grammar

Matching patterns are created from the production rules generated from training XML documents. The SEQUITUR-based rule generation algorithm does not guarantee that the first symbols of the production rules are the start tags. Thus, we transform the learned production rules to make them start with a start tag. The transformation processes of the production rules for repetitive rules and non-repetitive rules are different.

Transforming repetitive rules:

We use a transformation method called grammar peeling to make repetitive rules start with a valid start symbol. The grammar peeling shifts the right-hand side symbols of the repetitive rule as follows:

1. For each repetitive rule $r \in R$ of the form "$A \rightarrow \{\chi_1\ \chi_2 \cdots\ \chi_n\}$" apply the following transformations if $\chi_1 \ldots \chi_i$ are not valid start symbols (i.e., not start tags) where $A \in V$ and $\chi_i \in V \cup \Sigma$,

   1.1. Create following three rules:

```
N0 → <items> N1 # </items>
N1 → N2*
N2 → # <item> # <name> # </name> ...</item>
```

(a) Before transformation

```
N0  → <items> N1 # </items>
N1 → # N2 <item> # <name> ...</item>
N2 → N3*
N3 → <item> # <name> # </name> ...</item>
```

(b) After transformation

Figure 6: Transforming repetitive rules

- $B \rightarrow \chi_1 \ldots \chi_i\ C\ \chi_{i+1} \ldots \chi_n$
- $C \rightarrow D*$
- $D \rightarrow \chi_{i+1} \ldots \chi_n\ \chi_1 \ldots \chi_i$

where $A, B, C \in V$ and $\chi_i \in V \cup \Sigma$,

1.2. Replace all occurrences of `A` with `B`.

For example, in the production rules after the repetition recognition shown in Figure 6(a), the start symbol "`#`" of the production rule `N2` is not a valid start symbol of a matching pattern. Thus, the repetitive production rule `N1` can't be used as a matching rule. By applying the grammar peeling to this production rule, we can obtain the grammar shown in Figure 6(b). After grammar peeling, the first symbol of the new production rule `N3` becomes "`<item>`". The resulting repetitive rule `N3` can be used as a matching rule.

Transforming non-repetitive rules:

For each non-repetitive rule, we add a symbol before the reference of the rule by using the following grammar transformation:

1. For each production rule $r \in R$ of the form "$A \rightarrow \alpha \ldots$" that starts with an invalid start symbol α, where $A \in V$ and $\alpha \in V \cup \Sigma$, apply the following transformation,

   1.1. For each reference of the non-terminal symbol A from a rule "$B \rightarrow \chi_1 \ldots \chi_n\ A \ldots$", create a new production rule "$C \rightarrow last(\chi_1 \ldots \chi_n)\ A \ldots$", where $B, C \in V$, $\beta, \chi \in V \cup \Sigma$, and $last(\chi_1 \ldots \chi_n)$ means the last symbol that is valid for the start of the production rule.

In the example shown in Figure 7(a), the first symbol of the non-terminal symbol `N1` is a wildcard symbol "`#`" and is not a valid start symbol. The symbol `N1` is referenced from the right-hand side of the production of `N0`. Since $last($"`<list>`"$)$ = "`<list>`" and $last($"`<list> N1 </item>`"$)$ = "`</item>`", we create new production rules `N2` and `N3` from the production rule `N1`. We cannot simply delete the original rule, in this case `N1`, after this transformation because they may be referenced from other production rules. However, the following rule selection process ignores them in order to ensure that no invalid symbol appears at the front of the generated matching pattern.

```
N0 → <list> N1 </item> N1 <tail> # </tail>
      </item> </list>
N1 → # <item> #
```

(a) Before transformation

```
N0 → <list> N1 </item> N1 <tail> # </tail>
      </item> </list>
N1 → # <item> #
N2 → <list> N1
N3 → </item> N1
```

(b) After transformation

Figure 7: Transforming non-repetition rules

3.4 Rule selection

Generating a custom parser for all production rules generated from the previous method increases the applicability of the parser. However, if the matching input is short, the custom parser may not perform as well as it should because of the overhead for invoking it. Thus, we select production rules that are highly likely to improve performance from ones obtained in the learning phase. This selection considers the following three criteria: (1) the structure of the production rule, (2) the expected rule length, and (3) the frequency of occurrences. We prefer repetitive rules to non-repetitive ones. The expected rule length is calculated by recursively accumulating the rule length. We assume a fixed number of iterations for repetitive rules[3]. The rule selection process is as follows:

Rule selection:

1. Order rules with the following criteria:

   repetitive-rule > rule length > frequency of rule occurrences

2. Select rules according to the following criteria:

   2.1. Ignore following rules:

   - the start rule,
   - rules that start with an invalid symbol,
   - repetitive-rules,
   - surplus rules that is created by grammar peeling.

   2.2. ignore rules whose length is less than half of the maximum rule length of already selected rules,

   2.3. ignore rules that are included in the already selected rules.

Applying the rule selection process described above to the production rules after the rule transformation shown in Figure 6(b) results in only `N2` being selected as a matching rule because `N0` is a start rule, `N1` is a surplus rule created by grammar peeling, and `N3` is in the body of a repetitive rule.

[3]Our current implementation assumes that repetitive-rules iterate 100 times.

4. GENERATING CUSTOM PARSER

Above, we described a method for detecting frequently occurring patterns in XML documents that is based on grammar-based compression algorithm. A custom parser is generated from the obtained pattern and is used after recognition of the first input symbol to improve parsing performance using DFA-based character matching.

4.1 Generating recognition pattern

We generate a matching automaton from the generated production rules. First, inlining of production rules is applied to the rules. Because the SEQUITUR generated grammar do not include recursive rules, the inlining terminates in a finite number of applications. Figure 8 shows the grammar obtained by inlining the rule `N2`. Each production rule represents a sub-string of the input XML document and is equivalent to a regular language that consists of concatenation, alternation, and Kleene closure. Thus, we can generate an NFA from inlined production rules to make a sub-string matching of an XML document. The NFA is constructed by combining NFAs that accept each production rules with ϵ transitions. Figure 9 shows the generated NFA from the grammar of Figure 8.

```
N2 → { <item> # <name> # </name> ...</item> # }*
```

Figure 8: Rule after inlining

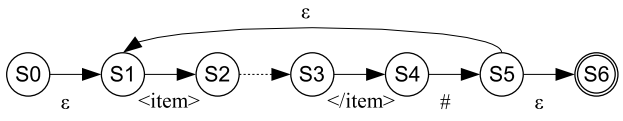


Figure 9: Generated NFA

Because the NFA contains ambiguous transitions, it is not efficient to use it for actual matching. Thus, we convert the NFA into DFA. This conversion is done by using a conventional DFA construction and compaction algorithm.

4.2 Generating and invoking the custom parser

The custom parser in Java class format is generated from the DFA representation of the matching pattern. The extended (i.e., ordinary+custom) parser loads the corresponding custom parser when parsing XML documents. If an input symbol matches the first symbol of the matching pattern, the parser invokes the custom parser. As stated before, this matching is only performed against the start tag in order to decrease the overhead for transferring control from the ordinary parser to the custom parser. The tag checking is performed by using an initial state transition table. The table consists of transition tag strings and target state numbers.

Figure 10 shows the code fragment for invoking the custom parser. The matching function `match` at (a) checks whether the next symbol is the first one of the matching pattern. If it is, it returns the next state number. If the next state number is obtained, the parser invokes the custom parser with the state number at (b). If the input does not match the first symbol of the matching pattern, ordinary parser processing is resumed at (c).

```
if((s = match(pattern selection table) > 0)) { // (a)
  customParser(..., s); // (b)
} else {
  // (c) ordinary parser processing
}
```

Figure 10: Calling code skeleton of the custom parser

```
public class CustomParser {
  extends AbstractCustomParser {
  public int parser(..., int start_id) {
    int next_state = start_id;
  loop:
    while(true) {
      switch(next_state) {
      case S0:
        if(match pattern) {
          // action code
          next_state = next state number;  // (a)
          break; // (b)
        } else {
          // recovery code
        }
      case S1: // (c)
        ...
      }
    }
  }
}
```

Figure 11: Code skeleton of the custom parser

4.3 Code generation for the custom parser

The code fragment of a custom parser is shown in Figure 11. One DFA state corresponds to one case block of the switch statement. The "*match pattern*" validates an input symbol with the state transition symbol to the next state. If the input symbol matches the transition condition of the current state, the corresponding event handler is called at "*action code*" and the next state is set. The "*recovery code*" is called when the input validation fails. This code normally returns control to the ordinary parser. However, if it is in the middle of processing a start tag with attribute values, the custom parser continues the input processing until the end of the start tag; then it returns to the ordinary parser. This is because the transition between the ordinary parser and the custom parser is performed on a tag by tag basis.

If state transitions continuously occur without branches, we can eliminate state transition instructions by placing codes corresponding to the next state after the code for the first state. Thus, we traverse the generated DFA in a depth-first manner and eliminate the transition instructions between states that do not have other branches. For example, we can eliminate code (a) and (b) from the code shown in Figure 11 if the "*next state number*" at (a) and the following state number *S1* at (c) are the same. If there is no transition to label *S1*, the label at (c) can also be eliminated.

5. EVALUATION

We evaluated an extended (i.e., ordinary+custom) parser based on Xerces XML parser[4] [2]. The evaluation used the following environments:

CPU:	Xeon[5]E5310 1.6GHz×4
Memory:	64GB
OS:	CentOS 4.2
Java[6]VM:	JDK1.6.0_06 64-bit server VM
VM option:	-server -Xms1024M -Xmx1024M

Table 1: XML benchmark documents

name	input	train.	description
periodic	114KB	9.3KB	Periodic table of elements.
soap_list	131KB	1.4KB	SOAP message.
much_ado	197KB	2.4KB	Shakespeare play.
inv1000	924KB	9.3KB	Invoice.
weblog	2.9MB	1.7KB	Web server access log.

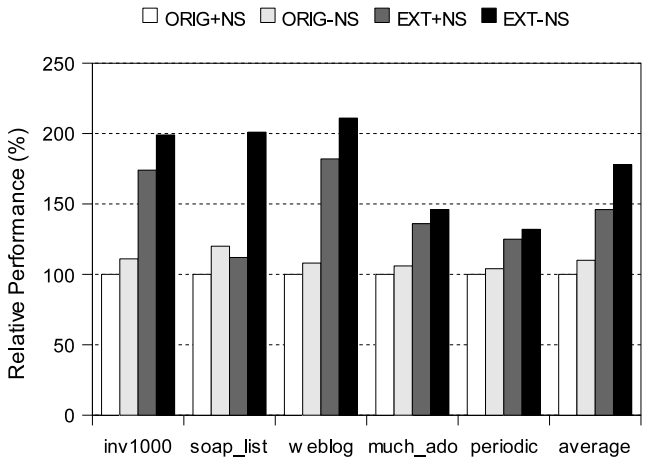


Figure 12: XML processing performance of baseline and extended parsers

The evaluation used 5 XML documents (Table 1) from XML-Bench [22] and XMLTest [23]. The evaluation was performed under the following conditions:

- The processing time of 300 iterations was measured after warm up of 300 iterations.
- The learning data were manually extracted from the first 5 repetitions appearing in the original XML documents.
- No schema validation was performed.
- The empty event handler was used.

[4]We used xerces-J 2.7.1 as the baseline XML parser.

[4]Intel Xeon is a trademark or a registered trademark of Intel Corporation in the United States and other countries.

[5]Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

We evaluated the baseline XML parser (**ORIG**) and the proposed XML parser (**EXT**) with namespace processing (**+NS**) and without namespace processing (**-NS**). The processing times were averages of 10 benchmark runs. Figure 12 shows the relative performance compared to the baseline XML parser with namespace processing.

With namespace processing, the extended parser was faster than the baseline by up to 182% for the weblog benchmark and 146% on average[7]. Without namespace processing, the extended parser was up to 211% (178% on average) faster than the baseline XML parser. The difference in the performance between the case without namespace processing and the case with namespace processing is caused by overhead for recognizing XML namespaces.

Table 2: Rule, DFA and custom parser characteristics of benchmarks

input XML document	generated rules	selected rules	DFA states	parser size
periodic	24	3	169	19,589
soap_list	7	1	41	9,010
much_ado	23	5	46	9,742
inv1000	48	1	89	14,474
weblog	3	1	45	9,197

The characteristics of the learning phase are summarized in Table 2. The extended parser selected only small rules as matching patterns. The selected production rules are maximal repetition patterns for all training data except the periodic benchmark. The periodic benchmark includes many alternating structures. Thus, they are recognized as different production rules, and they require many switchings between the ordinary parser and the customized parser. However, the alternating structures are all selected and the common part is merged in the generated automaton. This is the cause of the relatively small performance improvements for the periodic benchmark. The size of the custom parser in the class file is mostly in proportion to the number of DFA states.

Figure 13 plots the change in the percentage of SAX events covered by the custom parser versus the number of selected grammar rules. The percentage of SAX events covered by the custom parser is very high even when the number of selected rules is one. This means that the extended parser can appropriately select matching patterns. The event cover ratio saturates when the number of rules equals to two.

6. CONCLUSION AND FUTURE WORK

We developed an extended XML parser that increases processing speed by speculatively analyzing input data with a pre-learned input pattern. It extracts matching patterns as grammar rules from the learning XML documents to produce a customized parser. The customized parser is invoked from the ordinary parser to improve the processing performance on XML documents having the pattern learned by the custom parser.

[7]The relatively small performance improvement on the soap_list benchmark seems to be caused by the larger namespace processing overhead.

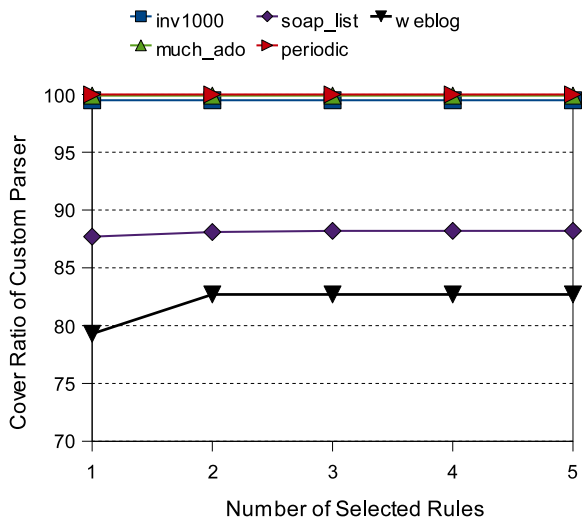


Figure 13: Cover ratio of SAX events by custom parser

The extended parser improved the SAX processing performance with namespace processing by 146% on average and up to a maximum of 182% when the input XML documents were similar in structure to the learning XML documents.

The remaining work of this research will be to enable learning of XML documents that contain alternating elements in a repetitive structure. For example, our method cannot produce optimal matching patterns for some benchmarks. This causes transitions between the ordinary parser and the custom parser when the input consists of alternating structures.

Another issue regards probability-based grammar construction and selection. Our method currently does not use the occurrence probability of the grammar rules in the learning XML documents. We could use such probability to select more appropriate patterns.

7. REFERENCES

[1] N. Abu-Ghazaleh and M. Lewis. Differential Deserialization for Optimized SOAP Performance. In *Proceedings of the SC2005 Conference on High Performance Networking and Computing*, 2005.

[2] Apache Software Foundation. Xerces. `http://xerces.apache.org/`, 2004.

[3] G. Bex, , F. Neven, and S. Vansummeren. Infering XML Schema Definitions from XML Data. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.

[4] G. Bex, F. Neven, and J. Bussche. DTDs versus XML Schema: A Practical Study. In *Proceedings of the 7th International Workshop on the Web and Databases*, pages 79–84, 2004.

[5] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1. `http://www.w3.org/TR/2004/REC-xml11-20040204/`, 2004.

[6] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving Scientific Data. *ACM Transactions on Database Systems*, 29(1), 2004.

[7] K. Chiu and W. Lu. A Compiler-Based Approach to Schema-Specific XML Parsing. In *Proceedings of the 1st International Workshop on High Performance XML Processing*, 2004.

[8] J. Clark. Trang Multi-format Schema Converter Based on RELAX NG. `http://www.thaiopensource.com/relaxng/trang.html`, 2003.

[9] R. Engelen. Constructing Finite State Automata for High Performance XML Web Services. In *Proceedings of the International Symposium on Web Services*, 2004.

[10] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[11] J. Hegewald, F. Naumann, and M. Weis. XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In *Proceedings of the 22nd International Conference on Data Engineering Workshop*, 2004.

[12] J. Clark and M. Murata. RELAX NG Specification. `http://relaxng.org/`, 2001.

[13] J. Kieffer and E. Yang. Grammer-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(4):737–754, 2000.

[14] M. 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 *Proceedings of the 15th International World Wide Web Conference*, pages 93–102, 2004.

[15] H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[16] D. Meggison. SAX: Simple API for XML. `http://www.saxproject.org/`, 2004.

[17] B. Nag. Acceleration Techniques for XML Processors. In *Proceedings of XML 2004 Conderence*, 2004.

[18] C. Nevill-Manning and I. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, pages 67–82, 1997.

[19] P. Sandoz and A. Triglia and S. Pericas-Geertsen. Fast Infoset. `http://java.sun.com/developer/technicalArticles/xml/fastinfoset/`, 2004.

[20] E. Perkins, M. Matsa, M. Kostoulas, A. Heifets, and N. Mendelsohn. Generation of Efficient Parsers Through Direct Compilation of XML Schema Grammars. *IBM Systems Journal*, 45(2), 2006.

[21] U. Schultz, L. Lawall, and C. Consel. Automatic Program Specialization for Java. *ACM Transactions on Programming Languages and Systems*, 24(4), 2003.

[22] Sosnoski Software Solutions. XMLBench Document Model Benchmark.

http://www.sosnoski.com/opensrc/xmlbench/, 2006.

[23] Sun Microsystems. Web Services Code Samples. http://java.sun.com/developer/codesamples/webservices.html#Performance, 2006.

[24] T. Suzumura, T. Takase, M. Takase, and M. Tatsubori. Optimizing Web Service by Differential Deserialization. In *Proceedings of the International Conference on Web Services*, 2005.

[25] Y. Takeuchi, T. Okamoto, K. Yokoyama, and S. Matsuda. A Differential-analysis Approach for Improving SOAP Processing Performance. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, 2005.

[26] W3C. Document Object Model (DOM) Technical Reports. http://www.w3.org/DOM/DOMTR, 2004.

[27] W3C. SOAP Specifications. http://www.w3.org/TR/soap/, 2004.

[28] W3C. XML Schema. http://www.w3.org/XML/Schema/, 2006.