Optimizing Differential XML Processing by Leveraging Schema and Statistics

Toyotaro Suzumura, Satoshi Makino, and Naohiko Uramoto

Tokyo Research Laboratory, IBM Research
1623-14 Shimo-tsuruma Yamato-shi Kanagawa-ken, Japan, 242-8502
{toyo, mak0702, uramoto}@jp.ibm.com

Abstract. XML fills a critical role in many software infrastructures such as SOA (Service-Oriented Architecture), Web Services, and Grid Computing. In this paper, we propose a high performance XML parser used as a fundamental component to increase the viability of such infrastructures even for mission-critical business applications. We previously proposed an XML parser based on the notion of differential processing under the hypothesis that XML documents are similar to each other, and in this paper we enhance this approach to achieve higher performance by leveraging static information as well as dynamic information. XML schema languages can represent the static information that is used for optimizing the inside state transitions. Meanwhile, statistics for a set of instance documents are used as dynamic information. These two approaches can be used in complementary ways. Our experimental results show that each of the proposed optimization techniques is effective and the combination of multiple optimizations is especially effective, resulting in a 73.2% performance improvement compared to our earlier work.

Keywords: XML, Web Services, XML Schema, Statistics.

1 Introduction

Recently XML (Extensible Markup Language) has come to be widely used in a variety of software infrastructures such as SOA (Service-Oriented Architecture), Web Services, and Grid Computing. The language itself is used in various ways such as for protocols, for data formats, for interface definitions, etc. Even though the nature of the language gives us various advantages such as interoperability and self-description, its redundancy of expression leads to some performance disadvantages compared to proprietary binary data. Many methods [1][2][3] have been proposed for enhancing its performance, and these efforts are critical research areas in order to achieve the same or better performance and replace existing legacy infrastructures with XML-based infrastructures such as Web services. Our previous work [1] proposed an approach for realizing high performance XML processing by introducing the notion of differential processing. In this paper, we present an approach for improving the performance of our earlier approach by leveraging the knowledge given by XML schema and statistical information about instance documents.

The rest of the paper is organized as follows. In Section 2, we will explore optimized XML processing by reviewing our previous work. Section 3 describes the

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 264 – 276, 2006.

contributions of this paper, which enhance our previous work by leveraging two optimization techniques. Section 4 describes the optimization approach with schema languages. Section 5 describes the statistics-based optimization approach. Section 6 describes a performance evaluation. In Section 7, we introduce some related work using approaches with XML Schemas, and finally we conclude this paper with Section 8.

2 Improved Performance in Our Previous Work

In [1], we proposed a new approach called "Deltarser" to improve the performance of an XML parser based on the fundamental characteristics of Web services [1]. Next we give an overview of Deltarser and then discuss some of its limitations.

2.1 Overview of Deltarser

Deltarser is designed for efficiently processing XML documents similar to previously processed XML documents. The efficiency for similar documents is relevant in situations like Web service application servers, where middleware needs to process many similar documents generated by other middleware. In addition, the parsing of Deltarser is "safe" in the sense that it checks the well-formedness of the processed documents. From the viewpoint of users, Deltarser looks just like an XML parser implementation and has the same functionality as normal XML parsers such as the Apache Xerces implementation. The key ideas and technologies of Deltarser are summarized as follows: The main action of Deltarser is byte-level comparison, which is much faster than actual parsing. When feeding the actual XML document to the state machine described next, we have only to compare the byte sequence of each state and the incoming document. For efficiently remembering and comparing previously-processed documents, it remembers the byte sequence of the processed documents in a DFA (Deterministic Finite Automaton) structure. Each state transition in the DFA has a part of a byte sequence and its resultant parse event. It partially processes XML parsing only the parts that differ from the previously-processed documents. Each state of the DFA preserves a processing context required to parse the following byte sequences. It reliably checks the well-formedness of the incoming XML documents even though it does not analyze the full XML syntax of those documents. Deltarser's partial XML processing for differences checks whether or not the entire XML document is well-formed. It retains some contextual information needed for that processing. The experiments in [1] show the promising performance improvements such as being 106% faster than Apache Xerces [14] in a server-side use-case scenario and 126% faster in a client-side use-case scenario.

2.2 Observed Performance Limitations

Although our approach has promising performance benefits, some limitations were observed in our experiments with various performance evaluations. These limitations are twofold:

- **Startup overhead of creating state transitions.** Although byte sequence matching is much faster than regular XML processing, the initial cost of preparing an automaton cannot be ignored. The experiment conducted in [1] shows the comparison of Deltarser and other existing parsers such as open source fast XML parser, Piccolo, and a well-known XML parser, Xerces, but Deltarser is slower until 25 documents are processed. This initial overhead can be ignored for most XML applications, especially for long-running Web services that process large numbers of SOAP requests. However, considering the use of the differential processing approach for general purposes, it would be best if we could eliminate this initial overhead.
- **Runtime overhead for a series of state transitions.** An automaton is constructed by adding state transitions with a granularity corresponding to each SAX event. The graph shown in Figure 1 shows the average time for processing a 64-KB document while changing the number of constituent state transitions within the automaton. Obviously, as the number of state transition increases, the processing time increases. The line graph shows a case in which whitespace is represented as one state transition, and the number of state transitions is about 12,000, and it takes 3.25 ms for byte sequence matching.

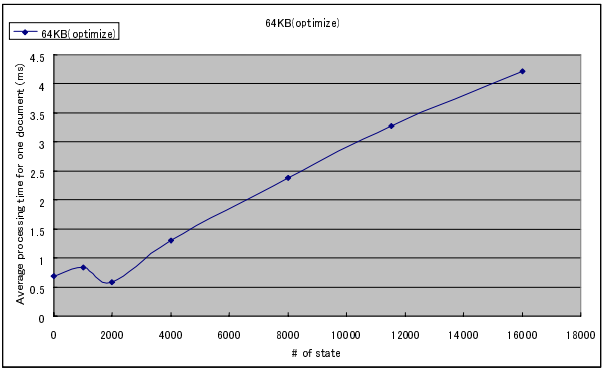


Fig. 1. Average processing time while changing the number of state transitions

When the whitespace is integrated with the other state transitions, the number decreases to about 8,000, and it takes 2.4 ms for byte sequence matching, which is 30% faster. As shown in this experiment, when there are fewer state transitions, there is less overhead incurred in differential processing. However this does not necessarily mean that the number of state transitions should be as small as possible. If the number is small and the automaton is compressed, then the probability of mismatching during the byte sequence matching will be high, and that results in the creation of many more new state transitions than needed for byte sequence matching.

3 Differential XML Parser with Optimized Automaton

To reduce the overhead mentioned in the previous section, we propose an approach that optimizes the internal automaton by leveraging some knowledge known before runtime as well as some runtime information. More precisely we leverage the following information:

- **XML Schema**
  An XML schema provides static information available before execution starts. The schema provides structural information as well as possible data values. By utilizing the schema information, it is possible to create an automaton even before the parser sees the target instance documents to be processed, and this results in the reduction of the startup overhead from the previous section. In addition, this can also be used for reducing the runtime overhead, since the schema gives the parser hints to design the shape of the state transitions.
- **Statistics**
  Statistical information is dynamic information obtained at runtime. By aggregating a set of instance documents, the parser obtains knowledge about the structural information as well as specific data values. This knowledge is similar to the kind of information provided by the XML schema, but the statistical approach can be used even if there is no schema, and it also provides more precise information than given by a schema.

These two approaches can be used in a complementary fashion. XML schema languages do not provide perfect information about instance documents. There are two reasons. One is because the languages themselves have some limitations in expressing all structures. These limitations are intentional to avoid too much complexity. The second reason is that users cannot always write a perfect schema before they have sufficient knowledge of the concrete documents. In addition, there may be situations where no XML schema is provided. Therefore, when the XML schema does not provide sufficient information, we can use the statistical approach. It may take some time to aggregate sufficient statistical data, but this can be more precise in expressing a class of instance documents compared to using an XML schema. In the following sections, we describe these two optimization approaches in more detail.

4 Optimization by XML Schema Language

This section describes an optimization approach leveraging the XML schema. Before describing the optimization approach, let us give a quick overview of the XML schema languages. An XML schema language is a formalization of the constraints, expressed as rules or as a model of a structure, that apply to a class of XML documents. Many schema languages have been proposed, such as XML DTD, W3C XML Schema, and RELAX NG. W3C XML Schema is the most common language and is widely used. Our approach does not depend on the specific XML schema language, but here we focus on using the set of schema representations used in W3C XML Schema.

The overall algorithm for optimizing state transitions is shown in Figure 1. First, our XML parser extracts one element and judges whether or not the element has attributes based on the schema information. When attributes exist, the parser fetches the attribute information from the schema. If any attribute is specified as a fixed value, the parser then executes Optimization 1 (Opt. 1), which is described in Section 4.1. Otherwise, it

proceeds to the regular processing, which means that regular automaton creation is occurring.

When the element has no attributes, or after the attribute processing has been finished, the parser determines whether or not the element has child elements. If the answer is “no”, processing continues with the next condition to check whether or not that element has default values. If the answer is “yes”, the processing continues with Opt. 2 as described in Section 4.1. If the answer is “no”, the regular processing will be executed.

In the branch where the element has child elements, the parser fetches the structural information from the schema. Then it branches on one of three paths depending on the type of the complex type. If the complex type is xsd:choice, then processing proceeds with Opt. 3 as described in Section 4.2. If the complex type is xsd:sequence, then processing checks whether the *maxOccurs* attribute equals to the *minOccurs* attribute, which leads to Opt. 4 as described in Section 4.2. If the schema provides a specific ordering, then processing continues with Opt. 5. Next we will describe each optimizations in detail.

Table 1. Excerpts from XML Schemas

Schema (1)	Schema (2)	Schema (3)
<xsd:simpleType name=“schemaRecommendations”> <xsd:restriction base=“xsd:string”> <xsd:enumeration value=”A” /> <xsd:enumeration value=”B”> </xsd:restriction> </xsd:simpleType>	<xsd:element name=”X”> <xsd:complexType> <xsd:sequence> <xsd:elementname=”A” xsd:type=”xsd:int”/> <xsd:elementname=”B” xsd:type=”xsd:int”/> </xsd:sequence> </xsd:complexType> </xsd:element>	<xsd:complexType name="all"> <xsd:all> <xsd:element ref="A" /> <xsd:element ref="B" /> <xsd:element ref="C" /> </xsd:all> </xsd:complexType>

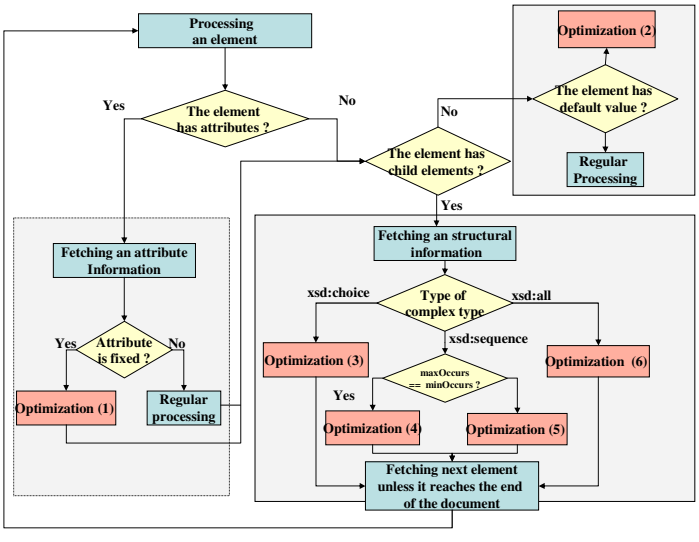


Fig. 2. Optimization Algorithms using XML Schema

4.1 Fixed Value or Enumerations

The xsd:enumeration is a facet that allows for definition of a list of possible values for the value of a specified data type. The example schema below shows a facet constraining the values. This enumerated information can be used for automaton creation.

I. Attribute (Opt. 1)
A fixed value of an attribute value is specified in the following schema fragment:

```
<xsd:attribute name="year" type="xsd:date" fixed="2004"/>
```

This allows a user to create one integrated state transition rather than creating a list of separate transitions.

II. Text Node (Opt. 2)
An element tag in the XML schema allows a user to write a fixed attribute. An example would be *<xsd:element name="name" type="xsd:string" fixed="IBM" />* and a sample instance document would be *<name> IBM </name>*. Using this schema, it is better to construct one integrated state transition containing one string, i.e. "*<name> IBM </name>*" rather than having three state transitions, "*<name>*", "*IBM*", and "*</name>*".

4.2 Compositors

Compositors such as xsd:choice, xsd:sequence, or xsd:all give hints to formulate optimized state transitions. However, such optimization tends to generate redundant state transitions that will never be used at runtime. For example, when creating all potential state transitions in advance using xsd:choice and xsd:all, the number of state transitions will be too large. This is a tradeoff depending on how much memory the users have available. Therefore this optimization should be controlled by some threshold given by the configuration of the implementation, or certain state transitions can be deleted by using a statistical approach at runtime, as described in Section 5.

- **xsd:choice (Opt. 3)**

The compositor xsd:choice defines a group of mutually exclusive particles. Only one can be found in the instance document per occurrence of an xsd:choice compositor. Since we know all of the possible elements that could appear in this schema fragment, it is possible to create the state transitions at startup time. This can be done at initialization time, not startup time.

- **xsd:sequence: Explicitly creating an expanded-type automaton using the maxOccurs and minOccurs attribute (Opt. 4)**

Some data-driven XML documents have repeating elements. For example, an XML document on customer data lists each customer's personal information. If these documents are expressed as one series of state transitions in a linear way, the automaton will become large. We call this type an "expanded-type state transition". In order to deal with these kinds of documents, it is possible to create "loop-type state transitions" in which backward state transitions are created to avoid the expansion of the same type state transitions. Loop-type state transitions have advantage over expanded-type ones in terms of memory consumption since the create automaton is more compressed, but in terms of processing cost, expanded-type state transitions

could become faster since state machines have fewer choices to next states at each states. In order to determine which type of automaton is created, we can use the *maxOccurs* attribute and the *minOccurs* attribute within the xsd:sequence structure. Especially in the case the *maxOccurs* attribute equals to the *minOccurs* attribute, we can explicitly create an expanded-type automaton.

- **xsd:sequence: Specific ordering (Opt. 5)**

The following schema specifies the order of the child elements of the X element, saying that the A element should come first and B comes next. Suppose that an instance document is "<X><A>A</A><B>B</B></X>". Then the initial automaton would consist of 8 state transitions: "<X>", "<A>", "A", "</A>", "<B>", "B", "</B>", and "</X>". By leveraging the schema information, this automaton can be optimized as 5 state transitions. "<X><A>", "A", "</A><B>", "B", "</B>", reducing the number of state transitions. Speaking more technically, if there are N child elements and each child element has a text node (but not an empty element), then the number of state transitions is 3 * N + 2, and after being optimized, this will be 2 * N + 1, reducing the number of states by N + 1.

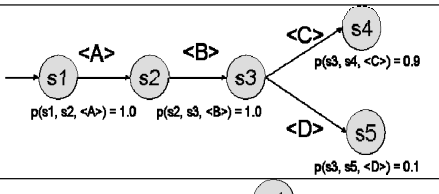
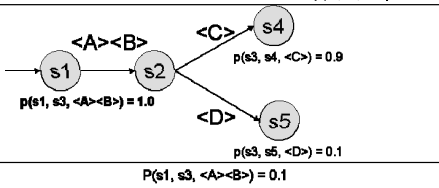
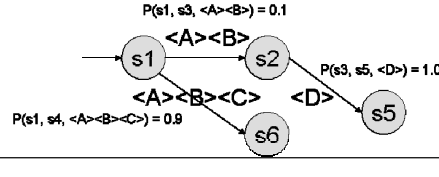
- **xsd:all (Opt. 6)**

The compositor xsd:all is used to describe an unordered group of elements whose number of occurrences may be zero or one. It is possible to create all of the state transitions that cover all of the combinations of the specified elements.

5 Optimization Using Statistics

In this section, we propose an algorithm to optimize the automaton statistically without considering any schema information. To determine which states are to be merged, a transition probability is assigned for the execution of each state transition.

Table 2. Examples of Automaton

Automaton I	
Automaton II	
Automaton III	

Automaton I in Table 2 shows a sample automaton created after some parsing. For each transition, a transition probability *P* is assigned. This is calculated by counting the occurrences of that transition when the parser recognizes an XML element corresponding to the transition label. Automaton II in Table 2 shows that when the automaton recognizes the element <A>, the element <B> always appears next (the transition probability equals 1). The element <C> then appears in 90% of the processed XML documents. In the remaining 10%, the element <D> is next.

An automaton with transition probabilities is defined by {*S, Tr, P*} where S is a set of states, Tr is a set of transitions from one state to another state, and P is a set of transition probabilities. The transition probability p(s_i, s_j, l) is the probability of a transition between states s_i to s_j upon recognizing a level string l.

The automaton shown in Automaton III of Table 2 is specified as follows:

S = (s1, s2, s3, s4, s5)
Tr = (s1, s2, <A>), (s2, s3, <B>), (s3, s4, <C>), (s3, s5, < D>))
P = (p(s1, s2, <A>)=1, p(s2, s3, <B>)=1, p(s3, s4, <C>)=0.9, p(s3, s5, <D>)=0.1

It is natural to merge a sequence of states with the transition probability *p=1* to minimize the number of states. It is plausible to merge a sequence with high transition probability (close to 1), but that may lead to an over-optimization that will cause

Table 3. Algorithm for optimizing the states of automaton

```
Suppose an automaton A = {S, Tr, P}.
For a list of states L = (s1, s2, …, sn),
[Step 1]
// pop a state s from the list L.
s = pop(L);
[Step 2]
if ( si that satisfies p(s, si, l)) = 1 exists) then
  if (p(sj that satisfies p(si, sj, l') = 1 exists) then
    // sj is merged with si and removed from L and N
    delete (sj, L);
    delete (sj, S);
 delete(p(si, sj, l'), P);
 delete((si, sj, l'), Tr);
    // sets a new transition probability to from s to si
    add(p(s, si, append(l,l')), P), p(s, si, ll') = 1;.
    return to [Step 2]
  end if
 else if (si that satisfies p(s, si, l)) > T exists) then
  if ( sj that satisfies P(si, sj, l') > T exists) then
    // create a new state
    add(s', S), add(p(s, s', ll');
    add((s, s', ll), Tr);
    p(s, s', ll') = p(s, si, l)*p(si, sj, l');
    // sj is merged with si and removed from L and N
    delete (sj, L);
    delete (sj, S); delete(p(si, sj, l'), P); delete((si, sj,l'), Tr);
    return to [Step 2]
  end if
 end if
 [Step 3]
 return to [Step 1]
```

unnecessary recreation of merged states[1]. An algorithm which satisfies these conditions is shown in Table 3.

Let's examine the algorithm using the automaton shown in Table 2. First, pop the first state s_1 from the initial list L = {s_1, s_2, s_3, s_4, s_5} and set it as *s* (Step 1). Since the transition probabilities *p*(s_1, s_2, <A>) and *p* (s_2, s_3, <B>) equal 1, these two states are merged (Step 2). The result of the merging process is shown in Automaton II of Table 2. In this figure, the state s_3 is removed from the automaton and the label from s_1 to s_2 is changed to <A> <B>. Next, Step 2 is iterated and s_2 and s_4 are merged, since *p*(s_1, s_2,<A> <B>) is 1 and *p*(s_2, s_4, <C>) is 0.9 (supposing that T = 0.8). In this case, the state s_4 is removed from the automaton and a new state s_6 is created with the transition probability *p*(s_1, s_4, <A> <B> <C>) = 1 * 0.9 = 0.9. The result of this merging process is shown in Automaton III of Table 2. Now the optimized automaton has two paths, <A> <B> and < D>, and <A> <B> <C>. The states s_2 and s_5 are not merged, since the transition probability does not exceed the threshold *T*.

After the merging process, there are no states that satisfy the conditions in Step 2, so the process returns to Step 1 to set a new state for *s*[2].

6 Performance Evaluation

This section describes the effectiveness of each optimization technique proposed in this paper. To simplify the actual XML documents used in our experiment, we use the following type of XML document so that the element *c* is repeated more than once and *C* is a constant value:

<a>X<c>C</c><c>C</c>...<c>C</c>

In reality we used an XML document with the above form that represents a purchase order containing a shipping address, a billing address, and one or more purchase items with some properties such as a product name, quantity, price, and comment[3]. A purchased item corresponds to the element *c* in the above example.

To evaluate the optimization techniques, we performed a series of experiments using the following optimization techniques:

- No Optimization: The approach of the original Deltarser.
- Loop Expansion: Opt. 4 is applied.
- Concatenation: Opt. 5 is applied.
- Variable Fixation: Opt. 2 is applied.
- All Optimizations: Opt. 4, Opt. 5, and Opt. 2 are applied.

Note that in our experiments we did not measure the initial overhead as the basis of the statistical information, but we can understand the performance improvements based on the hypothesis that sufficient statistical information was obtained. Even with the optimizations by using the statistics described in Section 5, the above experiments

---

[1] Note that the parser parses any XML document correctly even in such case s . It simply means there is a cache miss.

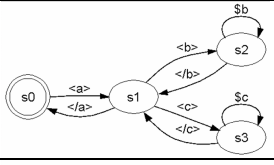
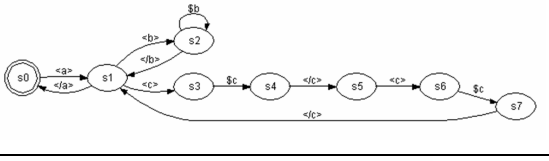
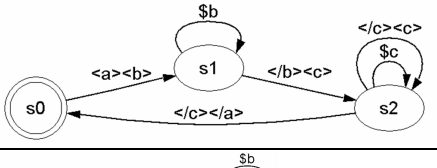
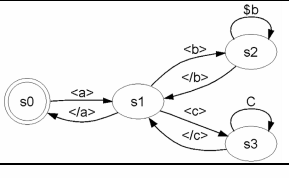
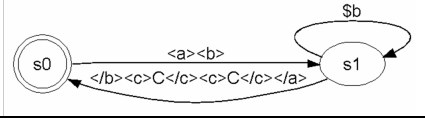
[2] For the sake of simplicity, the case in which a state to be removed has incoming links. In this case, the state should not be removed.

[3] To reviewers: We could show the sample document, but we omitted it to save space.

are meaningful, since the same optimizations are applied. In order to measure the pure cost of byte sequence matching and state transitions, we did not produce any SAX events as in [1], so we can not compare our approach directly with Xerces and Piccolo. However the processing costs to be optimized are the dominant factor described in Section 2, and it is clear that the observed performance improvements are also applicable when measuring the system as a SAX parser and deserialization component in a SOAP engine.

The resulting automatons for each optimization are shown in Table 4. The columns *M*, *N*, and *L* refer to the number of states, the total number of branches to be summed up at each state, and the number of transitions with variables that need partial parsing, respectively.

Table 4. Generated Automaton by Each Optimization Techniques and its properties

Optimization	Formulated automaton	*M*	*N*	*L*
No Optimization		4	20	3
Loop Expansion		8	15	3
Concatenation		3	10	3
Variable Fixation		4	20	1
All Optimizations		2	4	1

For the evaluation environment, we used Windows XP SP2 as the OS, IBM JDK 1.4.2 as the Java Virtual Machine running on an IBM ThinkPad X41 2525-E9J (CPU: 1.6 GHz, RAM: 1536 MB). The comparisons were measured by the average processing times for running 2,000 iterations after 10,000 warm-ups executions (to exclude the JIT compilation time).

The left graph shown in Figure 3 shows a comparison of average processing times for one document. The x-axis is the number of the element *item* that appeared. The element *item* corresponds to the element *c* in the previous example form and appears more than once. The y-axis is the average processing time for one document. The order of each optimization illustrated in the graph is "*No Optimization*" < "*Loop Expansion*" < "*Concatenation*" < "*Variable fixation*" < "*All Optimizations*" (Rightmost is the best). "*Loop Expansion*" obtains a performance improvement of 11.6% over "*No Optimization*" on average. "*Concatenation*" obtains a performance improvement of 31.1% on average. "*Variable Fixation*" obtains an average performance improvement of 26.2%. "*All Optimizations*" obtains has a performance improvement of 73.2%. We confirmed that the proposed optimization techniques are effective through the experiments. The right graph in Figure 3 is the same graph as the left one, but focuses on a small document.

The left graph in Figure 4 shows the memory consumption. Clearly the worst case is "*Loop Expansion*" and as the number of the element *items* grows, the memory consumption gets worse. The right graph in Figure 4 shows the experimental results

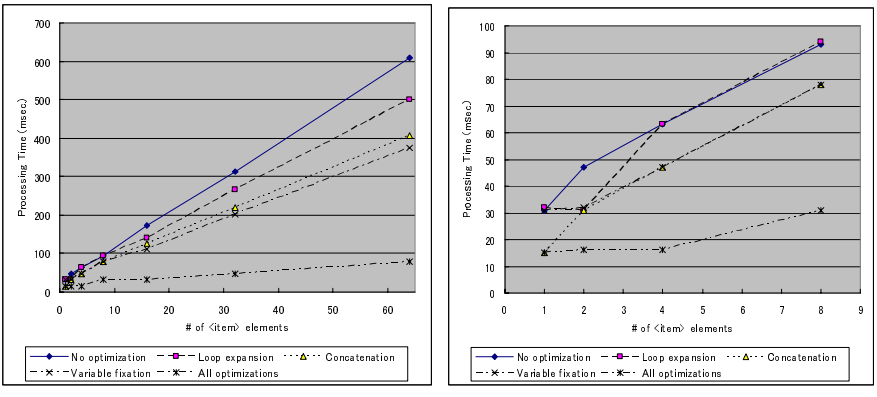


Fig. 3. Comparison of each optimization techniques in average processing time

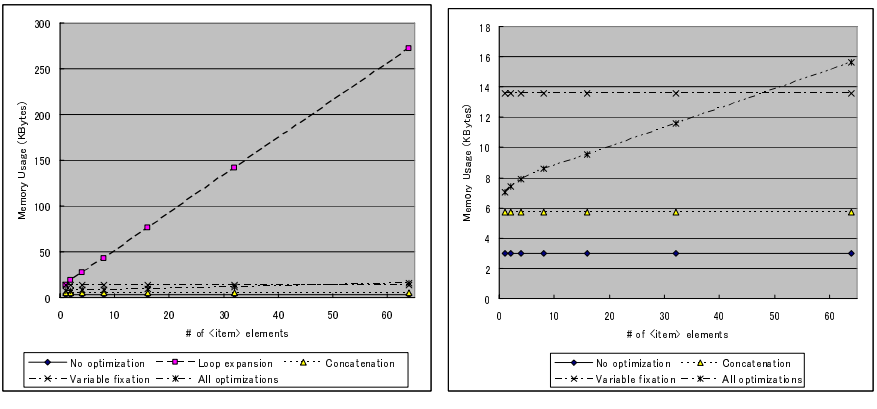


Fig. 4. Comparison of each optimization techniques in memory consumption

except for "*Loop Expansion*". The order of optimizations is "*All Optimizations*" < "*Variable Fixation*" < "*Concatenation*" < "*No Optimization*". "*All Optimizations*" consumes around 3.2 times more memory than "*No Optimization*", but considering the improvement in processing time, this number is acceptable.

Finally, we can conclude that each of these optimization techniques was quite effective, and the combination of multiple optimizations was especially effective in improving processing time with an acceptable level of memory consumption.

7 Related Work

[2][3] focus on the optimization of deserialization using the notion of differential processing. Meanwhile, our proposed XML parser is used not only for Web services but also for general XML-based infrastructures, insofar as it fulfills the condition that all of the processed XML messages are analogous to each other. Our optimization technique using XML schema can be compared with the work in [4], which proposes a parsing approach called schema-specific parsing. The validation of XML instances against a schema is usually performed separately from the parsing of the more basic syntactic aspects of XML. They posit, however, that schema information can be used during parsing to improve performance, using what they call schema-specific parsing. As mentioned earlier, statistical information can be complementary to the information that we can not obtain only from the XML schema, so there would be some situations where our approach has advantages. Currently there is no public implementation available, but it would be worthwhile to compare the performance between our approach and their approach.

8 Concluding Remarks

In this paper, we have presented an optimization approach to enhance XML parsing. Our approach is based on the notion of differential processing under the hypothesis that XML documents are similar to each other, and the proposed approach in this paper enhances our previous work [1] to achieve higher performance by leveraging static information as well as dynamic information. We use XML schema languages for static information that can be used for optimizing the internal state transitions. At the same time statistics for a set of instance documents are used as static information. These two approaches can be used in complementary ways. The experimental results show that all of the proposed optimization techniques are effective, and in particular the combination of multiple optimizations is most effective, yielding a 73.2% performance improvement compared to the original Deltarser. For future work, we will apply the proposed optimization approach to differential deserialization for the SOAP engine proposed in [2].

References

[1] Toshiro Takase, Hisashi Miyashita, Toyotaro Suzumura, and Michiaki Tatsubori. An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization, 14th International World Wide Web Conference (WWW 2005)

[2] Toyotaro Suzumura, Toshiro Takase, and Michiaki Tatsubori. Optimizing Web Services Performance by Differential Deserialization, ICWS 2005 (International Conference on Web Services)
[3] Nayef Abu-Ghazaleh and Michael J. Lewis, Differential Deserialization for Optimized SOAP Performance, SC 2005
[4] Kenneth Chiu and Wei Liu, A Compiler-Based Approach to Schema-Specific XML Parsing, WWW 2004 Workshop
[5] Florian Reuter and Nobert Luttenberger. Cardinality Constraint Automata: A Core Technology for Efficient XML Schema-aware Parsers. www.swarms.de/publications/cca.pdf.
[6] Nayef Abu-Ghazaleh, Michael J. Lewis, Differential Serialization for Optimized SOAP Performance, The 13th IEEE International Symposium on High-Performance Distributed Computing (HPDC 13)
[7] Evaluating SOAP for High-Performance Business Applications: Real Trading System, In Proceedings of the 12th International World Wide Web Conference.
[8] Y Wang, DJ DeWitt, JY Cai, X-Diff: An Effective Change Detection Algorithm for XML Documents, 19th international conference on Data Engineering, 2003.
[9] Markus L. Noga, Steffen Schott, Welf Lowe, Lazy XML Processing, Symposium on Document Engineering, 2002.
[10] J van Lunteren, T Engbersen, XML Accelerator Engine, First International Workshop on High Performance XML
[11] M. Nicola and J.John, "XML parsing: a threat to database performance", 12th International Conference on Information and knowledge management, 2003.
[12] W3C XML Schema, http://www.w3.org/XML/Schema
[13] RELAX NG, http://www.oasis-open.org/committees/relax-ng/
[14] Apache Xerces http://xml.apache.org/