An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization

Toshiro Takase Hisashi MIYASHITA Toyotaro Suzumura Michiaki Tatsubori
IBM Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan
{E30809,himi,toyo,tazbori}@jp.ibm.com

ABSTRACT

XML (Extensible Markup Language) processing can incur significant runtime overhead in XML-based infrastructural middleware such as Web service application servers. This paper proposes a novel mechanism for efficiently processing similar XML documents. Given a new XML document as a byte sequence, the XML parser proposed in this paper normally avoids syntactic analysis but simply matches the document with previously processed ones, reusing those results. Our parser is adaptive since it partially parses and then remembers XML document fragments that it has not met before. Moreover, it processes safely since its partial parsing correctly checks the well-formedness of documents. Our implementation of the proposed parser complies with the JSR 63 standard of the Java API for XML Processing (JAXP) 1.1 specification. We evaluated Deltarser performance with messages using Google Web services. Comparing to Piccolo (and Apache Xerces), it effectively parses 35% (106%) faster in a server-side use-case scenario, and 73% (126%) faster in a client-side use-case scenario.

Categories and Subject Descriptors

I.7.2 [**Computing Methodologies**]: Document and Text Processing—*markup languages, standards*; D.3.4 [**Software**]: Processors—*run-time environments*

General Terms

Performance, Design, Experimentation

Keywords

XML parsers, SAX, automata

1. INTRODUCTION

XML (Extensible Markup Language) [5] processing can be a significant runtime overhead in XML-based infrastructural middleware such as Web Services application servers. [28, 11, 15, 25, 8, 18] The good and bad of XML both lie in its verboseness. For example, well-formed pairs of named marking-up tags for XML elements contribute to its human-friendliness and vender-neutrality, but require extra computation in processing. The computation specific to XML includes variable representations for the same tag, the handling of namespaces, tolerance for multi-character encodings, etc.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

This paper proposes a novel mechanism for efficiently processing XML documents in most cases. Its notable feature is to remember the documents processed previously and to use those processed results to deal with a new XML document. Incoming XML documents that share very similar parts with previously-processed documents can be processed very quickly with this feature. Even though this feature adds some overhead in processing strange XML documents (those which are not similar to the documents processed before) compared to normal XML processing, the gain from the speed-up for similar documents can easily compensate for the overhead where there is repetitive processing as in situations like Web Services middleware.

Given a new XML document as a byte sequence, the XML parser proposed in this paper, in most cases, does not analyze most of the XML syntax in the document but just compares the byte sequence with the ones that were previously processed. The parser then reuses the resultant parse events stored during the previous processing. Only the parts differing from the previously-processed documents are processed in the normal way for XML parsing. It remembers the byte sequences in a *DFA* (Deterministic Finite Automaton), where each state transition has a byte sequence and its resultant *parse event*. In addition, the parser remembers processing contexts in DFA states so that it can partially parse unmatched byte sequences. The parsing process normally follows state transitions in the DFA by matching byte sequences. If no states to transit to are found, it partially parses the unmatched byte sequence until it finds a resultant state from which it can transit to existing states. Then it continues to make transitions in the DFA.

The notable feature of our DFA structure is its "safety" and "efficiency" in reusing parse events from previously-processed XML documents. The safe reuse means that the resultant parse events conform to the XML specification [5, 4]. Note that it must accept well-formed XML documents and reject ill-formed ones. Efficient reuse means that it should reuse as many parse events as possible from previous parsing while minimizing computation costs for the reuse. Less reuse can cause extra partial parsing, which is costly, and higher reuse computation cost could cancel the benefits of reuse. We need to find good trade-offs for efficient reuse, since aggressive reuse sometimes increases the computation cost for reuse, and vice versa. We carefully designed the strategy for constructing DFAs so that they can find a good trade-off point for efficiency.

Several improvements have been created for efficiently processing XML documents using application-specific infor-

mation available from their schemas, such as DTDs. A few investigated optimization of lexical analysis [8, 9]. However, schema-based techniques cannot capture the optimizations from byte-level resemblances among documents, especially in prefix-namespace binding and in the representations of tags, since they are not specified in schemas. Most of the schema-based optimization efforts are focused on higher-level XML processing rather than on lexical analysis of XML documents serialized as byte sequences. For an example of XML document validation, XSM [20] uses cardinality constraint automata specialized by accepting schemas, but it assumes SAX events are the input of the automata, not low-level byte sequences.

We implemented in Java an XML parser named *Deltarser* based on the described mechanism. Our SAX implementation complies with the JSR 63 [24] (Java API for XML Processing 1.1) standard specification. We conducted some experiments and observed its advantages over ordinary XML parsers in certain representative scenarios which we believe cover the majority of practical Web Service usages. Though this paper focuses on the SAX implementation, the same technology is easily applied to other streaming parsers. In fact, our pull-parser implementation complies with the JSR 173 [13] (Streaming API for XML) standard specification and we observed efficiency similar to the SAX implementation.

The rest of the paper is organized as follows: In Section 2, we specify the overheads in XML parsing and explain why there should be opportunities for reducing these overheads in many practical cases. We describe the design and implementation of our XML parser, Deltarser, in Section 3 and show its effectiveness through the experimental results in Section 4. Finally, we conclude this paper with Section 5.

2. XML PROCESSING OVERHEAD

XML (Extensible Markup Language) [5] is heavily used in XML-based infrastructural middleware such as Web Services [22, 10] application servers, and its processing can be a significant runtime overhead. The reason why the processing overhead of XML is high lies in its verboseness.

2.1 XML Processing in Web Services Middleware

The cost of XML parsing has a large impact on the execution performance of recent XML-based middleware [28, 11, 15, 25, 8, 18]. XML-based middleware analyzes XML documents to construct data objects for its application programs or for its own use. For example, a Web Service application server for service applications in Java receives SOAP messages encoded in XML and deserializes them into Java objects while it reads its settings from a file, whose format is also defined in XML by JSRs [16, 17].

Web Service middleware uses XML heavily since XML is a key construct of Web Services. Web Services are enabling technologies for vender-independent distributed environments where loosely coupled servers can interconnect to each other. In such environments, standard specifications define most of things to be written in the XML-based languages recommended in the specifications. For example, messages passed in Web Services are enveloped according to SOAP [22], which is a protocol using XML for messaging.

In addition to messaging, there are a number of other places where XML is used in Web Services middleware. For publishing available Web Services, the services are described in WSDL [10] (Web Services Description Language), which specifies the format for service descriptions. Configuration files like `webservice.xml` for application servers are often described in XML, as defined in JSR 109 [16] and JSR 921 [17].

2.2 Overhead Constructs in SAX Processing

The computations specifically required to process XML documents include tolerance for various character encodings, variable length data, ignorable white spaces, line break normalization, the handling of namespaces, and the creation of parsed result objects. Figure 1 shows the normal processing model in conventional XML parsers for SAX [24].

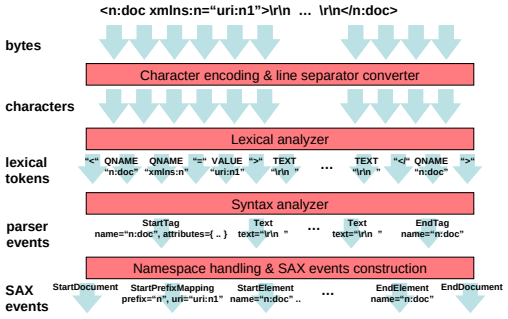

Figure 1: Processing stages in normal XML parsers for SAX.

First, an XML parser needs to convert the character encodings, since the external encoding, the encoding format in which an XML document is encoded, may be different from the internal encoding, the encoding format used in a program. For example, an XML document is often encoded in UTF-8, while a Java program handles characters in UTF-16. An XML parser must convert the original UTF-8 encoded characters into UTF-16 encoded characters in order to use the characters in the middleware or to pass them to application programs.

Processing XML documents requires the recognition of token delimiters like "<" and ">" [5] since XML data objects normally have varying lengths. Like most programming languages, an XML document must be "parsed" to recognize lexical tokens, even though the syntax is relatively simple and easy to analyze. This is extra runtime overhead compared to efficient wire formats with fixed-length structures [6].

XML allows ignorable spaces in its tags. For example, a start tag:

```
<doc language="Japanese">
```

can also be written as:

```
< doc  language = 'Japanese' >
```

White spaces, tabs, new lines, and carriage returns are allowed in many places. It means that there are a variety of ways to express a tag with the same meaning. An XML parser must skip such ignorable spaces. This adds an overhead cost since it requires testing whether or not each character data is one of these ignorable characters.

The normalization of line separators is also a task handled in an XML parser. The XML specification allows lines to

be separated by single LF characters, by CR LF character sequences, or by single CR characters. However, internally these separators are always converted to single LF characters. An XML parser must detect LFs, CR LF pairs, and CRs to convert all of them to LFs.

An XML document may use namespaces [4] to avoid collisions among tag names or attribute names, so an XML parser must handle namespaces for documents using them. Since XML namespaces are scoped within elements, namespace handling involves stacked map management for matching each namespace prefix to its namespace URI. For example, the nested elements:

```
<n:doc xmlns:n="uri:n1">
  <n:title xmlns:n="uri:n2"/>
  <n:item/>
</n:doc>
```

have the same namespace prefix "`n`" but they must be resolved to different namespace URIs, `uri:n1` for `doc` and `item`, but to "uri:n2" for `title`.

Finally, an XML parser needs to construct parsed result objects so that it can pass them to its users (middleware or application programs) through an API like SAX. For example, with SAX 2.0 [24], an application subclass overrides a method `startElement()` in the class `ContentHandler`. A SAX parser is responsible for creating `String` objects for its namespace, local name and qualified name parameters, and `Attributes` objects for its attributes parameter to pass them through this interface.

2.3 Opportunities and Challenges in XML Processing

On balance, we believe there should be a lot of opportunities to optimize the processing of XML documents. Even though optimizing for general cases is hard to do, optimizing for special cases is possible. For example, if we knew all the messages were formed in a certain way, we could optimize the parsing process for such XML documents.

In an environment of Web Service application servers, we can expect that most of the messages will be generated by machines. In particular, RPC-style request-response messages are often generated by middleware with XML serializers. When accessing Web services in client code, proxy classes and frameworks provided by middleware handle all of the infrastructure coding. For example, a C# client uses the .NET framework to access Web Services, and a JAX-RPC client may use the `javax.xml.rpc.Call` instances implemented by some particular vendor.

Though formatting styles are different for various programming languages, implementation vendors, or versions, the same XML serializer implementation generates the same kind of service requests and responses with different parameters and return-values in very similar byte sequences. This is because such XML serialization is performed by a certain runtime library or by proxy code generated by a certain tool provided by middleware or a development environment. As a result, recurring SOAP messages often form very similar byte sequences except for the contents of elements representing different parameter values, as long as they are sent by application programs with the same infrastructural middleware.

A key challenge in optimizing XML processing is to preserve the interoperability of the XML documents. We can do anything in closed environments using XML as the wire format, but it is expected that XML is mainly used in open environments, where there may be many participants conforming to the XML specifications [5]. An optimized XML parser must not reject or be broken by XML messages in forms unexpected by the optimization.

In addition to the compatibility, another key challenge is to make optimizations tolerant of implementation changes in infrastructural middleware. An optimization for a limited set of XML serializer implementations can easily be obsolete and ineffective, since they are easily changed by new implementation versions or by new implementation vendors.

3. DELTARSER

We designed an XML parser named *Deltarser*, which can efficiently and safely process XML documents similar to previously processed documents. The efficiency for similar documents applies in situations like Web Service application servers, where middleware needs to process a lot of 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 [27] implementation. In fact, the SAX implementation of Deltarser complies with JSR 63 [24], which is a standard API for XML processing in Java. Applications or middleware using XML parsers through the standard API can easily be changed to use Deltarser without modifying the code.

In the best cases, given a new XML document as a byte sequence, Deltarser does not analyze most of XML syntax in the document, but just compares the byte sequence with those that were already processed. The parser reuses the processed results stored in memory for the matching parts. Figure 2 depicts the abstract view of SAX processing with Deltarser.

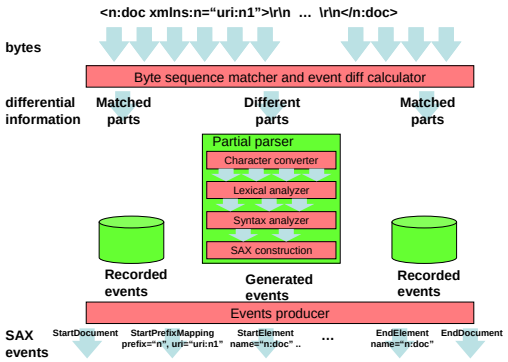

Figure 2: Processing stages in Deltarser for SAX.

The key ideas and technologies we developed for Deltarser are as follows:

Byte-level matching for XML processing It makes use of byte-level matching for the most parts of the document to be processed. Since we take the context of the XML document into consideration, we can reliably compare XML documents by only doing byte-level matching.

Remembering processed documents in a DFA For efficiently remembering and comparing previously processed documents, it remembers the byte sequences 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.

Partial parsing 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.

Incremental well-formedness checking 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.

In the rest of this section, first, we introduce the fundamental framework of the automaton used to extract differences from incoming documents and how to incrementally construct the automaton. We go on to explain how to process the partial results. Finally, we describe how to efficiently notify the applications of the SAX events.

3.1 Byte-level Matching for XML Parsing

The main action of Deltarser is byte-level comparison, which is much faster than actual parsing. When feeding the actual XML document to this state machine, we have only to compare the byte sequence of each state and the incoming document.

Applying naive byte-level comparison to XML documents gives rise to a serious problem. We might misinterpret the document because the byte-level representation of some XML fragment is not uniquely interpreted based on the information currently in view of the XML infoset, i.e., the same byte sequence can be interpreted with different semantics, which depend on the current context. Let us look at the following example.

```
<x:a xmlns:x="ns1"> </x:a>
<x:a xmlns:x="ns2"> </x:a>
```

Let us focus on both of the end tags. Although they match exactly, they have different semantics in the XML view. This is because the namespace declarations are different.

In order to address this problem, we must take *context* into consideration when matching XML documents at the byte level. We rely on Proposition 1 below to process XML documents.

In preparation for presenting the proposition, we define a context, $C = (E, N, D, l)$, which consists of E, a sequence of elements to which it currently belongs; N, all of the namespace definitions that are currently visible; D, all declared entities of the document; and $l \in \{0, 1, 2, 3, 4, 5\}$, the location in the document, where $l = 0$ means the beginning of the document, $l = 1$ means either a document declaration or a document element has not occurred, $l = 2$ means a document declaration has occurred but the document element has not, $l = 3$ means it is now within the document element, $l = 4$ means it is now after the document element, and $l = 5$ means the end of the document. As a special treatment for the uniqueness of initial and final states, when $l = 0$ or $l = 5$, we define all the other components in the context as empty. This treatment does not cause any problem because at the beginning and the end of the document we do not need any other information than l for that context.

DEFINITION 1. *Let d be an XML document, and $p \in$ Integer be the offset in bytes that points to d. By definition, $C(p)$ is the context of the point p.*

Let us consider the following XML document.

```
<a xmlns:p="xxx"> <b> A </b> </a>
```

In this example, the context at the point of `A` is

$$E = \{a, b\}, N = \{(\text{prefix} =' \text{p}', \text{uri} =' \text{xxx}')\}, D = \{\}, l = 3.$$

PROPOSITION 1. *Let ev_1 be an event and C_1 be a context that ev_1 belongs to. Provided that the document from a certain point, p_0, matches with the byte sequence of ev_1 for its length and $C(p_0)$ is equal to C_1, then the parsed event from p_0 must be equal to ev_1.*

Proposition 1 is naturally inferred from the specification of XML [5, 4].

Strictly speaking, an external entity reference is resolved every time it is about to be parsed, which allows for the possibility that the referred entity might have been changed. In this paper, however, we do not take this into consideration since it is a very unusual case. In addition, SOAP does not allow document type declaration.

In order to simplify the character-level interpretation, we assume the character encoding of the document is "stateless". Both UTF-8 (1 byte is 1 octet) and UTF-16 (1 byte is 2 octets according to ISO/IEC 10646), which are the only encodings XML parsers must support, satisfy this requirement. Stateless here means that 1) one character directly corresponds to the certain number of bytes; and 2) an XML special character forms a distinct one byte in the encoding. For other encodings such as ISO/IEC 2022 variants, we can convert them before processing a document (as the normal parsers do). In addition, such complex encodings are rarely used in XML, and especially for SOAP messages, because such encodings cause many problems for interoperability[3]. Thus, this approach is not a drawback compared to other XML parser implementations.

3.2 Representing Parsed Documents as an Automaton

In Deltarser, each parse event and its corresponding byte-sequence are stored in one edge of an automaton. This automaton has two major characteristics: 1) it can directly process byte-level events; and 2) it accepts only well-formed XML documents. Let us look into how this automaton is constructed.

First, we represent a parsed document as a sequence of events. Figure 3 shows the static structure of the event classes.

Each event has a one-to-one correspondence to a fragment of the XML document so that it has a byte representation in the actual document. Note that a context is updated after each event is processed, which consists of a sequence of *StartTagEvent* and all of the declared entities. From the context, we can identify 1) what namespace declarations have been

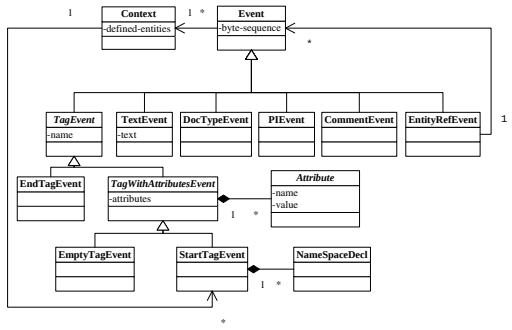

Figure 3: Static structure of an event in a UML diagram.

declared so far; 2) the hierarchy of elements where the event is located; 3) how to resolve entity references; and 4) which part of the document is currently processed. This information is an integral part of how to construct an automata.

Let us look at the following sample document.

```
<p:e xmlns:p="urn1">text<x a="ccc" p:b="ddd"/></p:e>
```

This is converted to a sequence of events as follows.

```
[StartTag: name="e" uri="urn1"
  {Attributes: }
  {NSDecls: (prefix="p", uri="urn1")}]
[Text: value="text"]
[EmptyElementTag: name="x" uri=""
  {Attributes: (name="a", uri="", value="ccc")
               (name="b", uri="urn1", value="ddd"}]
[EndTag: name="e" uri="urn1"]
```

After converting the document into events, we can construct an automaton by regarding the context itself as its state. In this step, the same contexts are unified into only one state. The equivalence of contexts is naturally defined as being all of the items in these context are equal, i.e., all the element in E, all of the declared namespaces in N, all of the entity definitions in D, and the integer value of l are equal. We treat the initial and final states as special. The initial state is regarded as the start of the document, i.e., l in its context is must be 0; and the final state is regarded as the end of the document, i.e., l in that context must be 5.

Let us look at the process step by step. First, we define all the contexts in the document as states in the automaton. Next, by tracking the sequence of events that come from the actual document, we can connect these states with the event as edges of the automaton. In this phase, events that have the same byte representations are regarded as equal, and we do not add duplicated connections with states, because if byte representations and contexts are equal, then the interpreted events are also equal. Finally, we mark the states where l in the state's context is 0 and 5 as initial and final states, respectively. The above example can be represented as shown in Figure 4.

3.3 Matching a Document with an Automaton

Although Deltarser compares an incoming document at byte level, it identifies the differences at the event-level, i.e., whenever it finds any discrepancies at the byte level, it interprets them as event-level differences.

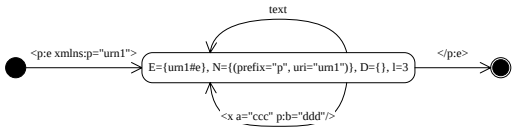

Figure 4: Example directed graph that corresponds to a parsed document.

```
<p:e xmlns:p="urn1">text<y/></p:e>
```

Figure 5: A sample XML document.

Now let us see how a similar document is actually processed. We suppose that we have the state machine shown in Figure 4 beforehand, and process the sample document shown in Figure 5.

First, we set the current state to the initial state and the current position to the head of the incoming document. Since the only possible next state is `<p:e xmlns:p="urn1">`, we compare its byte sequence with the sample document from the current position. These are exactly the same until the end of the state's byte sequence, and therefore, we move the current state to the next state,
$E = \{\text{urn1\#e}\}, N = \{(\text{prefix} ='' \text{ p}'', \text{uri} ='' \text{ urn1}'')\}$,
$D = \{\}, l = 3$,
and the current position to the corresponding position. At this point, the possible next events are `text`, `<x a="ccc" p:b="ddd"/>`, or `</p:e>`. In order to efficiently compare these with the incoming document, we use a binary search technique. Provided that the byte sequence of these events was sorted beforehand, we can quickly find the appropriate state. In this case, we select `text` as the accepted event, and stay in the same state.

3.4 Updating an Automaton

Our parser incrementally modifies the original DFA by adding new events made from the byte sequence of a new document when it fails to match the incoming byte sequence with the DFA. Let us consider this situation by continuing with the example.

On continuing, the possible next events are still the same, `text`, `<x a="ccc" p:b="ddd"/>`, or `</p:e>`, neither of which matches with the incoming documents byte-sequence, `<y/>`. Therefore we cannot move to any next state in the state machine. At this point, we must partially parse the document from the last matched position. In this example, we parse from the position immediately after `text`. Hence, we parse the document from `<y/></p:e>` with only one event by using the information about the last matched state. Notice that we also have to check that the parsed result is well-formed. In this case, we obtain a new event:

```
[EmptyElementTag: name="y" uri="" {Attributes: }]
```

After parsing it, we are able to incrementally add a new edge. In this example, the context is still the same even after the `EmptyElementTag` event. Thus, we only have to add a connection to the same state, as shown in Figure 6.

This process eventually causes a new bifurcation in the state machine, but it should be stressed that this bifurcation leaves the automaton deterministic, because if these byte sequences exactly match with each other, these events must

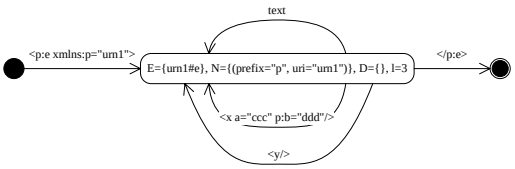

Figure 6: Adding a new edge as a parse event.

be exactly matching under the same context so that we do not have to add a new edge.

In practice, since adding edges inevitably consumes some memory, we should limit the number of edges for one state (i.e. a context). In particular, text events, processor instruction events, or comment events all lack "patterns," so that many of those are not worth remembering. Our Deltarser can limit the number of text events for each context to a certain fixed number, denoted by M_{text}.

Next, we can continue to process the rest of the document by byte matching. The next byte-sequence is `</p:e>` and it exactly matches an event in the state machine. Thus, we can move to the final state. Since the document ends here, we have finished the processing.

It is clear that we can efficiently process the document with byte matching and incrementally process the document by following the above procedure. However, is this procedure safe? In other word, does it really ensure accepting only the well-formed documents? We discuss this question in the next subsection.

3.5 Well-formed Automata

Let us first state the conclusion. The automata constructed by the above procedure accepts a document only if it is well-formed. We call such automata *well-formed automata*. This characteristic is essential for the safe processing. In addition, suppose that the partial parser accepts all well-formed documents, and then the automata is extended to accept these documents.

A formal definition of a well-formed automaton is that all of the paths in the automaton from the initial state to the final state must correspond to a well-formed document. Whenever the automaton remains well-formed even after it is extended due to incremental processing, then the processed document is certain to be well-formed by definition, because it corresponds to a valid path from the resulting well-formed automaton.

We do not give rigorous proof here, but sketch why the above procedure assures the automaton remains well-formed. Context is a key concept in the proof. Let us suppose we have a well-formed automaton beforehand, and the partial parser adds some events as edges in the automaton, and then we obtain a new distinct path, which consists of the sequence of contexts, $C_1, ..., C_n$, and C_1 and C_n must exist in the previous automaton. This condition is guaranteed by the fact that there is only one initial and one final state in a well-formed automaton, so we will share at least the initial and the final state for every well-formed document.

By Proposition 1, the interpretation of the incoming byte sequence as XML at C_1 is always the same. Therefore the partial parser can check whether or not the incoming byte sequence at the point of C_1 is well-formed, because all of the required information to check it (other than the incoming byte sequence itself) is completely defined within the context. Strictly speaking, to prove this statement, we have to check all of the well-formedness rules in the specification of XML [5, 4]. However, since we do not need any information from farther ahead in the document during parsing to check the well-formedness (according to the design of XML), we can give the well-defined context anyway. Since, according to our definition, the context has all of the information that affects the interpretations of incoming byte-sequences, it can always be mapped to (some other) well-defined context. Therefore, we can also regard our definition as a well-defined one.

Suppose the automaton is still well-formed after adding C_1, then we create a new context, C_2, and proceed to parse the next incoming byte sequence, and then check if it is well-formed in C_2. Inductively, we can reach C_n that comes from the previous well-formed automaton, and after C_n, since the interpretation of the byte sequence is exactly the same, all of the paths after it must correspond to a part of a well-formed automaton. Therefore all of the new paths in the extended automaton correspond to well-formed documents.

As for the first well-formed automaton, we can construct it by simply connecting from the initial state to the final state with the empty element tag event of the document element. Therefore, the above procedure insures the automaton is well-formed.

3.6 Partial Parsing

When the incoming byte-sequence does not match with any events in the state, we have to partially parse the document. We require a context in order to parse the document from an intermediate point, which is available in each state of the state machine. The partial parser parses the byte-sequence for one event and updates the context if required.

The partial parser has few disadvantages compared to the typical non-partial parsers. We can instantly continue to parse a document since the context has all of the essential information.

Note that we have to provide a special treatment for general entity references. According to the XML specification Section 4.3.2 [5], if all of the entities are well-formed, each logical and physical structure must be properly nested. Therefore we can regard the replacement text of such an entity reference as one event. That is, it does not alter the context after the entity reference if it is well-formed. Eventually, the partial parser only has to treat an entity reference as a distinct event, and check that its replacement text follows the constraints of well-formedness.

3.7 Notifying SAX Events

While processing an XML document, Deltarser sends SAX events to an application as its output. To minimize the computation cost for reuse by avoiding object construction as much as possible, it prepares SAX event objects in their final form when remembering the processing results to avoid object construction.

For example, a `TagWithAttributesEvent` instance has an object that implements the `Attribute` interface required for SAX events. Since this object is immutable, we can safely reuse it without duplicating it. `StartTagEvent` and `EndTagEvent` have arrays for a `startPrefixMapping` SAX event and a `endPrefixMapping` SAX event, respectively, both of which are reusable. In contrast, we cannot reuse character

arrays in `TextEvent` for `characters` SAX events because applications may alter them. Thus, Deltarser duplicates them to send to the applications.

4. PERFORMANCE EVALUATION

This section describes a performance evaluation of our prototype implementation. First, we conducted some fundamental performance analysis experiments. Then we applied it to real-world application scenarios of XML parsing for Web services middleware.

We did performance comparisons between our parser and two other XML parsers: Apache Xerces [27] and Piccolo [30]. Xerces is one of the most popular parsers and Piccolo is regarded as one of the fastest SAX parsers. The experimental environment was using Sun Hotspot Server VM 1.4.2 on Linux Kernel 2.4.18-14, running on an IBM IntelliStation M Pro 6850-60J (Intel Xeon 2.4GHz CPU with 512KB cache) with 2GB of memory.

In this experiment, we set the number of remembered text events in each context (defined as M_{text} in Section 3.4) to 1. This value is determined by our observation that most texts fall into two classes, those that never change (so one string is enough) and those that change almost every time they appear (so there is no benefit from saving them).

4.1 Fundamental Performance Analysis

First, a fundamental performance analysis was conducted by comparing the total elapsed time with the other XML parsers in two cases. In the first case, the incoming document is highly similar to the previously parsed document, so the parser can skip the parsing and generation of SAX events for a large portion of the XML message. In contrast, the second case is when the incoming document is completely different from the previously parsed documents so that the performance of our parser includes the total of the time for parsing, for creating new automata states, and for generating the SAX events.

We excluded the compilation time of the JIT compiler from the measurements by running the benchmarking program for 10,000 iterations before the measurements. In Web Service middleware, it is expected that the stable running state continues unchanged for a long time. Therefore, it is more appropriate to measure the execution time after the JIT compiler has completely compiled the code. Then we measure the total time to perform 10,000 iterations of parsing XML documents to estimate the average time for parsing one document.

The Cost of Matching

In this way we can measure the performance when a large number of XML documents with similar structures are arriving. In other words, all of the start tags and end tags are matched, but the text content may vary. We performed the experiment in two cases: the case where all of the text content is the same, and the case where the text content is always different.

For Web Services, the structure of the request XML documents is defined in WSDL. Therefore, request documents for the same operation are very similar if the documents are generated by the same implementation. For example, requests for a certain search operation are likely to differ only in the text content. Hence, in this experiment, we changed just the text content for the incoming documents.

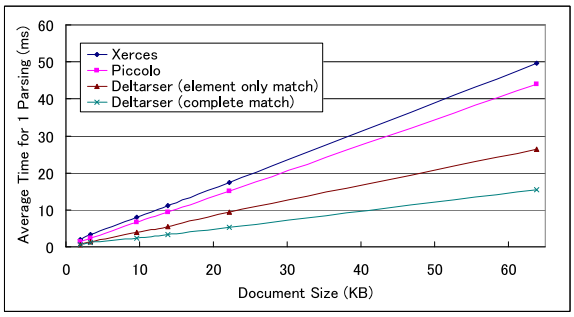

Figure 7: Average elapsed times for parsing an XML document.

Figure 7 shows the results of our experiment. In this experiment, some XML documents which include Google search responses are used. We generated some XML documents of various sizes, from 1 KB to 64 KB, by changing the number of search results. Then we measured the elapsed time for 10,000 iterations, and the average time for one parsing is shown in Figure 7.

"Deltarser (complete match)" and "Deltarser (element only match)" show the complete match and the case when all of text content has changed, respectively. When our parser receives a similar document, our parser is approximately 90% faster than Xerces and 70% faster than Piccolo.

The Cost of Creating New Automaton Paths

Comparing to partially parsing text content, parsing elements is costly in Deltarser, since this creates new states in the DFA. These costs are in proportion to the number of elements that must be partially parsed.

In order to estimate the cost of partial parsing involving state creation in a DFA, we measured the elapsed time for applying partial parsing to a full XML document. We used 1 KB Google search response messages, which is the same message used in the previous experiment.

We also wanted to estimate in how many times Deltarser can compensate for the penalty of partial parsing by processing similar documents. Therefore we parsed many structurally similar documents with different text contents. This parsing process is identical to "Deltarser (element only match)" in the previous experiments.

Figure 8 shows the cumulative parsing times for a group of similar documents. The x-axis is the number of similar XML documents that were parsed.

For partially parsing and creating full states for an XML document, it takes 2.32 ms, whereas Xerces parsing takes 0.14 ms and Piccolo parsing requires 0.08 ms. We can see that Deltarser overtakes Piccolo with regard to the total parsing time before 50 similar documents are processed.

Memory Usage

We also evaluated memory usage for the number of memorized XML documents. For this experiment, we temporarily modified our parser to force the automaton to store even the same XML document as a new path. Then we measured the memory usage when it stored many XML documents. Figure 9 shows the memory usage when many 1 KB and 5 KB XML documents are stored. The result suggests that the memory consumption is acceptable in such an environment.

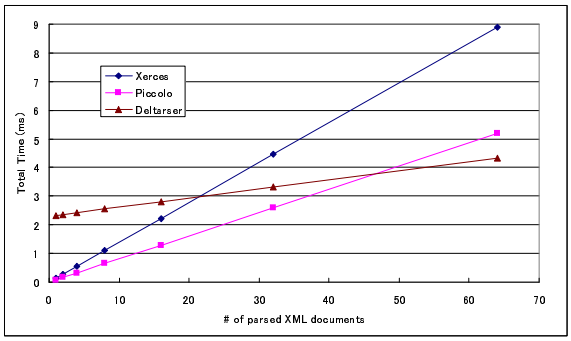

Figure 8: Cumulative parsing costs for a group of similar documents.

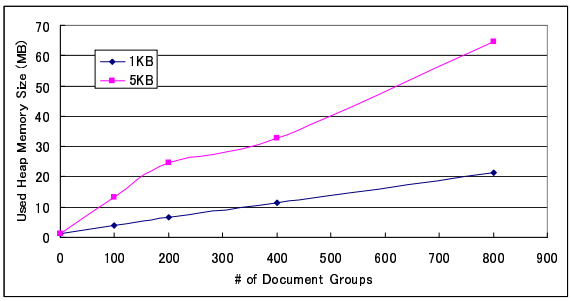

Figure 9: Memory consumption for storing documents.

A key assumption is that our parser is intended for server-side machines with relatively large memories. We believe that the result in Figure 9 shows that the memory consumption is acceptable in such an environment. Even though 64 MB of heap memory are used for 800 documents, it should be rare that one parser needs to handle hundreds of different kinds of XML documents without any similarities. However it would be good to have a function for discarding relatively ineffective automaton paths. We could employ some well-known cache replacement algorithm such as the LRU (Least Recently Used) algorithm to do so.

4.2 An Application Benchmark

We evaluated Deltarser through scenarios based on a real-world application, the Google Web APIs [14]. This is a Web service that allows software developers to access a set of services provided by Google via the SOAP and WSDL standards. We provided scenarios for server-side and client-side services.

The Google Web Services

Google Web Services has three services: searching, returning a cached page, and spelling suggestions. The corresponding request operations received by the servers are `doGoogleSearch`, `doGetCachedPage`, and `doSpellingSuggestion`, and the corresponding response operations received by the clients are `doGoogleSearchResponse`, `doGetCachedPageResponse`, and `doSpellingSuggestionResponse`, respectively.

Note that the properties of the incoming SOAP messages are different between the server side and the client side as regards Deltarser performance. In this scenario, a server tends to receive different formats of incoming SOAP messages from various clients that serialize the messages with a variety of SOAP implementations such as .NET, Axis, and so forth. Meanwhile, the client tends to send its request to only one SOAP container and then receives XML messages in a specific format serialized by a single serializer implementation.

For testing the server-side scenario, mixed messages of the three request operations were used. We assumed that the searching service is used more frequently (60%) than the other services (20% for each). For the client-side scenario, our experiment used only response messages for the searching service. This client scenario is reasonable since a client does not need to use all of the services.

Message Serializers

We used two serializer implementations to generate SOAP messages: Apache Axis and Microsoft .NET Framework. These two serializers generate SOAP messages with different namespace prefixes and different white spaces for identical operations.

We provided two sets of messages to parse. The first set includes only the Axis version of the request messages. It results in three "different" message groups, which would make Deltarser to generate three distinct paths of state transitions in its DFA. The second set includes both Axis and .Net versions, resulting in six different message groups.

We experimented with two sets of messages on the server-side scenario to see the impact of the numbers of serializer variations. For the client-side scenario, we only experimented with the first, Axis-only set of messages.

Parameter Values

The parameter values of the three operations in the request and response messages may vary for each message. For example, the query keyword parameter value varies frequently while the encoding parameter value remains in UTF-8. We assumed that 4/10 of the parameter values in the searching operation request vary frequently and that all of the parameter values in the other two operations requests vary frequently. (A total of 57% of the parameter values vary in the server-side scenario.) Similarly, we assumed that 11/18 of the parameter values in the search responses varied frequently. (A total of 61% of the parameter values varied in the client-side scenario.)

Experimental Results

Figure 10 shows the throughputs of the parsers in the steady state for each scenario. For reference and to see the best and worst cases of parameter value variability, we measured unrealistic message scenarios for "Deltarser (element only match)" and "Deltarser (complete match)", using the same idea as in Section 4.1. Also for reference, we measured two unrealistic server-side scenarios of "Server-side (.Net)" and "Server-side (Axis)". Each of these scenarios uses only single-vender messages.

In the server-side scenarios of "Server-side (.NET and Axis)", Deltarser parses 106% faster than Xerces and 35% faster than Piccolo. In the client-side scenario, Deltarser parses 126% faster than Xerces and 73% faster than Piccolo. In addition, we do not see any significant negative effects from the increased number of serializer implementation variations.

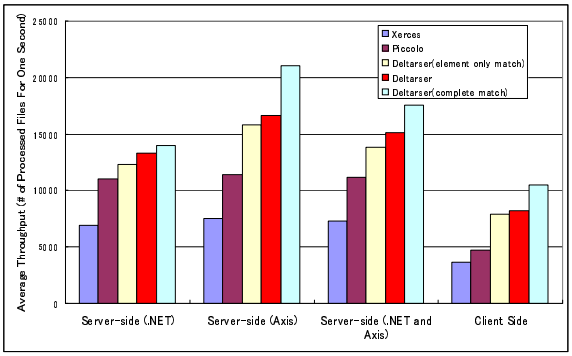

Figure 10: Average throughputs for 100,000 documents in the server/client-side scenarios.

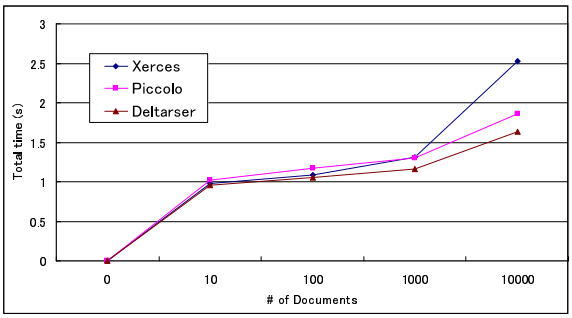

Figure 11: Elapsed times for parsing XML messages in the server-side scenario.

Figure 11 shows the elapsed times for parsing the XML messages before the parsers get to their steady state in the server-side scenarios. Since the client-side scenario also shows a similar pattern, we did not present those results.

Figure 12 shows the consumed heap memory size in the server-side scenarios. We observed an approximately 50 KB increase for additional serializer implementation variations by comparing "Deltarser (.NET and Axis)" to "Deltarser (.NET)" or "Deltarser (Axis)".

In our experiments, the creation time for new automaton paths is included in the runtime execution time. However the cost of creation is relatively large. It is possible to exclude this overhead from the runtime execution time. For example, we can update the automaton in a separate thread process, or in a batched process at a time of low system load.

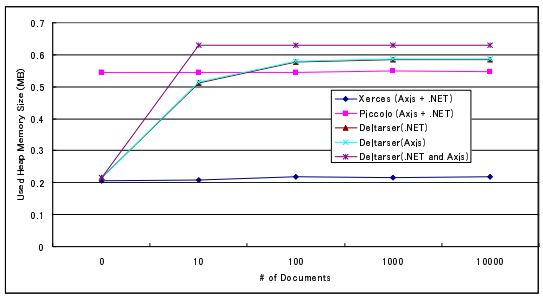

Figure 12: Memory usage in the server-side scenario.

5. RELATED WORK

5.1 Difference Extraction

The problem of computing similarities between two files has been studied extensively. Efficient difference extraction implementation is available in many programs like UNIX's diff command. However, it is not straight-forward to apply it for computing the similarity of a message against multiple messages. Moreover, we need to find similar parts extracted from multiple messages, rather than finding a single, most similar file.

Manber studied how to efficiently find similar files in a large file system [21] and presented a tool called *sif*. While our parser directly remembers the previously-processed documents in a DFA, sif partitions files into small parts and then remembers their hash values computed based on their textual representation. These hash values are used to look up small parts of files in the file system which is identical to a small part of a newly-given file. The purpose of these hash values is to estimate the overall similarity of each file in the file system to the new file.

Deltarser, however, must take account of processing context for the equality of two document fragments in addition to textual representation. Though a similar hashing technique is used for looking up identical contexts in Deltarser, it just compares two byte sequences for looking up identical document fragments.

5.2 Protocol Filters

Efficient network packet filters/classifiers [29, 2] often use pattern matching in order to determine appropriate filters applied to incoming packets. Since packet formats are not verbose, they do not need to optimize for sender-specific packet formats.

XML schema-based optimization techniques [20, 8, 9] correspond to the packet filtering techniques in terms of XML processing because packet formats can be regarded as schemas for packets. Both do not optimize for variable wire representation of XML messages or network packets.

5.3 Delta-encoding

Techniques for compressing a data object like a file relative to another object is called delta-encoding [1, 19], and have been applied to network data transmission [23, 7, 26, 12], just to mention a few. Since delta-encoded XML documents have explicit differential information for previously-received documents, it is more straight-forward to reuse the results of previous processing.

However, in order to use delta-encoding, document producers and consumers must make some agreements with each other for encoding methods, which are application specific. Without standard specifications and recommendations for the encoding format, the original content of XML documents cannot extract from the encoded documents. Such XML documents are no more interoperable in terms of the standard specifications.

6. CONCLUDING REMARKS

This paper proposed an XML parser named Deltarser, which has a novel mechanism for efficiently processing XML documents in most cases of XML-based infrastructural middleware like Web Services application servers. Given a new

XML document in a byte sequence, the XML parser proposed in this paper usually does not analyze the XML syntax but just compares the byte sequences with those which have already been processed. It reuses the processed results stored before.

The SAX implementation of our parser complies with the JSR 63 standard, and the Java API for XML Processing (JAXP) 1.1 specification. In an experiment with a message for Google Web Services, it parses 126% faster than Apache Xerces and 73% faster than Piccolo at its best.

7. REFERENCES

[1] M. Ajtai, R. C. Burns, R. Fagin, D. D. E. Long, and L. J. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *JACM*, 49(3):318–367, May 2002.

[2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. In *Proc. OSDI'94*, pages 115–123. USENIX, 1994.

[3] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, and P. Yendluri. Xml basic profile. W3C Note April 14 2000, http://www.ws-i.org/Profiles/BasicProfile-1.0.html.

[4] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. W3C Recommendation, January 14, 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (third edition). W3C Recommendation February 04 2004, http://www.w3.org/TR/REC-xml-20040204.

[6] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient wire formats for high performance computing. In *Proc. SC2000*. IEEE Computer, 2000.

[7] M. C. Chan and T. Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proc. INFOCOM'99*, pages 117–125. IEEE, 1999.

[8] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proc. HPDC-11*, pages 246–254. IEEE Computer, 2002.

[9] K. Chiu and W. Lu. A compiler-based approach to schema-specific XML parsing. In *Proc. Workshop on High Performance XML Processing, May 18, New-York, NY, USA*, 2004. Online publication.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, March 15, 2001, http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[11] D. Davis and M. Parasha. Latency performance of soap implementations. In *Proc. CCGrid'02, Workshop on Global and Peer-to-Peer on Large Scale Distributed Systems, Berlin, Germany, May 22-24, 2002*, pages 407–412. IEEE Computer, 2002.

[12] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proc. 2003 USENIX Annual Technical Conference*, pages 113–126. USENIX, 2003.

[13] C. Fry. JSR 173: Streaming API for XML. Java Community Process Specification Final Release, March 25, 2004.

[14] Google. Google web APIs. http://www.google.com/apis/.

[15] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and evaluation of RMI protocols for scientific computing. In *Proc. SC'00*. IEEE Computer, 2000.

[16] J. Knutson and H. Kreger. Web services for J2EE, version 1.0. Java Community Process Specification Final Release, Septemer 21, 2002.

[17] J. Knutson and H. Kreger. Web services for J2EE, version 1.1. Java Community Process Specification Public Draft, October 10, 2002.

[18] C. Kohlhoff and R. Steele. Evaluating SOAP for high performance business applications: Real-time trading systems. In *Proc. WWW2003*. ACM, 2003.

[19] D. G. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proc. 2002 USENIX Annual Technical Conference*, pages 219–228. USENIX, 2002.

[20] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-based XML query processor. In *Proc. VLDB'02*, pages 227–238. Morgan Kaufmann, August 2002.

[21] U. Manber. Finding similar files in a large file system. In *Proc. USENIX Winter 1994 Technical Conference*, pages 1–10. USENIX, January 1994.

[22] N. Mitra. SOAP version 1.2 part 0: Primer. W3C Recommendation, June 24, 2003, http://www.w3.org/TR/2003/REC-soap12-part0-20030624.

[23] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proc. SIGCOMM'97*, pages 181–194. ACM, 1997.

[24] R. Mordani. JSR 63: Java API for XML processing 1.1. Java Community Process Specification Final Release, September 10, 2002.

[25] S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating Web Services based implementations of GridRPC. In *Proc. HPDC-11*, pages 237–245. IEEE Computer, 2002.

[26] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. SIGCOMM'00*, pages 87–95. ACM, 2000.

[27] The Apache Software Foundation. Apache Xerces. http://xml.apache.org.

[28] R. A. van Engelen. Pushing the SOAP envelope with web services for scientific computing. In *Proc. ICWS'03*, pages 346–352. CSREA Press, June 2003.

[29] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proc. USENIX Winter 1994 Technical Conference*, pages 153–165. USENIX Association, 1994.

[30] Yuval Oren. Piccolo xml parser for java. http://piccolo.sourceforge.net/.