Eighth International Conference on Web Engineering

Exploiting Structure Recurrence in XML Processing

Dong Zhou
DoCoMo USA Labs
zhou@docomolabs-usa.com

ABSTRACT

Transmitting, parsing, and transforming XML documents (and messages) are particularly costly in cellular environments because of the limitations in handset and access network capabilities. A big part of XML processing cost is caused by the processing of the structure of the documents. By structure we refer to the entity resulted from an XML document after removing the text nodes and attribute values of the document. While XML is a flexible, extensible language, real-world data exchanged in XML often exhibit some degree of stability in its organization. In other words, a computer receiving an XML data item of certain structure is likely to encounter the same structure among future data items. Since most structure-related processing is identical for data items with identical structure, it is thus evident that the overall performance of XML processing will improve if redundancy in structure related processing can be reduced. In this paper we present the concept of Structure Encoding and the approaches to quickly identifying recurring structures, including one relying on collision-resistant hash function. The paper then describes in detail techniques to improving the performance of XML transmission, tokenization, parsing, and transformation by using Structure Encoding. Evaluation experiments with our prototype implementation and industry benchmark suite demonstrate huge performance improvement potential in the presence of structure recurrence: up to 7 times faster for DOM-style parsing, up to 38 times faster for transformation, and up to 97.4% in size reduction when 20% of the text and attribute values change. In the worst case when there is no structure recurrence, structure encoding causes an overhead of about 11.1% for DOM-style parsing and about 8.9% for transformation.

1. INTRODUCTION

XML and Web Services are becoming important constituents of computing, and are starting to be adopted by the cellular world. This trend is manifested by Open Mobile Alliance's bracing of XML in its specifications, and by Nokia's recent adoption of a service-oriented Web Services (which is based on XML) framework for mobile phones [1]. XML data processing (including, but not limited to, transmission, parsing and transformation), however, is costly compared to native data processing, especially on mobile handsets. Because of the gaps in processor frequency, memory bandwidth and access latency, and software maturity (such as the existence/maturity of the Java JIT compiler), today's smartphones are 1 to several orders of magnitude slower than desktop computers for processing intensive applications. This, combined with lower bandwidth and frequent interference in mobile access networks, translates to longer latencies and more energy consumption for XML processing on mobile handsets. It is thus of particular importance to provide efficient middleware for XML processing in cellular environments.

While XML is a flexible, extensible language, real-world data exchanged in XML often exhibit some degree of stability in its organization. In other words, a computer receiving an XML data item of certain format is likely to encounter the same format among future data items. For example, SOAP requests received by a weather forecast Web Service might have the same form, and so might the responses that it sends back. As another example, financial data and sports scores broadcasted to mobile devices are often in unvarying organization. Similar are the protocol messages used for mobile device management and mobile data synchronization [22], the data passed around with XML-RPC [21], RSS feeds from a Web site [23], and the results from Web Continuous Query systems [11]. In each of these examples, although the way a messages or data is organized is not fixed, it is usually quite stable. In other words, the *structure* of XML-described data changes much less frequently than the real content of the data.

By structure we refer to an entity constructed from an XML document by removing the text nodes and attribute values of the document (see section 3 for a precise definition). The structure of a document is much more stable than the document itself: other XML constructs, such as processing instructions and element names and namespaces, changes much less frequently than text node values, and attribute names are nearly constant compared to changing attribute values.

The cost of structure-related processing, however, is a big part of the overall XML processing cost. Structure needs to be transmitted along with the rest of the document. Syntax checking for markups is as costly as, if not more so than, syntax checking for text nodes. So is validation. Well-formedness checking and tree building are almost exclusively for structure constructs. Finally, in stylesheet transformation, many expensive operations, such as node selection and template rule matching, are usually operated on structure information alone. Furthermore, such structure-related processing is usually identical for documents with identical structure. It is thus evident that the overall performance of XML processing will improve if redundancy in structure related processing can be reduced.

In this paper we present the concept of Structure Encoding. We describe approaches to quickly identifying recurring structures, including one that treats document structure as a string, and uses message digest generated by collision-resistant hash function as ID of the structure. Structure Encoding explores various techniques to improve the performance of XML tokenization, parsing, transformation, and transmission, by exploiting the recurrence of document structure.

We have implemented prototypes of Structure Encoding based XML parser and transformer, by extending the kXML parser [9] and the XT XSLT processor [12], respectively. Evaluation experiments demonstrate that Structure Encoding offers a median potential speedup of over 4.34 (up to 8 in best cases) for parsing, and a median potential speedup of 5.38 (up to over 39 in best cases) for transformation. In adversary cases, when there is no structure recurrence, the median overhead is about 11.1% for parsing, and about 8.9% for transformation even after the cost of ID generation through hashing has been taken into account.

Structure Encoding is particularly well-suited for mobile environments where a proxy (such as a WAP gateway) mediates between a mobile device and the servers on the Internet. The close-coupling between a mobile device and its service gateway allows efficient Structure Encoding based XML transmission over wireless access network, without violating standard conformance and the loose-coupling between the mobile device and the servers on the Internet. With Structure Encoding, our experiments demonstrated up to 97.4% in size reduction when 20% of the text and attribute values change. Such high efficiency in transmission also further improves the efficiency in parsing and transformation, as it reduces the time required for tokenization.

In the rest of this paper, section 2 discusses related work. Section 3 presents Structure Encoding in detail. In section 4, we describe our current implementation, and in section 5 we describe evaluation experiments and analyze experiment results. We conclude the paper with brief deliberation on limitations of Structure Encoding and our future work in this area.

2. RELATED WORK

Related work in XML processing optimization can be roughly categorized into the following:

API specialization, where different application programming interfaces (APIs) are proposed to meet the needs of different applications. For example, the Push API (such as SAX [2]) is usually used when there is no need to create a document tree; the Pull API (such as XML PULL [3]) is a refinement over Push API so that the application, rather than the parser, takes control; and the Tree-base API (such as DOM [4]) is used when the application needs to navigate the document. Another direction in this area is API simplification, where complex functionalities are intentionally left out, so that the system requires less resource and executes more efficiently. For example, kXML provides a tree-based document access without fully implementing the DOM specification, so that it can fit into devices with little memory [9]. The concept of Structure Encoding is independent of any specific API. Existing and new APIs can be ported to be Structure Encoding based. However, potential gains from utilizing Structure Encoding are different for different APIs. For example, a DOM-style API will benefit more than a SAX parser as the former does more structure-related processing.

Data structure optimization, where the effort is on the efficiency of the internal and external representation of XML documents. For example, binary XML (such as WBXML [5]) uses more efficient representation to reduce storage and transmission size of XML documents; esXML uses flexible internal representation to reduce document update costs [6]; and VTD-XML uses special data structure to avoid extracting data from XML documents [7]. Work in this area can generally be adopted in Structure Encoding implementation to achieve greater combined performance improvement.

Pre-processing, where some steps of XML processing is conducted ahead of the time, so that the amount of processing needed after document arrival is reduced. XSLTC is such an example. It offline compiles an XML stylesheet into executable code used for directly transforming source document. Structure Encoding based XSLT transformation goes a step further in pre-processing: it pre-processes the combination of a stylesheet and a document structure, instead of only a stylesheet, thus further reduces the amount of online processing.

Structure-related optimization, where the structure of an XML document and the real content of the document are treated with different optimization approaches. Work in this area is closely related to Structure Encoding. The only work we are aware of in this area is XMill, which compresses document structure separately from the rest of the document. Structure Encoding is broader in structure-related optimization in that it targets not only compression but also other processing, including parsing and transformation. For compression, Structure Encoding focuses on inter-document optimization, while XMill mostly focuses on intra-document optimization. There is also work on using parsers dedicated to specific schemas to improve XML parsing performance [27][28]. The approach, however, is not applicable to aspects of XML processing other than parsing. Structure is also used in the context of lazy XML parsing to defer actual parsing until demanded [29][30].

Processing result caching, where previous processing results are cached and can be used to expedite later processing. An example is Deltarser, which uses a state-machine based approach to cache parse events resulted from state transitions, and reuses such parse events for matching byte sequences. Our work also falls into this category. Specifically, we cache structure-related processing results. Compared with Deltarser, Structure Encoding is more aggressive in reusing processing results, as we extend result reusing to tree-building and XSLT transformation.

Hardware acceleration, where either redundant hardware is used to parallelize or pipeline XML processing steps [13][14], or dedicated hardware is used to speedup bottleneck operations such as XPath evaluation [15]. Hardware acceleration typically either focuses on processing throughput or requires complex special hardware. In contrast, Structure Encoding focuses on processing latency and doesn't require special hardware support.

Delta-encoding, which is a technique used for optimizing network transmission by only transmitting the difference between a file and one of its preceding files [25][26]. Structure Encoding, when used for transmission, can be considered as a special form of delta-encoding. However, Structure Encoding is not limited to transmission: it extends the concept of "delta-transmission": into "delta-processing", i.e., it optimizes XML processing by, attempting to, only transmitting, parsing, and transforming the difference between a file and one of its preceding files.

In native data processing area, PBIO is an efficient wire format for high performance data exchange in heterogeneous environments [18]. It uses format server to dynamically register data formats, and it dynamically generates data unmarshalling code for a data format to speed up the exchange of data of that format [19]. While PBIO is used for the exchange of in-memory data structure, Structure Encoding is for the exchange of text-form XML documents which require optimizations in syntax checking and well-formedness checking. Structure Encoding also offers optimization techniques for document tree-building and transformation which are not addressed in PBIO. While PBIO and Structure Encoding both reduces the costs for the transmission of data formats/structures, Structure Encoding also eliminates the transmission of recurrent data field values.

3. STRUCTURE ENCODING

In this section, we first give our definition of document structure, and briefly describe approaches to identifying document structure. The rest of the section then discusses in detail techniques for improving XML processing performance by exploiting document structure recurrence.

3.1 Document Structure

In Structure Encoding, the *structure* of an XML document is derived from the serialized text of the document, after removing non-whitespace text nodes and after "canonicalizing" element tags. "Canonicalizing" element tags includes "whitespace canonicalization" and "attribute canonicalization". "Whitespace canonicalization" removes any optional whitespaces in element start tags and end tags, and replaces any required whitespace with a single space character. "Attribute canonicalization" removes '=' character, the attribute value, and any delimiters surrounding attribute value.

Note that namespace declarations are not "canonicalized", as we believe that namespace attribute values are much more stable than other

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
   <title>Book on XML</title>
   <author>Foo</author>
   <price>49.99</price>
   <special>34.99</special>
   <size cover="Paperback">
      974 pages; Dimensions(in inches): 2.08x9.00x7.26
   </size>
   <publisher>ABC Inc; 1st edition (Jan. 2002)</publisher>
   <isbn>1111111111</isbn>
</book>
```

Figure 1. An XML Document for Book Record

attribute values. More importantly, having static namespace declarations enables the optimization of operations such as well-formedness check for element names.

Figure 1 shows a simple XML document for a book record. The corresponding structure of the document is shown in Figure 2. Figure 3 depicts one probable tree-style memory representation of the document structure. Note that in this figure, except for pointers to attribute values and text nodes, all other pointers have been assigned and each points to either another node or a character string. That is, two different book record documents will only differ in pointers to text nodes and attribute values.

3.2 Identifying Recurring Structure

Identify recurring structure requires approaches to quickly map a document structure to an identifier (ID). The mapping can be either based on explicit-naming or based on hashing:

- Explicit-naming based approach: in this approach, the application that generated the document explicitly associates an ID to the document. The application is responsible in guaranteeing that documents with different structures are associated with different structure IDs.
- Hash based approach: in this approach, a collision-resistant hash function (such as MD5 or SHA1) generates digest for a document structure, and uses the digest as the structure ID.

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
   <title></title>
   <author></author>
   <price></price>
   <special></special>
   <size cover>
   </size>
   <publisher></publisher>
   <isbn></isbn>
</book>
```

Figure 2. Structure of the Book Record

The hash-based approach incurs processing overhead, but it makes it easier for making structure ID's globally unique: although hash functions such as MD5 are not collision-free, the possibility of collision is extremely low when digests are sufficiently long (e.g., 128-bit), so digests generated by such functions are practically globally unique IDs.

This mapping from structure to structure ID can be done by either the sender or receiver of the document:

- Sender ID Assigning: in this approach, the sender (or any computer between the sender and the receiver) uses either explicit-naming or hashing to generate ID, and uses a special XML processing instruction to embed the ID into the XML document. Figure 4 shows an example.
- Receiver ID Assigning: in this approach, the receiver uses hash based approach to generate IDs and identify recurring structures. Since hashing itself incurs processing overhead, this approach makes more sense when either hashing is very fast (e.g., there is a hardware

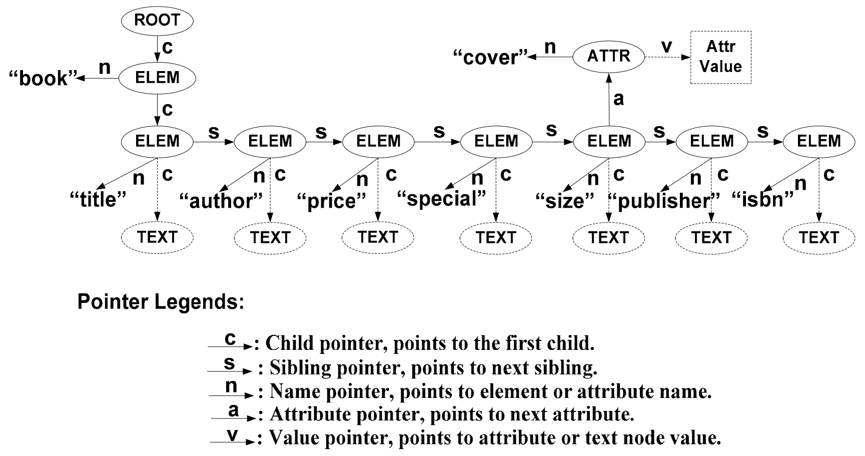

ROOT
ELEM
"book"
ATTR
"cover"
Attr Value
ELEM
"title"
"author"
"price"
"special"
"size"
"publisher"
"isbn"
TEXT
c
s
n
a
v
Pointer Legends:
c: Child pointer, points to the first child.
s: Sibling pointer, points to next sibling.
n: Name pointer, points to element or attribute name.
a: Attribute pointer, points to next attribute.
v: Value pointer, points to attribute or text node value.

Figure 3. Visualizing Structure of the Book Record Document

```
<?xml version="1.0" encoding="UTF-8"?>
<?se id=c5bae136d32e1d7763a1a970d186e0ff?>
<books>
…
</books>
```

Figure 4. Sample Document with Structure ID Embedded

hash implementation), or costly further processing (such as tree-building or transformation) on the document is expected.

3.3 Exploiting Structure Recurrence

Structure information can be exploited to speedup various XML processing stages: from tokenization, well-formedness check and validation, to tree building, transformation and transmission. In this subsection we describe in detail how Structure Encoding improves performance in each of these stages.

3.3.1 Tokenization

Tokenization scans the source document, checks syntax, and generates tokens. Tokens have types as well as values. In Structure Encoding, token types include: Start Element, End Element, Immediate Close, Attribute Name, Attribute Value, Namespace Attribute Name, Namespace Attribute Value, Text, Whitespace, Comment, Doctype, and Processing Instruction. Note that an attribute is composed of two tokens, an Attribute Name token, and an Attribute Value token. Also notice that namespace attributes are treated differently from other attributes.

Tokens values are in the form of strings. For example, the value of an Attribute Name token is the name of the attribute, while the value of an Attribute Value token is the value of the attribute.

It is quite evident that two token lists generated from two documents of the same structure have tokens of exactly the same types, and they appear in the same order. Furthermore, other than tokens of types Text and Attribute Value, token values are also the same. This has two implications: one is that they occupy the same number of characters in the source document; the other is that if the token in the first document is syntactically correct, then the corresponding token in the second document must also be syntactically correct.

Structure Encoding exploits these facts. It caches token lists for document structures. When a new document comes in, it first acquires the structure ID of the document by either hashing or extracting the ID embedded in the document. It then uses the ID to retrieve cached token list for the document structure. Tokenization of the incoming document can be greatly improved in case of a cache-hit because:

- The type of next token is known, so there is no need to identify token type of the next construct.
- For most tokens, the tokenizer only needs to skip a known number of characters. For example, if next token is of type Doctype, then it only needs to skip a number of bytes based on the length of the value of the cached token.
- For most tokens, the tokenizer does not need to store values for them as it can share with the cached token. In above case, there is no need to create a string to store the Doctype construct.

3.3.2 Well-formedness Check and Validation

Well-formedness check in a typical XML parser verifies that:

- There is exactly one top-level element.
- All open tags have a corresponding close tag or an immediate close construct.
- All tags are correctly nested.
- Attributes of an element have different names.
- Namespace attributes of an element have different names.
- All namespace prefixes have corresponding namespace attributes defined in scope.

Note that none of these activities involve Text and Attribute Value tokens. As a result, if document D1 is well-formed, and document D2 has the same structure as D1, then D2 must also be well-formed. So there is no need for well-formedness check on D2.

XML validation checks if a document conforms to the rules of a Document Type Definition (DTD) or a Schema. Continuing with above example, if D1 is valid for schema S, then D2 must also be valid for S if its Text and Attribute Value tokens are valid for S. That is, validation of D2 is reduced to the validation of its Text and Attribute Value constructs.

3.3.3 Tree Building

Tree-based parsers need to build a tree representation of the document. This tree building process involves operations such as the allocation (and later de-allocation) of tree nodes, assign values to allocated nodes, and the linking of allocated nodes into a tree. Using structure information, once we know that the incoming document (D2) has identical structure as a previous document (D1), tree operations can be optimized in following ways:

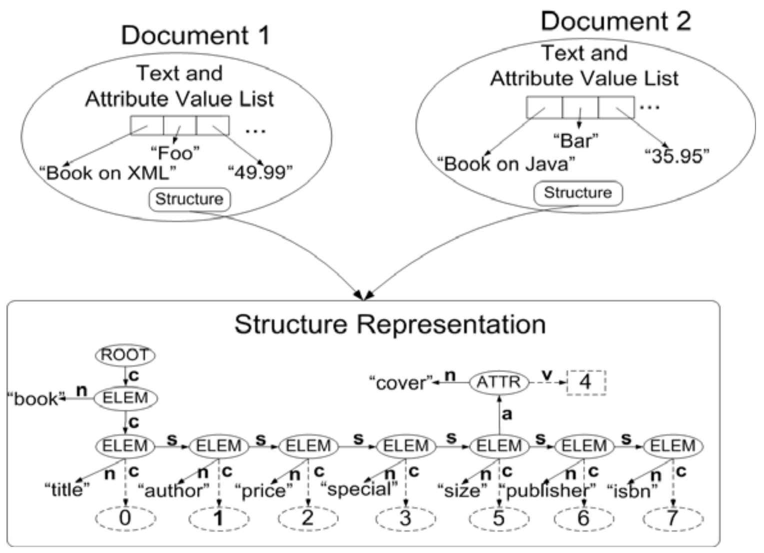


Figure 5. Documents of Identical Structure Sharing Document Tree

- Tree reuse: if D1 had previously constructed a tree and no longer needs it, then D2 can reuse it. All it needs to do is to reassign values for text and attribute nodes. Values for text nodes are extracted from corresponding Text Tokens, while the values of attribute nodes are extracted from corresponding Attribute Value tokens. Since, for a given value, the rank of its corresponding token in the token list is fixed, this reassignment process is extremely simple. This tree reuse approach obviates the operations for D1 to de-allocate (or garbage collect) the tree, the operations for D2 to allocate objects for each of its nodes and assign node values, as well as the operations for D2 to link these nodes.
- Fast tree duplication: in case D1 still needs its document tree, a duplicate of D1's document tree is first made for D2. Fast tree duplication is possible by either pre-duplication when system is idle, or by re-organizing (also could occur when system is idle) D1's document tree so that nodes of the tree are aligned in an array, and that tree duplication is reduced to one simple memory copy.
- Tree sharing: by replacing pointers for text and attribute value pointers with corresponding offsets in token lists, D1 and D2 can share the document tree. This tree sharing approach is ideal for reducing memory requirements if system expects concurrent use of documents of identical structures (such as in a concurrent server environment). Figure 5 illustrates a simple implementation of two in-memory documents of a same structure sharing the in-memory representation of the structure. Each document contains a pointer to the structure, along with a list of texts and attributes values.

3.3.4 Transformation

An XSL-based transformer generates output from a source document (D) with a stylesheet (SS). If the system has SS and the structure of D (ST) before D arrives, then this transformation can be split into two phases:

- The Pre-Processing (PreP) phase, which occurs before D arrives. It takes SS and ST as input, and generates *Stencil* (SN, see below) as output.
- The After Arrival Processing (AAP) phase, which occurs when D arrives. It uses SN and D's token list as input, and generates the same output as would a usual XSLT processor for SS and D.

PreP is a partial transformation. It is a form of partial evaluation based on the partial document information contain in the structure of the document. Many costly XSLT operations, such as template matching and node selection, can be carried out in the PreP phase, so that they won't be involved in the AAP phase, thus reducing the latency between the arrival of the document and the completion of the transformation.

The result of the PreP phase is a *Stencil* which, assuming transformation result is in the form of text, is a list containing elements of the following types:

- String, which is the result of fully executed XSLT operations. (Note that there are no adjacent strings in the list as they can be concatenated)
- *Future Expressions*, which can only be fully evaluated after the document is received. Future Expressions can't be resolved during the PreP phase as their values depend on the values of text nodes and/or attributes. Future Expressions are evaluated at the AAP phase, taking the token list of the incoming document as input.
- Branches, which contains a Conditional Expression, a True Stencil which is selected when the Conditional

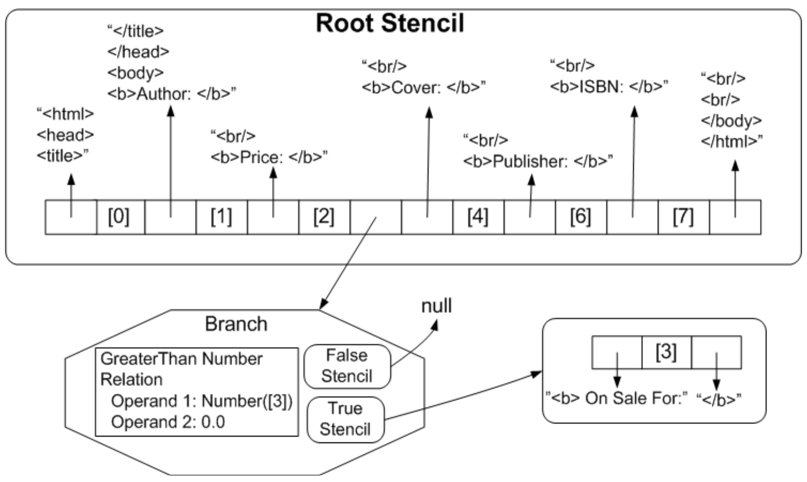


Figure 6. An Example of Transformation Stencil

Expression yields true, and a False Stencil which is selected when the Conditional Expression yields false.

A sample Stencil is shown in Figure 6 and described in section 4.3.

The AAP phase uses the token list of the document to resolve the Stencil, by resolving the elements in the Stencil in order. String elements are directly emitted to the output. Future Expressions are evaluated, the result of which then converted to String and sent to the output. For each Branch element, depending on the evaluation result of its Conditional Expression, either the True Stencil or the False Stencil is selected and recursively resolved.

Note that Future Expressions can only take document token list as input and doesn't have access to XSLT variables. As a result, if an expression depends on an XSL variable whose value can't be determined at PreP phase (because its value is conditionally updated by a conditional element), then there is no transformation optimization for this combination of stylesheet and structure.

3.3.5 *Compression for Efficient Transmission*

Efficient transmission of XML documents is particularly important in mobile environments. Structure recurrence can be exploited for compression and efficient transmission, by avoiding the redundant transmission of structural information and by compressing recurring test nodes and attribute values:

- Assuming sender S first sends document A to receiver R, and later sends to R document B which has the same structure as document A, if S knows that R has locally kept the structure information for A, then S only needs to send the structure identification of B, along with the text node and attribute values of B. Efficiency is achieved by replacing the structure of document B with the (much shorter) identification of the structure of the document.
- Further, the text node and attribute values in document A has a 1-to-1 mapping with those in document B. If sender S is aware that receiver R also keeps text node and attribute values of document A, then when S detects that a value in B is the same as the corresponding value in A, S only needs to signal R to reuse the value in A, instead of retransmitting it.

The combination of the two offers potentially high compression ratio with very low computation overhead.

Note that it is not necessary for the sender and the receiver to be both aware of the existence of the compression. For example, in wireless mobile environments, the gateway of a mobile device can serve as the proxy between the outside world and the mobile device: XML documents are transmitted in compressed form between a mobile device and its gateway, while the gateway and the outside world communicate in usual text form.

4. IMPLEMENTATION

We have implemented in Java a XML tokenizer, a DOM-style parser, and a XSLT processor, based on Structure Encoding. The DOM-style parser is implemented as an extension to kXML, while the XSLT processor is implemented as an extension to XT.

4.1 Tokenizer

The tokenizer operates in three different modes:

- The Swift mode, which is activated when the incoming document contains sender assigned structure ID, and that cached token list for the structure is found.
- The Hash mode, which is activated when the incoming document does not contain sender assigned structure ID. The hash function we use is an implementation of MD5 which generates 128-bit digests.
- The Nature mode, which is normal XML tokenization and is activated under all other conditions.

Under Swift mode, the tokenizer checks each token in the token list, and takes different action depending on the types of the token:

- If it is a Text token, it reads a text string (delimited by '<' and may contain entity references) from the document character stream, and uses the text string to replace the value of the token.
- If it is a Start Element token, the tokenizer first skips a number of characters from the document character stream. The exact number of characters is determined by the length of the name of the element. It then enters a loop that reads attributes of the element. Within the body of the loop, it first checks if the next token is Attribute Name. If it is not, then the processing for the Start Element token completes. Otherwise, it skips the name of the attributes, reads the value of the attribute and updates the value of the next token (which must be of type Attribute Value).
- In all other cases, the tokenizer simply skips a number of characters. The exact number of characters is determined by the type of the token and the length of the value of the token.

In the Nature mode, the tokenizer reads in XML constructs, makes sure that they are syntactically correct, and then converts them into tokens. Operations in Hash mode are mostly the same, except that the tokenizer also needs to send structure characters to the hash function for ID generation.

4.2 Structure Encoding Based Parser

Figure 7 shows a structure encoding based XML parser. It has three components: The Tokenizer as described above;

- The Structure Manager, which contains a cache manager that manages the reuse of document trees, and an optional Structure Optimizer, which does tree optimizations described in section 3.3.3.
- The Controller, which includes an interface with the kXML DOM parser. It also includes a Value Loader which does text node and attribute value reassignments.

The Controller falls back to kXML DOM parser, by passing to it a SAX parser implemented on top of the tokenizer, when it is unable to get a reusable document tree from the Structure Manager. Each document structure has a mapping table that associates a text node or an attribute value object with a token in the token list. The Value Loader in the Controller uses this mapping table to quickly grab text or attribute values from incoming document's token list, and reassigns them to corresponding text node or attribute value objects. Figure 8 illustrates using Structure Encoding based parser to parse the book record document shown in Figure 1.

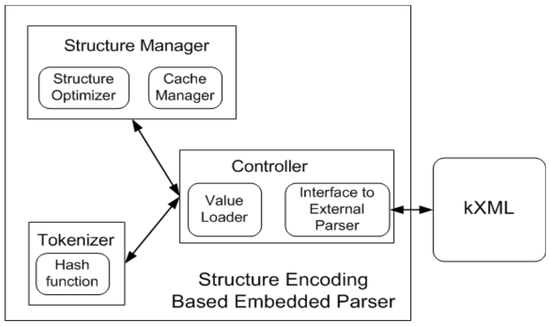


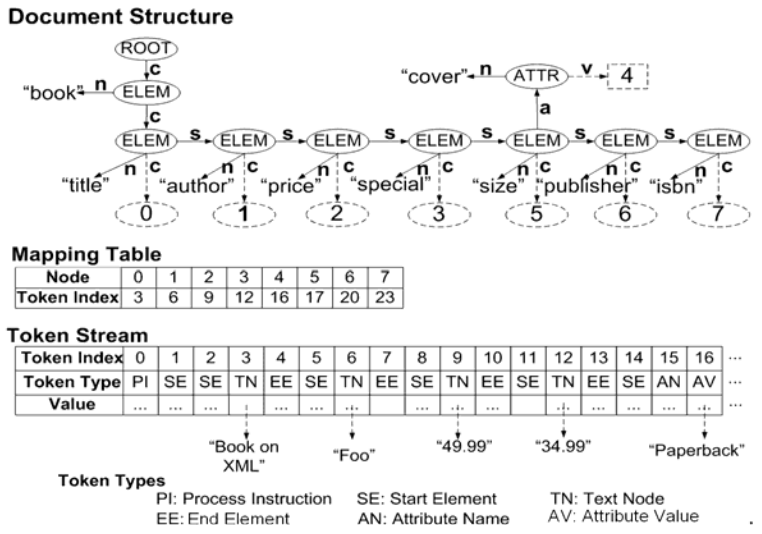
Figure 7. A Structure Encoding Based XML Parser

4.3 Structure Encoding Based Transformer

Figure 9 shows the components of a Structure Encoding based transformer hosted in the XT XSLT processor. Of its four components, the tokenizer and the Structure Manager are the same as in the parser. The two different components are the Pre-processor and the Transformation controller.

Pre-processor carries out the PreP phase of structure encoding based transformation, using stylesheet and document structure to generate transformation Stencil. Within the document structure, the text and attribute nodes are changed to objects of a special type, which, upon access, throws a special exception which uses an integer index to indicate which text node or attribute is accessed. The structure is then treated as a regular document and fed to XT along with the stylesheet. We have also extended XT, so that any expression that may catch the above-mentioned special exception is extended into a Future Expression, which can be evaluated by taking only token list as input. A Future Expression in turn throws itself as a special exception, which is then either caught by another Future Expression or, eventually, by an action. An action generates a result segment, which can be one of the following:

- A string, in the case when there is no special exception caught,
- A Future Expression, when one or more special exceptions are caught, and
- A Branch, when the action is an IfAction generated from an xsl:if or xsl:choose XSL element.The Stencil Resolver in the Transformation Controller is called in the AAP phase after the source document arrives and a transformation is requested. It retrieves the Stencil using the stylesheet and the document structure as key, then uses the token list for the source document as input to



Mapping Table

Node	0	1	2	3	4	5	6	7
Token Index	3	6	9	12	16	17	20	23

Token Stream

Token Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	···
Token Type	PI	SE	SE	TN	EE	SE	TN	EE	SE	TN	EE	SE	TN	EE	SE	AN	AV	···
Value	…	…	…		…	…	…		…	…	…		…	…	…	…	…	···

Token Types

PI: Process Instruction	SE: Start Element	TN: Text Node
EE: End Element	AN: Attribute Name	AV: Attribute Value

Figure 8. Example of Structure Encoding Based

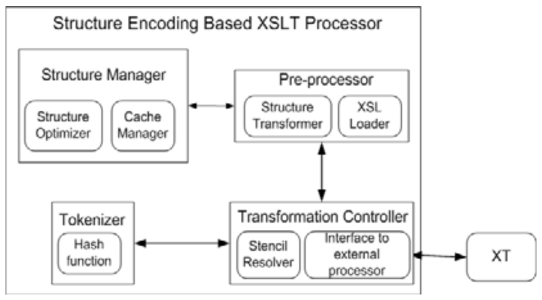


Figure 9. Components of a Structure Encoding Based XSLT Processor

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="book">
  <html>
      <head>
      <title><xsl:value-of select="title"/></title>
    </head>
    <body>
        <b><xsl:text>Author: </xsl:text></b><xsl:value-of select="author"/><br/>
        <b><xsl:text>Price: </xsl:text></b><xsl:value-of select="price"/>
        <xsl:if test="special > 0">
          <b><xsl:text> On Sale For: </xsl:text><xsl:value-of select="special"/></b>
        </xsl:if><br/>
        <b><xsl:text>Cover: </xsl:text></b><xsl:value-of select="size/@cover"/><br/>
        <b><xsl:text>Publisher: </xsl:text></b><xsl:value-of select="publisher"/><br/>
        <b><xsl:text>ISBN: </xsl:text></b><xsl:value-of select="isbn"/><br/>
        <br/>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Figure 10. Transforming the Book Record into

solve the Stencil, using methods described in section 3.3.4.

Figure 10 shows a simple XSL stylesheet used to convert the book record document shown in Figure 1 into HTML format. Note that it contains an xsl:if element, which tests if the "special" price is greater than zero, and if the condition is true, it outputs an on-sale price. Figure 11 shows the result of the

```
<html>
<head>
<title>Book on XML</title>
</head>
<body>
<b>Author: </b>Foo<br>
<b>Price: </b>49.99<b> On Sale For: 34.99</b>
<br>
<b>Cover: </b>Paperback<br>
<b>Publisher: </b>ABC Inc; 1st edition (Jan. 2002)<br>
<b>ISBN: </b>1111111111<br>
<br>
</body>
</html>
```

Figure 11. Result HTML File from Sample Transformation

transformation.

Figure 6 shows the Stencil generated by the PreP phase, where '[*i*]' denotes the string value of the *i*th entry in the mapping table (see section 4.2). The "Root Stencil" contains a Branch, which has a "GreaterThan Number Relation" that can't be pre-evaluated, as it depends on the number value of a Text or Attribute Value token. The True Stencil of the Branch is a simple Stencil, while its False Stencil is empty.

4.4 Structure Encoding Based Compression

The implementation of Structure Encoding based compression for transmission is straight forward. The sender and receiver both maintain an encoding table. Each entry of the table is a mapping from a structure ID to a cached document structure as well as a list of template values for attributes and text nodes associated

Table 1. XML Documents Used in the Experiments, and Their Corresponding Stylesheets

Document	Size (in KB)	Stylesheet
axis	0.38	Tests XPath selection along the different axes
backwards	2.62	Reverses order of elements using the document used in game
chart	1.29	Generates an HTML chart of some sales data
book	1.21	Convertin book record to HTML
game	2.62	Produces a HTML table of the data
midsummer	146	Converting the play to HTML
nitf-stylized	5.79	NITFML to HTML
recipes	16.7	Converting Recipes in XML to HTML
sort	10.3	Sorting input tree according to element name
sp	3.53	Web site construction kit
total	1.31	Reports on sales data
trend	1.9	Computes trends in the input data.
wai	0.58	Schematron validator for WAI docs

Table 2. Measurement Results (all numbers are in milliseconds)

Document	**Tokenization**			**Parsing**					**Transformation**				
	Nature	**Swift**	**Hash**	**kDOM**	**100% Cache Hit**		**0% Cache Hit**		**XT**	**100% Cache Hit**		**0% Cache Hit**	
					Sender Assign	Receiver Assign	Sender Assign	Receiver Assign		Sender Assign	Receiver Assign	Sender Assign	Receiver Assign
axis	7.1	4.8	9.5	11.4	5	9.6	10.7	13	36.3	6.9	11.4	36.3	39.3
Back-wards	33.3	7.4	43.4	60.7	7.6	43.7	56.1	66.6	232.2	12.9	49.5	232.7	252.4
chart	18.5	7.7	24	32.1	7.8	24.9	30.1	36.3	117.8	21.9	39.7	118	129.3
book	15.1	8.6	18.2	24.4	8.8	18.5	21.1	25.5	48.3	11.5	20.9	48.9	55.2
game	33.1	7.2	43.2	60.7	7.6	43.5	56.4	67.3	92.2	8.9	45.1	92	105.5
mid-summer	1990	913.4	2311.5	4686.9	997	2332.5	4012.8	4360.8	7398.5	1475	3068	7268.1	7555.2
nitf-stylized	67.3	25.1	76	105.3	27	77.3	93.8	107.7	178.3	34.4	85	188.5	204.3
recipes	164.3	99	180.1	249.2	107.6	184.8	209.9	234.3	574.4	144.3	224.1	566	585.1
sort	129.8	32.2	159.8	248.2	33	163.6	242.3	281.5	1094.9	55.6	188.1	1121.9	1174.4
sp	36.2	21.5	40	53.7	23.2	40.6	45.5	51.3	122.8	28.2	45.2	126.6	138.9
total	18.6	6.8	24.5	32.4	7.4	25	30.6	36.2	53.9	7.6	25.5	54	61.5
trend	36.5	10.8	41	62.1	10.4	41.1	66.3	70.9	2249.3	56.9	89.1	2249.3	2279.5
wai	9.1	4.5	12.5	14.2	4.6	12.6	13.4	17.1	96.4	7.7	15.8	99.9	105

with the structure (Note that document structures and template values can be flushed out of the cache and reconstructed from files stored in persistent storage.).

The first document of a given structure will be sent uncompressed. But an entry will be added to the encoding tables on both sender and receiver sides, with the attribute and text values of this first document used as template values. A following document of the given structure will be encoded as a two-byte integer indicating the index of the structure in the encoding table, followed by the list of attribute and text string values. A special 1-byte symbol replaces any value that is an exact match of its corresponding template value.

Such compression is used between a mobile handset and its service gateway in cellular environments. Note that compression happens on a document send from the handset, through the gateway, to a server on the Internet, as well as on a document received by the handset through the gateway.

5. EVALUATION

Evaluation experiments described in this section are conducted on a TI OMAP 1511 Innovator device, which runs in frequencies up to 200MHz. It has 32MB ROM as well as 32MB RAM. The OS used is a version of embedded Linux, and the JVM used is Intent from TAO Group.

The source documents and stylesheets used for evaluation are randomly selected from Sarvega's XSLTBench [16] and DataPower's XSLTMark [17], after initially excluding some unrealistic transformations. We select these two benchmarks as they are influential industry benchmark and that they both support XSLT benchmarking. Note that some of the large source documents are less likely to have recurrent structures in real-world applications. We included them regardless as they might help us understand the implication of document size on system performance. Table 1 lists sizes of sample documents, and the transformation stylesheets used for them. Table 2 lists the raw numbers used in following discussions. Documents are fully read into memory before any measurement starts to minimize external disturbance. Warm-up runs are always conducted prior to real measurement runs.

5.1 Tokenization

Figure 12 compares the speed of Swift mode and Hash mode against the speed of Nature mode tokenization. The Swift mode is by far the fasted, having a median speedup of 2.40 over Nature mode. The median speed of Hash mode is about 81% of that of Nature mode, which means a median hashing overhead of about 23.5%. In the worst case, this overhead is about 37.4% (for wai).

5.2 DOM-style Parsing

Figure 13 and Figure 14 compare speed of Structure Encoding based parsing against that of kXML parsing for both Receiver ID Assigning and Sender ID Assigning. In Receiver ID Assigning, the underlying tokenizer always uses Hash mode, while in Sender ID Assigning, the tokenizer uses Swift mode when cache hit ratio is 100% (i.e., structure of incoming document is always in cache), and uses Nature mode when cache hit ratio is 0%.

Figure 13 shows the results for the ideal case when cache hit ratio is 100%: the median relative speed is 4.34 when Sender ID Assigning is used, and is 1.35 when Receiver ID Assigning is used. The best case is in backwords and game (which use the same source document), where the relative speed is close to 8 under Sender ID Assigning.

Figure 14 shows the results for the worst case when cache hit ratio is 0%: the median relative speed is 1.07 when Sender ID Assigning is used, and is 0.90 (or an overhead of %11.1) when Receiver ID Assigning is used. The worst sample is wai, where the overhead is about 20.5% under Receiver ID Assigning. The reason

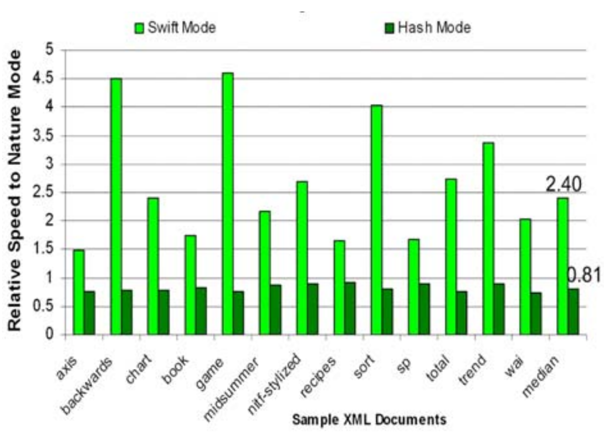


Figure 12. Comparing Performance of Tokenization Modes

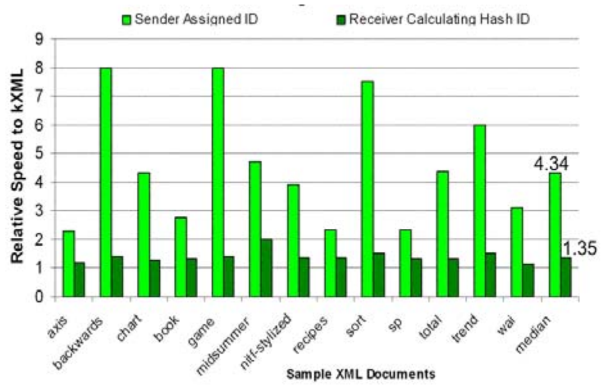


Figure 13. Comparing Structure Encoding Based XML Parsing and kXML Parsing (Cache Hit Ratio = 100%)

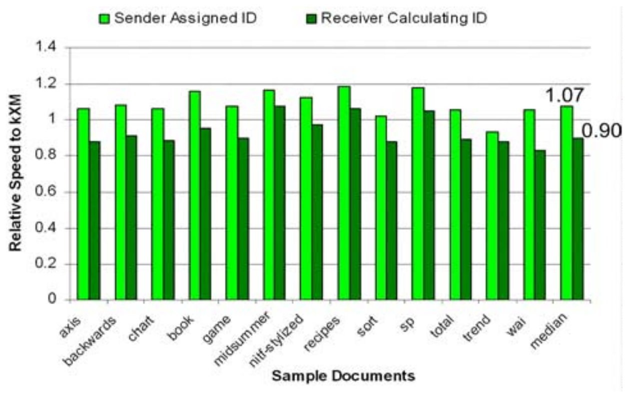


Figure 14. Comparing Structure Encoding Based XML Parsing and kXML Parsing (Cache Hit Ratio = 0%)

that Structure Encoding is a little faster than kXML even when cache hit ratio is 0 is likely because of the fact that, while we use kXML for tree building, we modified it slightly to interface it with our tokenizer, which resulted in differences in execution traces and slight differences in runtime performance when execute in JVM.

Overall, these two figures demonstrate that Structure Encoding based parsing provides substantial speedup when structure recurrence is frequent, and incurs low overhead when such recurrence is rare.

5.3 Transformation

Similarly, Figure 15 and Figure 16 compare transformation speed of Structure Encoding based XSL processor against that of XT for both Sender ID Assigning and Receiver ID Assigning. Again, under Receiver ID Assigning, the underlying tokenizer always uses Hash mode, while under Sender ID Assigning, the tokenizer uses either Swift mode or Nature mode.

Figure 15 shows the results for the ideal case when cache hit ratio is 100%: the median relative speed of Structure Encoding based implementation is 5.38 when Sender ID Assigning is used, and is 2.72 when Receiver ID Assigning is used. The best case is trend, where the relative speed is over 39 for Sender ID Assigning, and over 25 for Receiver ID assigning.

Figure 16 shows the results for the worst case when cache hit ratio is 0%: the median relative speed is 0.996 when Sender ID Assigning is used, and is 0.918 (or an overhead of 8.9%) when Receiver ID Assigning is used. The worst sample is nitf-stylized, where the overhead is about 14.6% under Receiver ID Assigning.

These two figures show that Structure Encoding based XSL processing offers even higher potential speedup than parsing, yet incurs lower overhead in worst cases. This is understandable as, typically, a large portion of operations in transformation is for costly structure-related operations, which we have pre-processed offline. Note that, as long as structure recurrence is frequent, the potential improvement is very high (up to a median relative speed of 2.72) even if the receiver has to calculate structure ID through hashing.

5.4 Compression

We conducted an experiment to examine the effectiveness of Structure Encoding based compression. In the experiment, for each sample document, we randomly changed the values of 3rd, 8th, 13th, … (i.e., one in every 5, starting from the 3rd) attribute and text node values. Thus our experiment

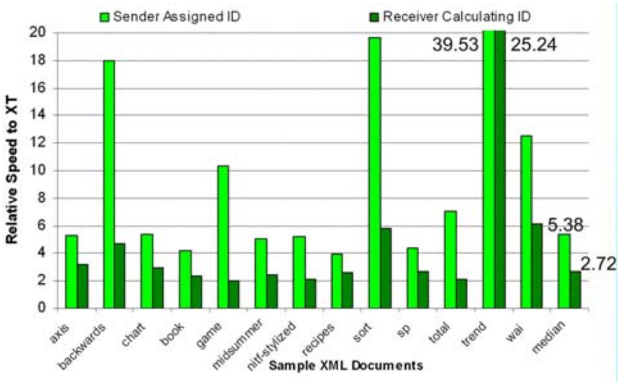


Figure 15. Comparing Structure Encoding Based Transformation and XT (Cache Hit Ratio = 100%)

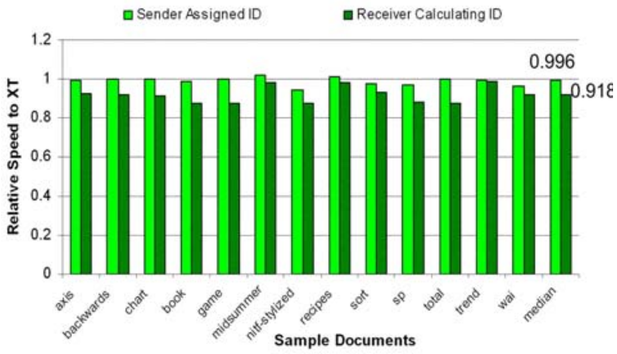


Figure 16. Comparing Structure Encoding Based Transformation and XT (Cache Hit Ratio = 0%)

Table 3. Compression and its effect on tokenization speed (Assuming changes in 20% of the text and attribute values)

Document	Size (in KB)	Comp. Size (Byte)	Nature (ms)	Swift (ms)	Comp. (ms)
axis	0.38	48	7.1	4.8	0.8
backwards	2.62	81	33.3	7.4	1.8
chart	1.29	91	18.5	7.7	1.3
book	1.21	129	15.1	8.6	1.1
game	2.62	81	33.1	7.2	1.8
midsummer	146	20760	1990	913.4	111.7
nitf-stylized	5.79	564	67.3	25.1	4.0
recipes	16.7	2355	164.3	99	11.4
sort	10.3	752	129.8	32.2	8.8
sp	3.53	554	36.2	21.5	2.6
total	1.31	91	18.6	6.8	1.3
trend	1.9	169	36.5	10.8	2.7
wai	0.58	15	9.1	4.5	0.6

assumes that around 20% of the attribute and text values are different from the template values.

Table 3 lists results from this experiment. The third column of the table shows the number of bytes transmitted for each sample document, while the 6th column shows the time used to construct tokens from the compressed document. The table shows that, under above-described setup, the amount of data transmitted are reduced to 2.6% to 15.7% of the original size (second column, listed in KB), and the tokenization time is reduced to 5.4% to 11.3% of the Nature mode tokenization time, or 11.5% to 25% of the Swift mode tokenization time.

5.5 Discussion

Since in our implementations, we make changes to base systems (KXML and XT) only when such changes are required to implement Structure Encoding, performance differences demonstrated in this section are caused by differences in techniques rather than differences in implementations.

Our experiments clearly show that Structure Encoding can, potentially, greatly improve the efficiency of XML processing with relatively low penalty for worst cases. The worst case scenario happens when a receiver uses hash function to identify the structure of a document, only to find out that the document structure is not in cache. The penalty it pays for such worst case scenario, 11.1% for DOM-style parsing and 8.9% for transformation, can easily be compensated by future structure recurrence. Such cache-miss penalty is negligible when sender-assigning scheme is used. Systems that are conscious of such client-side penalty can let the sender or anyone in the middle of the transmission path to assign ID without altering the semantics of the message or document.

Although we didn't measure the impact of compression on parsing and transformation, it can be inferred from tables 2 and 3. For example, in the book case, with tokenization time reduced from 8.6ms (Swift mode) to 1.1ms (compressed), parsing time will likely be further reduced from 8.8ms to around 1.3ms (compared with 24.4ms for KDOM). Similarly, transformation time may be further reduced from 11.5ms to around 4.0ms (compared with 48.3ms for XT). In other words, Structure Encoding based XML compression offers additional, significant, performance improvements for XML parsing and transformation in mobile environments, where closely-coupled proxies commonly exists. With compression turned off, Structure Encoding is fully compatible with Web specifications and can be used between any two Internet hosts, and it still offer very significant performance improvements when there is structure recurrence.

Without compression, for documents with recurrent structures, tokenization and structure hashing cost dominates the overall cost for parsing, and is the major part of the cost for transformation. Tokenization and structure hashing however are relatively simple operations that may be implemented in hardware with low cost. (In fact, some mobile chipsets already have hardware implementation of hash functions such as MD5 for security purposes.) If a hardware implementation can reduce tokenization and structure hashing cost to a fifth of the current cost, then there will be additional substantial improvement for the parsing and transformation of most of the documents used in our experiments.

In our system, a document having a structure slightly different (e.g., added a new element) from a previous structure will not be able to reuse the structure processing result of the previous structure. However, our system pays off as long as this new, slightly different structure recurs in future documents.

6. CONCLUSION, LIMITATIONS, AND FUTURE WORK

In this paper we motivated exploiting structure recurrence to speedup XML processing in mobile environments, by reducing structure related transmission and processing costs. We presented the concept of Structure Encoding, and described approaches to quickly identifying recurring structures, including one using collision-resistant hash function. We explained in detail how to use structure encoding to speedup XML transmission, tokenization, tree-building, and transformation. We described our implementation of structure encoding based tokenizer, DOM parser (based on kXML), XLT processor (based on XT), and compression scheme. Our experiments conducted on a mobile test-bed demonstrated dramatic performance improvement in the presence of structure recurrence and low overhead otherwise. In ideal cases, structure encoding offers speedups of up to 7 for parsing and over 38 for XSL transformation, and up to 97.4% in size reduction when 20% of the text and attribute values change

Structure encoding, however, is not applicable to all XML applications. Rather, it is more applicable to data-centric XML processing than to document-centric XML processing. A user randomly browsing Web pages is not likely to have high structure recurrence probabilities.

Our current implementation does not support documents with "variable-length arrays" – lists of identically structured elements with non-fixed lengths. Otherwise identically structured documents with different array lengths are currently considered as having different structure.

We are currently working on supporting "variable-length arrays" to extend the applicability of Structure Encoding. We are also looking at provide similar, but less aggressive, optimization support for schema-conforming documents.

REFERENCES

[1] Nokia Web Services – Helping Operators Mobilize the Internet. Http://www.projectliberty.org/resources/whitepapers/WS_Operators_A4_0408.pdf.

[2] The SAX Project. http://www.saxproject.org/.

[3] XML Pull Parsing. http://www.xmlpull.org/

[4] W3C Document Object Model. http://www.w3.org/DOM/

[5] WAP Binary XML Content Format. http://www.w3.org/TR/wbxml/

[6] Efficiency Structured XML. http://www.esxml.org/

[7] VTD-XML. http://vtd-xml.sourceforge.net/

[8] XSLTC Documentation. http://xml.apache.org/xalan-j/xsltc/

[9] kXML. http://www.kxml.org/

[10] Liefke, H. and D. Suciu. XMill: An Efficient Compressor for XML Data. In Proc. of the ACM SIGMOD Conference on Management of Data. May, 2000.

[11] Liu, L., C. Pu, and W. Tang. WebCQ: Detecting and Delivering Information Changes on the Web" In the Proceedings of International Conference on Information and Knowledge Management (CIKM), Nov. 7-10, 2000.

[12] The XT XSLT processor. http://www.blnz.com/xt/index.html

[13] Sarvega,Inc. http://www.sarvega.com/

[14] DataPower Technology, Inc. http://www.datapower.com/

[15] Rax Content Processor. http://www.tarari.com/rax/index.html

[16] The Sarvega XSLT Benchmark Study, Sarvega Inc. http://www.sarvega.com/xslt-benchmark.php.

[17] XSLTMark. http://www.datapower.com/xmldev/xsltmark.html

[18] Eisenhauer, G. and L. K. Daley. Fast Heterogenous Binary Data Interchange. In Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000), pp 90-101.

[19] Bustamente, F., G. Eisenhauer, K.Schwan, and P. Widener. Efficient Wire Formats for High Performance Computing. In Proceedings of High Performance Networking and Computing Conference, 2000 (SC'2000).

[20] Toshiro Takase, Hisashi Miyashita, Toyotaro Suzumura, and Michiaki Tatsubori, An Adaptive, Fast, and Safe XML Parser Based on Byte Sequence Memorization. In Proc. of WWW'2005.

[21] XML-RPC. http://www.xmlrpc.com/

[22] Open Mobile Alliance. http://www.openmobilealliance.org/

[23] RSS 2.0 Specification. http://blogs.law.harvard.edu/tech/rss

[24] Open Mobile Alliance. http://www.openmobilealliance.org/

[25] Mogul, J., F. Douglis, A. Feldman, and B. Krishnamurthy. Potential benefits of delta-encoding and compression for HTTP. *In Proc. SIGCOMM'97*, 1997.

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

[27] Chiu, K., and W. Lu. A Compiler-Based Approach to Schema-Specific XML Parsing. In First International Workshop on High Performance XML Processing, May 2004.

[28] Matsa, M., E. Perkins, A. Heifets, M. G.aitatzes Kostoulas, D. Silva, N. Mendelsohn, M. Leger. A high-performance interpretive approach to schema-directed parsing. In Proceedings of the 16th International Conference on World Wide Web, 2007.

[29] Noga, M. L., Schott, S., and Löwe, W. 2002. Lazy XML processing. In Proceedings of the 2002 ACM Symposium on Document Engineering (McLean, Virginia, USA, November 08 - 09, 2002). DocEng '02. ACM, New York, NY.

[30] Farfán, F., V. Hristidis and R. Rangaswami. Beyond Lazy XML Parsing. In Proceedings of the 18th International Conference (DEXA 200), September 3-7, 2007.