XML PROCESSING. NO PARSING

Yevgeniy Guseynov
Optimal Solutions & Technologies, 2001 M Street, NW, Suite 3000, Washington, DC 20036, U.S.A.

Keywords: XML, Data Exchange, XML Parser, XML Processing Efficiency, DOM, SAX.

Abstract: The main properties considered lacking from XML for a potentially efficient interchange format are Compactness and Processing Efficiency, and Parsing being the main deterrent to Processing Efficiency. The proposed Contiguous Memory Tree (CMT) and its XML API completely resolve Parsing and Processing Efficiency permitting an efficient interchange format for XML. CMT is based on the presentation of XML documents as a tree that contiguously resides in memory and is simultaneously a stream that can be directly copied as a message and an application object that can be directly accessed through the CMT XML API. CMT XML API does not need to read and evaluate markup or decode information items that takes much CPU time when processing, thus is significantly more efficient than any existing formatting schemes, SAX and DOM parsers.

1 INTRODUCTION

Many modern technologies in distributed computing and Web Services are powered by XML (Cauddwell, et al., 2001). Extensible Markup Language, abbreviated XML, was defined in the XML 1.0 Specification (Bray, et al., 2006) published by the Worldwide Web Consortium (W3C). XML documents are made up of a sequence of characters with the textual encoding UTF-8, UTF-16 or others for storage and interchange data. Thus in memory an XML document is usually formatted as a stream of characters (bytes), some of which form character data, and some of which form the markup. The computer must interpret this stream as application objects in order to access the content of the document. In some applications like insurance, banking, and financial businesses, the advantages of XML usage are tempered by inefficiencies that stem from the textual encoding (White, et al., 2007, Matthiaas and Jasmi, 2003). The main properties that are considered lacking in XML for a potential efficient interchange format are Compactness and Processing Efficiency (Schneider, et al., 2007). These shortcomings have led to the development of alternative encoding formats for example (Schneider, et al., 2007, Conner, 2003, Sandoz, et al., 2004). This paper concentrates on developing the format for resolving the Processing Efficiency of XML.

While the main rules for constructing XML documents are relatively simple a document itself may have a very complex hierarchical (tree) structure. In order to read XML documents as a stream of characters, almost all applications rely on an XML parser that also provides an API to receive or request information from the documents. There are two major models for processing XML documents: the Simple API for XML (SAX) parser (Megginson, 2004) and the Document Object Model (DOM) parser (Le Hégaret, et al., 2005). Parsing - the step where components of an XML document are transformed (read) from a stream of text data into application objects - is the main part of the Processing Efficiency property. For a SAX parser it is the creation of events for callbacks; for DOM it is the creation of a tree in memory that is compatible with the XML document at the XML Information Set level (Cowan, et al., 2004). Regardless of the parser type, SAX or DOM, if an application needs to maintain an XML document or its part in memory for extended processing it uses a DOM tree structure that is based on Composite Pattern (Gamma, et al., 1994) where each component of an XML document or node is created as a separate application object and linked to its parent and sibling nodes. By design the DOM tree structure is spread in memory so, in order to store, the application must rewrite it back to a stream of text data. Inevitability having these two substantially different instances of an XML document, a stream of text data for storage and

Guseynov Y.
XML PROCESSING. NO PARSING.
DOI: 10.5220/0001819500810084
In *Proceedings of the Fifth International Conference on Web Information Systems and Technologies (WEBIST 2009)*, page
ISBN: 978-989-8111-81-4

exchange, and application objects in memory in order to access information, a situation that occurs in all existing applications, causes inefficiency in processing XML documents.

The proposed Contiguous Memory Tree (CMT) and its XML API (Guseynov, 2006) completely resolve Parsing and Processing Efficiency by creating an efficient interchange format for XML. It is based on the presentation of XML documents as a tree structure that contiguously resides in memory and is simultaneously a stream that can be directly copied as a message and an application object that can be directly accessed through the CMT XML API. CMT is the universal way to exchange XML documents and any hierarchical information regardless of operating systems and languages like C++ with direct access to memory or Java, Visual Basic, Perl, others with the ability to contiguously allocate arrays in memory. CMT and its XML API has all features of existing formats: the compatibility with the XML document at the XML Information Set level, serialization, parsing as DOM and SAX, XML schema independence and self-description, support a sequential or fragments processing (Streamability), indexing of repeated strings, preservation of the state for documents with the same schema and vocabulary, platform and language neutrality, reduced document size, and fast processing speed. In addition, CMT and its XML API have significant advantage. CMT XML API does not need to read and evaluate markup or decode information items that takes much CPU time when processing, thus is significantly more efficient than any known parser by the elapsed time that a parser needs to parse an input stream before actual processing.

2 CONTIGUOUS MEMORY TREE

We may define CMT based on pointers for languages like C++ with direct access and explicit allocation of memory and like Java based on the ability to contiguously allocate arrays in memory. The approach based on arrays is universal because almost all programming languages have the ability to contiguously allocate memory arrays of the basic types, integer and character.

To build CMT for an XML document or any hierarchical information we need three arrays: the array of integers, Hierarchy[], to hold the hierarchical (tree) structure of the document; the array of characters, SchemaComponents[], to hold tag names, attribute names, and other components for all documents with the same XML schema; the array of characters, DocumentValues[], for each document to hold elements and attributes values for the whole document.

The Hierarchy[] array is built with blocks of six integers:

```
SchemaNode
{     int   parent;
  int   firstChild;
    intnextSibling;
  int   tagName;
  int   offset;
  int   numOfAttributes;
}
```

The first four integers allow CMT to be built from any hierarchical information. For each element E1 from the hierarchy of an XML document, the parent, firstChild, and nextSibling members of SchemaNode are positions in the Hierarchy[] array that are start positions, respectively for parent, first child, and next sibling elements for E1. The member tagName is the start position in the SchemaComponents[] array for the element name. The last two integers in the struct SchemaNode pertains to XML documents. The member offset is the start position in DocumentValues[] for the content or value that is the text between two tags in an XML element. numOfAttributes represents the number of attributes in an XML element.

Each CMT element in memory consists of a SchemaNode followed by numOfAttributes pair of integers: the first element of a pair is the start position of the attribute name in the SchemaComponents[] array and the second is the start position of the attribute value in the DocumentValues[] array. Next three tables present an example of CMT for an XML document

```
<Product bottles="12" size="9oz" >
      <ItemName>Chartreuse verte</ItemName>
      <ItemPrice>$18.00</ItemPrice>
</ Product >
```

To define CMT we need to build each element in the XML document one by one into Hierarchy[], SchemaComponents[], and DocumentValues[] arrays. They will remain unchanged if we copy them to any location and we may also directly store these arrays contiguously on the disk or any other medium as a stream of bytes to exchange the XML document with other applications. After copying these three arrays back into memory an application can access without parsing all the information that the XML document has, starting from any position in the Hierarchy[] array.

Table 1: Memory layout for CMT array Hierarchy[23].

0 NULL	1 Parent	2 firstChild	3 NextSibling	4 tagName	5 offset	6 numOfAttributes
0	0	11	0	1	0	2
7 Attribute Name	8 Attribute Value	9 Attribute Name	10 Attribute Value	11 Parent	12 firstChild	13 nextSibling
9	1	17	4	1	0	17
14 tagName	15 Offset	16 numOfAttributes	17 Parent	18 firstChild	19 nextSibling	
22	8	0	1	0	0	
20 TagName	21 offset	22 numOfAttributes				
31	25	0				

Table 2: Memory layout for CMT array SchemaComponents[41].

	1							9								17				
\0	P	r	o	d	u	c	t	\0	b	o	t	T	l	e	s	\0	s	i	z	e

	22									31									
\0	I	t	e	m	N	a	m	e	\0	I	t	e	m	P	r	i	c	e	\0

Table 3: Memory layout for CMT array DocumentValues[32]].

	1			4				8											
\0	1	2	\0	9	o	z	\0	C	h	a	r	T	r	e	u	s	e		V

		25						
e	\0	$	1	8	.	0	0	\0

There are two more important features of CMT: direct access to the XML schema hierarchical information without navigating the tree and the ability to search for an item in CMT based on its starting position (integer) in the array. After CMT or Hierarchy[], SchemaComponents[], and DocumentValues[] arrays are built they remain unchanged and we may write out starting positions for their elements as constants and make them available for applications for direct access or fast search. Applications may use these constants even when CMT is stored or exchanged for direct access to all information in an XML document without navigating its CMT structure.

Defined by three arrays, CMT is compatible with XML at the XML Information Set level (Cowan, et al., 2004) but not as readable for human eyes, as the original XML format is, to satisfy the XML design goal: "XML documents should be human-legible and reasonably clear" (Bray, et al., 2006). Meanwhile, the integer and character arrays that define the CMT are human readable and self descriptive. In addition, when reading a CMT, humans do not need to read and evaluate markup or decode information items that all known formatting schemes have.

3 PERFORMANCE

CMT eliminates the parsing step in XML processing and it is evident that the usage of the CMT XML API is more efficient than any existing XML parser: both of them need to read documents from the file (stream); after reading, the CMT object is ready for use but the XML document in the textual encoding still needs to be parsed before it will be available for access by an application. The bigger the XML documents are the more efficient the CMT XML API is against any Parser. Direct access to all information in CMT also greatly increases Processing Efficiency.

To evaluate CMT performance we use a simple XML Document Customer-Sales.xml similar to (Conner, 2003) to build different XML documents with sizes between 1 kilobyte and 1 megabyte. Xerces C++ SAX and DOM Parsers version 2.8.0 (Xerces, 2007) were chosen for the base measurement. For comparison the CMT Document Customer-Sales.cmt was built from Customer-Sales.xml and processed by the CMT Parser. All timing runs are for 300, 600, 1200 iterations after a warm up of 500 iterations on a Pentium(R) 4 CPU 2.66GHz, 512 MB of RAM.

The SAX Parser test is based on the MemParse project from the Xerces-C++ package. It reports all SAX events and outputs the number of elements and attributes from Customer-Sales.xml. The same functionality SAX CMT Parser processes the Customer-Sales.cmt document that is compatible with the initial XML document at the XML Information Set level. The DOM Parser test is based on the DOMCount project from the Xerces-C++ package that builds DOM for Customer-Sales.xml. The DOM CMT Parser processes the Customer-Sales.cmt document. The table below presents the results for 3 kilobytes XML documents. For other sizes the comparison is similar.

Table 4: Simple Performance Test.

Iterations	SAX CMT (ms)	SAX Xerces (ms)	DOM CMT (ms)	DOM Xerces (ms)
1200	40	2070	40	5340
600	20	1030	20	2740
300	10	510	10	1400

These results show a significant advantage when using the CMT XML API against the Xerces SAX and DOM Parsers. They also demonstrate that CMT SAX and CMT DOM parsers are equivalently fast which is expectable based on the definition of CMT. Similar comparison in Efficient XML (Schneider, et al., 2007, White, et al., 2007), Fast Infoset (Sandoz, et al., 2004), and CBXML (Conner, 2003) against Xerces parser show 2 to 3 times improvement.

The Demo and Sample Project CMT XML API are available from the author upon request via email. The Sample Project shows how to use the CMT XML API to manipulate XML documents: build CMT objects, store and retrieve CMT from a file, navigate CMT as a tree, update CMT - set values for attributes and elements, and delete and add nodes. You may try it on your own XML data and a favoured parser to compare with the provided experiments.

REFERENCES

Bray, T. et al. Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/xml/, September 2006.

Cauddwell, P. et al. Professional XML Web Services. Wrox Press Ltd., 2001.

Conner, M. CBXML: Experience with Binary XML, IBM Corporation, http://www.w3.org/2003/08/binary-interchange-workshop/19-IBM-CBXML-W3C-Submission-updated.zip, 2003

Cowan, J. et al. XML Information Set (Second Edition), W3C Recommendation, http://www.w3.org/TR/xml-infoset/, February 2004.

Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994

Guseynov, Y. U.S. Patent Application (pending) No. 11/603,299 for Contiguous Memory Tree, Filing Date: November 21, 2006.

Le Hégaret, P. et al. Document Object Model (DOM), http://www.w3.org/DOM/, January 2005.

Matthiaas, N., Jasmi, J. XML Parsing: A Threat to Database Performance. CIKM'03 November 3 – 8, 2003, New Orleans, Louisiana, USA. http:// lists.w3.org/Archives/Public/www-ws/2004Oct/att-0032/MNicola_CIKM_2003_1_.pdf.

Megginson, D. Simple API for XML (SAX), http://www.saxproject.org/, April 2004.

Sandoz, P. et al. Fast Infoset, http://java.sun.com/develeper/technicalArticles/xml/fastinfoset/, 2004.

Schneider, J. et al. Efficient XML Interchange (EXI) Format 1.0, W3C Working Draft, http://www.w3.org/ TR/exi/, July 2007.

White, G. et al. Efficient XML Interchange Measurements Note, W3C Working Draft, http://www.w3.org/ TR/exi-measurements, July 2007.

Xerces-C++ Parser. http://xerces.apache.org/xerces-c/, 2007.