An Efficient XML Parser Generator Using Compiler Compiler Technique

KAZUAKI MAEDA
Department of Business Administration
and Information Science, Chubu University
1200 Matsumoto, Kasugai, Aichi 487-8501, JAPAN

Abstract: - This paper describes design issues and experiment results of an efficient XML parser generator, Xsong. A traditional compiler construction technique is applied to Xsong so that it realizes both expressiveness and efficiency for parsing XML documents. To compare with the performance of DOM based programs, SAX based programs and a program generated by Xsong, experiments were designed. The experiments showed that the program generated by Xsong is faster than the DOM based programs. Moreover, in regard to memory usages, it is as efficient as the SAX based programs.

Key-Words: - XML, DOM, SAX, Compiler Compiler, Parser Generator, C++, Java, C#

1 Introduction

Due to the growth of the computing power and the proliferation of the Internet, XML (Extensible Markup Language) becomes very popular to represent data in many application fields. XML is designed as a text-based, human-readable, and self-describing language. In addition, XML is a markup language derived from SGML (Standard Generalized Markup Language) so that it can control the format and the presentation of documents.

XML enables data exchange between different platforms (computers, operating systems, and programming languages) using the characteristics of platform independence which XML has. For example, we can exchange data between applications via the Internet, or we can extract data from a database and reuse it with other applications.

The orientation of XML documents is generally one of two types: document-centric and data-centric[1, 2]. The target of the document-centric XML documents is for visual consumption so that the documents have less structured characteristics. Books, articles, and E-mails are typical examples for the document-centric. XHTML[3] is a language to describe web pages as the document-centric XML documents.

In contrast to this, the data-centric XML documents tend to include very granular collections of data so that it is applied to computer processing and databases storage. For example, bibliography data and order forms are typical examples for the data-centric. The data exchanged with Web services[4] is mostly the data-centric XML document.

XML has gained the prominence within the technology community in a short time. XML documents, however, take up lots of space to represent data that could be similarly modeled using a binary-format or a simple text file format because the XML documents are human-readable, platform-neutral, meta data-enhanced, structured code. It can be from 3 to 20 times as large as a comparable binary or alternate text file representation[5]. In the worst case, it's possible that 1G bytes of database information could expand to over 20G bytes of XML encoded data.

This paper describes a XML parser generator, Xsong, which was developed for the data-centric XML documents. Xsong is both easy to describe user defined functions and useful to generate efficient codes to parse XML documents[1].

Xsong was designed from experiences to develop a commercial software development tool. The tool supports client/server application development on a commercial database management system by Oracle. It stores all the information in XML documents and generates GUI based Java applications. At the design phase of the tool, it expected that the size of XML documents stored by the tool is more than a few megabytes. Therefore, an efficient XML parser was desperately needed. DOM and SAX were not satisfy with the needs because DOM has the poor performance and SAX has only a few functionality. As a result of this, Xsong was developed.

[1] Xsong is available for the document-centered XML documents. It, however, does not make use of the efficiency because the document size is comparatively small.

Xsong has the following characteristics.

- The generated program is efficient.

  Xsong reads a schema definition of XML documents and user defined programs, and generates grammar rules for Antlr (ANother Tool for Language Recognition) [6]. Antlr is a parser generator which generates a recursive descendent parser written in a specified programming language. The generated XML parser using Xsong and Antlr reads XML documents, and checks their validity in comparison with the schema definition. Thanks to the compiler technology, the generated XML parser is as efficient as a parser using SAX.

- User defined functions are separated from the schema definition.

  The position of the schema definition is specified using XPath and user defined functions are embedded into the XML parser according to the specified position. As a result, the schema definition and the user defined functions are clearly separated.

  If the user defined functions are embedded and merged into the schema definition, it is hard for human to read and maintain it. Therefore, the separation of the schema definition and user defined functions is very important.

- More than one programming languages are supported.

  The schema definition does not depend on a specific programming language, but user defined functions are written in one of favourite programming languages. If a user changes from the programming language to another one, all the user has to do is to rewritten only the user defined functions. Currently, three programming languages (Java, C++, and C#) are supported.

This paper describes the design issues of Xsong and experiments to check the performance. In section 2, the current major XML parsers will be briefly discussed. In section 3, the overview of Xsong, the input and output file will be explained. Moreover, experiments to compare the performance of DOM based programs, SAX based programs, and a program generated by Xsong are described. Finally, the paper will be summarised.

2 Background

2.1 XML

The key rules of XML syntax are based on an element and an attribute. For example, the simplified research paper information in proceedings is defined using DBLP Bibliography[7] shown in Figure 1 [2].

The element defines structural parts of a document by wrapping and labeling. In Figure 1, an author is defined by wrapping it in a start tag and an end tag labeled "author." The attribute is a name-value pair that qualifies an element. In Figure 1, an attribute, key, is defined using the name "key" and the value "conf/robocup/MaedaKT98."

```
<?xml version="1.0"?>
<dblp>
  <inproc key="conf/robocup/MaedaKT98">
    <author>Kazuaki Maeda</author>
    <title>Ball-Receiving Skill Dependent on
      Centering in Soccer Simulation Games
    </title>
    <pages>152-161</pages>
    <year>1998</year>
    <booktitle>RoboCup</booktitle>
  </inproc>
</dblp>
```

Figure 1: An example of DBLP

Schema definition languages are used to specify XML documents. There are some schema definition languages, those are DTD, XML Schema[8], and RELAX NG[9, 10]. In Xsong, RELAX NG is used because of the simple and powerful language specification. Figure 2 shows an example of RELAX NG, that is the specification of the bibliography in Figure 1.

In the Figure 2, the "dblp" element declaration specifies the child element "inproc." Moreover, the "inproc" element declaration specifies some child elements, which are "author," "title," "booktitle," "pages," or "year." To specify text data, such as a name of an author, we can use <text/> in those element declarations. The "inproc" element also has an attribute "key."

2.2 DOM

Most popular approach for XML data processing is a tree-based manipulation. To process XML documents, firstly, they are read and parsed to a hierarchical tree of elements and other XML entities in a main memory. After construction of the tree, each node can be accessed using tree traversal APIs. For the standard tree access, the Document Object Model (DOM) is defined by W3C[11].

DOM provides a language independent definition to access and modify XML documents. The DOM APIs deal with the generic structural components of XML documents. For example, there are many APIs including

[2]This example is a part of the XML document used for experiments in section 4.

```
<?xml version="1.0"?>
<grammar xmlns=
    "http://relaxng.org/ns/structure/1.0">
  <define name="dblp">
    <element name="dblp">
      <zeroOrMore>
        <element name="inproc">
          <attribute name="key"/>
          <zeroOrMore>
            <choice>
              <element name="author">
                <text/>
              </element>
              <element name="title">
                <text/>
              </element>
              <element name="booktitle">
                <text/>
              </element>
              <element name="pages">
                <text/>
              </element>
              <element name="year">
                <text/>
              </element>
            </choice>
          </zeroOrMore>
        </element>
      </zeroOrMore>
    </element>
  </define>
</grammar>
```

Figure 2: RELAX NG Schema Definition for DBLP bibliography

- appendChild(): to add a node to the end of the list of children for a specified node,
- getFirstChild(): to get the first child of this node,
- getNextSibling(): to get the node immediately following this node, and
- setAttribute(): to set the value of an attribute for the element.

We can develop programs to read XML data, modify them, add nodes to them, and delete nodes from them using DOM implementations (ex. Xerces-C++[12] and Xerces-J[13]) provided by open source organizations or companies. Figure 3 is an example of a C++ program to count the number of elements. It is a fragment of a test program for the performance evaluation described in section 4.

DOM has a drawback that entire XML documents must be loaded in the main memory before manipulating them. The tree-based approach is useful for programmers to manipulate XML documents according to the hierarchical tree structure. The programs walk through the structure of the

```
int countChildElements(DOMNode* n) {
  DOMNode* cn;
  int count = 0;
  if(n){
    if(n->getNodeType() ==
                DOMNode::ELEMENT_NODE)
      count++;
    for(cn=n->getFirstChild(); cn != 0;
                cn=cn->getNextSibling())
      count += countChildElements(cn);
  }
  return(count);
}
```

Figure 3: A fragment of C++ programs using DOM

XML documents in order to access them so that entire XML documents must be loaded in the main memory before the manipulation. When a program reads large XML documents to use DOM APIs, it puts a great strain on system resources such as memory and CPU. Moreover, if we need to convert the XML documents from a DOM representation to a program specific data structure, memory shortages are made worse.

This results in that DOM provides much expressiveness for processing XML documents, but it consumes a lot of system resources.

2.3 SAX

As an alternative approach, Simple API for XML (SAX) has been designed[14]. It provides an event-based processing. Instead of constructing an internal tree, SAX sends parsing events for basic XML contents, for example, start of an element, end of an element, and so on. The events are sent to application handlers in exactly the order they are found in the XML documents. XML documents can be processed incrementally so that they can discard information if it is not needed. To deal with the different events, programmers can construct their own data structures using event handlers.

Figure 4 is an example of a C++ program to count the number of elements using SAX. Class SAXCountHandler overrides the method startElement in class DefaultHandler, and it increments the variable elementCount by one.

The SAX parser can be fast with small memory usage. It provides a lower-level access so that it puts no strain on system resources even if the size of XML documents is large. It, however, has a drawback that it is difficult for programmers to manage the structure using only parsing events if the structure of XML documents is complex.

This results in that SAX provides good efficiency for parsing XML documents, but it needs many lines of codes for structure-based processing.

```
class SAXCountHandler: DefaultHandler {
  ........
  public: void startElement(
    const XMLCh* const uri,
    const XMLCh* const localname,
    const XMLCh* const qname,
    const Attributes& attrs) {
      elementCount++;
    }
  private: int  elementCount;
};
```

Figure 4: An example of C++ programs using SAX

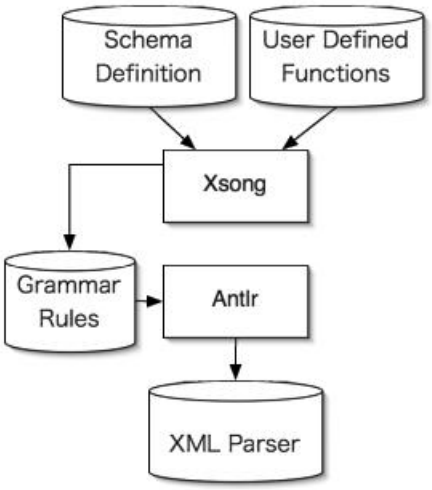

Figure 5: Data flow for Development of a XML parser using Xsong

3 Design of Xsong

This section describes a XML parser generator, Xsong, which supports both expressiveness and efficiency for parsing XML documents.

3.1 Outline of Xsong

As depicted in Figure 5, Xsong reads two files, a schema definition file for target XML documents, and a user defined function file to specify actions for elements and attributes. It generates a grammar rule file including the user defined functions. The generated file is read by Antlr and Antlr generates an XML parser program written in a specified programming language. The generated program not only checks the grammatical correctness, but also invokes the user defined functions.

Antlr is one of traditional parser generators. It generates recursive descent parsers from LL(k) grammars ($k > 1$) in Extended Backus-Naur Form notation. It allows each grammar rule to have parameters and return values, facilitating attribute passing during parsing. Antlr converts each rule to a function (or a method) in a recursive descent parser. The generated program parses input XML documents in according with the grammar for the XML documents. Thanks to the compiler technology, it is possible to parse it efficiently.

3.2 Input of Xsong

An input of Xsong is a schema definition file written in RELAX NG. The reason why RELAX NG was chosen is that it enables simple description to define the schema of XML documents. For example, the schema definition of DBLP bibliography has already described in Figure 2.

We can describe actions to elements and attributes at the user defined function. In the user defined functions, there are some rules to specify the functions. A rule is composed of three parts, those are a keyword, an XPath expression, and a fragment of programs. Using the XPath expression, the position of the schema definition is specified and the fragment of programs is embedded into the specified position.

For the keyword, either "startOf" or "endOf" is specified in consideration of the following;

- If "startOf" is specified and the position in XPath is an element, the program is invoked just after the specified start tag is read.
- If "endOf" is specified and the position in XPath is an element, the program is invoked just after the specified end tag is read.
- If "startOf" is specified and the position in XPath is not an element, the program is invoked just before the specified data is read.
- If "endOf" is specified and the position in XPath is not an element, the program is invoked just after the specified data is read.

Figure 6 is an example of the rule for specifying to increase the value of the variable elemCount by one.

```
startOf  //element    { elemCount++; }
```

Figure 6: An example of user defined rules

Figure 7 is another example for specifying to print all text contents. In the rule, $$ is a special variable for a text content.

```
endOf  //text    { printf("%s",$$); }
```

Figure 7: Another example of user defined rules

Figure 8 is a more complex example for specifying to print data in the form "author=....." when an element "author" is read. The XPath expressions describe that a value of an attribute "name" in an element is "author."

```
startOf  //element[@name="author"]
    { printf("author="); }
endOf    //element[@name="author"]/text
    { printf("%s",$$); }
endOf    //element[@name="author"]
    { printf("\n"); }
```

Figure 8: An example to print author names

3.3 Grammar Rules as Output

Figure 9 describes a fragment of grammar rules for Antlr. It is generated by Xsong from the schema definition (Figure 2) and the user defined functions (Figure 8). In the grammar rules,

- inproc_element is a rule to analyze the element "inproc" and the child elements with an action for increasing the variable elemCount by one.
- inproc_attr is a rule to analyze the attribute "key."
- inproc_body is a rule to analyze contents between the start tag and the end tag.
- inproc_content is a rule to define the element "inproc" has zero or many elements of "author," "title," "booktitle," "pages," or "year," or character symbols.

```
inproc_element :
    BGN_inproc (inproc_attr)* inproc_body
    { elemCount++; }
    ;
inproc_attr :
    Attr_key   EQ attrValue
    ;
inproc_body :
    CLS inproc_content END_inproc
    ;
inproc_content :
    ( author_element | title_element
    | booktitle_element | pages_element
    | year_element | CHAR )*
    ;
```

Figure 9: An example of grammar rules for Antlr

After Xsong generates the grammar rules, Antlr reads the rules and generates a XML parser program in a specified programming language (C++, Java, and C# are currently supported).

4 Experiments for Performance Comparison

To check the performance improvement, experiments were designed with the following conditions:

Computer: Compaq Evo N200 with Mobile Pentium III 700MHz and 192M bytes of memory

OS: Red Hat Linux 9 (kernel 2.4.20), Windows 2000 SP4

Programming language: C++ (gcc 3.2.2), Java (1.4.2_03), C# (.NET and Mono 0.30.1)

XML parser: Apache Xerces-C++ Version 2.2.0[12], Apache Xerces-J Version 2.4.0[13]

Ten test data+ Ten XML documents with various sizes
1M bytes, 2M bytes, 3M bytes, 4M bytes,
5M bytes, 6M bytes, 7M bytes, 8M bytes,
9M bytes, and 10M bytes

The test data were extracted with appropriate sizes from DBLP Bibliography[7] (more than 130M bytes in total). Figure 1 is a fragment of the XML test data, and Figure 2 is a fragment of the RELAX NG to define the XML documents.

Seven test programs They count the number of elements in the XML documents,

- using DOM with Xerces-C++, is written in C++ and executed on Linux,
- using DOM with Xerces-J, is written in Java and executed on Linux,
- using SAX with Xerces-C++, is written in C++ and executed on Linux,
- using SAX with Xerces-J, is written in Java and executed on Linux,
- using class XmlDocument, is written in C# and executed under Mono on Linux,
- using class XmlDocument, is written in C# and executed under .NET on Windows 2000, and
- using Xsong, is generated by Xsong, written in C++ and executed on Linux.

These programs were executed ten times, and check the memory usage and the execution time.

The results of the experiment, from the viewpoint of memory usage, are shown in Figure 10. To check the details, Figure 11 depicts three test programs with the least memory usage.

The figures show that the program in C++ using DOM consumes quite a lot of memory. Surprisingly, the program consumes about 140M bytes of memory when it parses 4M bytes of XML documents. When the DOM based program parses more than 7M bytes of the XML documents, it aborted due to lack of memory.

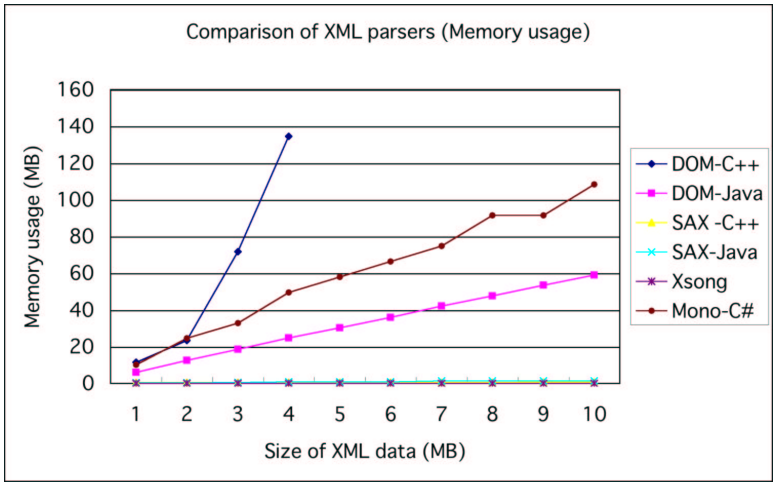

Figure 10: Experiment results (memory usage)

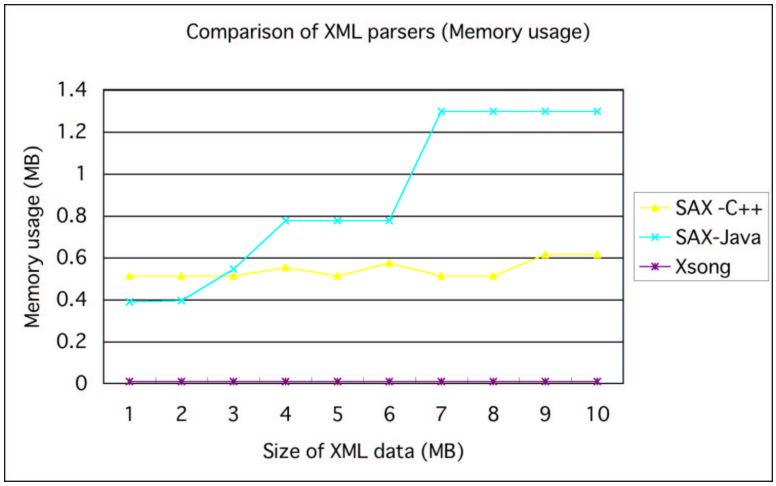

Figure 11: Experiment results of top three (memory usage)

The figures also show that the programs in both C++ and Java require less memory usage. The program generated by Xsong can be executed under less memory than SAX based programs.

The results of the experiment from the viewpoint of execution time, are shown in Figure 12. Figure 13 depicts top three test programs with the least execution time usage to check the details.

The figures show that the programs using DOM take much time in comparison with the programs using SAX. In the case of the program in C++ using DOM, the performance drastically decreased when the size of the XML document was more than 4M bytes. In comparison with Figure 10, the program might exhaust the physical memory.

The figures show that the program using Xsong can be executed in equivalent execution time to SAX based programs

These results show the good performance of Xsong from the point of view of both memory usage and execution time in comparison with DOM and SAX.

5 Conclusion

This paper described an efficient XML parser generator Xsong and experiment results to check the performance. Xsong realizes both expressiveness and efficiency for parsing XML documents. The experiment results showed the good performance from the point view of memory usage and execution time.

References

[1] Akmal B. Chaudhri, Awais Rashid, and Roberto Zicari ed., XML Data Management, Addison Wesley (2003).

[2] Ronald Bourret, XML and Databases, http://www.rpbourret.com/xml/XMLAndDatabases.htm.

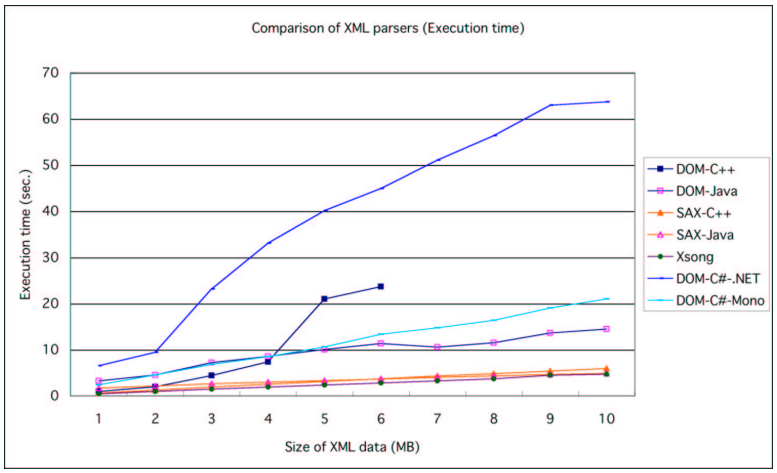

Figure 12: Experiment results (execution time)

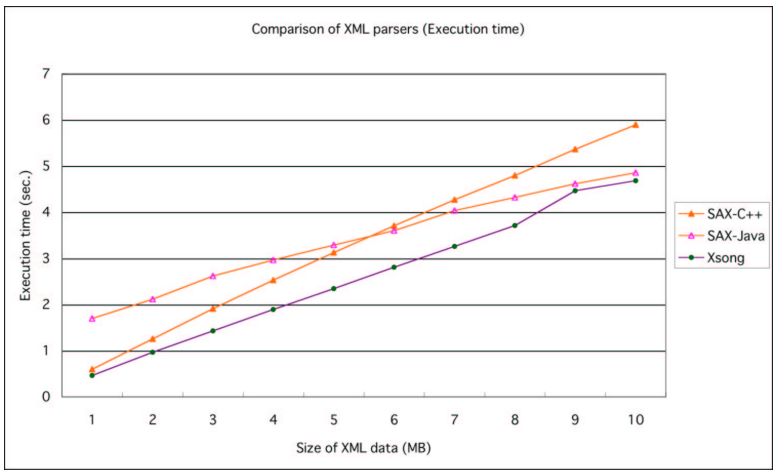

Figure 13: Experiment results of top three (execution time)

[3] W3C, XHTML 1.0 The Extensible HyperText Markup Language, http://www.w3.org/TR/xhtml1/.

[4] W3C, Web Services, http://www.w3.org/2002/ws/.

[5] Zap Think, The "Pros and Cons" of XML, Zap Think Research Report (2001).

[6] ANTLR Parser Generator Translator Generator Home Page, http://www.antlr.org/.

[7] DBLP Bibliography, http://www.informatik.uni-trier.de/?ley/db/.

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

[9] OASIS, RELAX NG Specification, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[10] James Clark, RELAX NG Home page, http://www.relaxng.org/.

[11] W3C DOM Working Group, Document Object Model (DOM), http://www.w3.org/DOM/.

[12] The Apache Foundation, Xerces C++ Parser, http://xml.apache.org/xerces-c/.

[13] The Apache Foundation, Xerces2 Java Parser Readme, http://xml.apache.org/xerces2-j/

[14] SAX,http://www.saxproject.org/.

[15] Mono, http://www.go-mono.org/.