VALIDATION OF XML DOCUMENT BASED ON PARALLEL BIT STREAM TECHNOLOGY

by

Shiyang Yang

B.Sc.,Beihang University, 2009

a Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2013

APPROVAL

Name: Shiyang Yang

Degree: Master of Science

Title of Thesis: Validation of XML Document Based on Parallel Bit Stream Technology

Examining Committee: Dr. William Sumner
Chair

Dr. Rob Cameron, Senior Supervisor

Dr. Thomas Shermer, Second Supervisor

Dr. Fred Popowich, Examiner

Dec 23, 2013

Date Approved: _____

Partial Copyright Licence

Abstract

The validating of XML files is a main component of XML file processing. This thesis investigates single-instruction multiple-data(SIMD) and parallel bit stream technologies in high performance XML validation. The content model and datatypes of the schema are translated into regular expressions and then into parallel bitwise operations. The element content and data of the instance file are extracted to form byte streams, and then transformed into parallel bit streams. Finally, the parallel bitwise operations are applied on corresponding bit streams to validate the content model or datatype. This method is then studied by changing the characteristic of the instance files, such as the proportion of content data, occurrences of elements. Comparisons of the performance are also made with Xerces, the well known XML parser with validator. Whereas the parallel bit stream validation algorithm requires less than 20 cycles per byte, while Xerces requires 40 to 300 cycles per byte.

Acknowledgments

I would like to thank all the people who has helped and supported me during my graduate study.

My first and sincere appreciation goes to my senior supervisor Dr. Robert D. Cameron, for all I have learned from him and for his continuous help in all stages of this thesis. I would also like to thank him for encouraging me and providing valuable insights. I could not have completed this thesis without him.

I would also like to thank my supervisor Dr. Thomas C. Shermer, my thesis examiner Dr. Fred Popowich and Dr. William Sumner for being in my committee and reviewing this thesis.

I would like to express my deep gratitude to all of my colleagues in Dr. Cameron's lab for their help and support. They are Dan Lin, Nigel Medforth, Ken Herdy, Vera Lukman, Hua Huang, Qiang Zhang and Rui Yang. I specially thank Nigel Medforth for his help and suggestions during my research.

Last but not least, I would like to thank my family for their un questioning love and encouragement. This thesis is dedicated to them.

Contents

Chapter 1

Introduction

The emergence of XML as a standard representation format for data on the Web has led to a rapid increase of databases that store, query, and update XML data. XML validation is the process of checking whether an XML document is both well-formed and also valid to follow a defined structure. An XML document must respects the rules dictated by a particular Document Type Definition (DTDs)[1] or XML schema[6] or more recently, RELAX NG schema.

An XML document can be processed abstractly as a tree of nested elements. Checking that a word satisfies a regular expression[5] is the start of checking whether an XML document satisfies a DTD.

Single-instruction-stream, multiple-data-stream (SIMD) is particularly applicable to common tasks of graphics, audio, video and other applications. It is also proven useful in application of high-performance text processing. SIMD is a parallel processing technology which is capable of deploying the same instruction on multiple data simultaneously.

Nowadays, processor manufacturers including Intel, AMD, ARM and IBM have embraced their instruction set architectures with SIMD extensions to improve the performance of process.

Instead of a byte-at-a-time text processing method, SIMD based text processing method is based on the concept of parallel bit streams. The idea is to transpose byte oriented character stream data into eight parallel bit streams, each of which comprises one bit of each byte. In the validation of an XML document, there are several components which are able to be converted into a byte stream processing issue. The content model validation and the datatype validation are of those components.

This thesis proposes an algorithm to convert the sequence of elements into byte streams, of which, each stream consists of the IDs of each child-elements of a corresponding element. Therefore, the problem of the content model validation is converted into a byte stream processing problem, and we can apply the parallel bit stream technology to achieve a significant speed-up.

However, the problem of datatype validation can also be converted into a byte stream processing problem. By abstracting the content of each element, and streaming them into corresponding byte streams, the byte streams of different types are created and available to be processed using parallel bit stream technology.

To evaluate the performance of the algorithm, tests are run and comparisons are made with the traditional byte-at-a-time validator. We apply both the parallel bit stream technology algorithm and the traditional validator on several different test cases.

The rest of this thesis is organized as follows. Chapter 2 introduces the background of XML parsing and validation, SIMD technology, and parallel bit stream technology. Chapter 3 presents the algorithm of abstraction of content model and the validation against grammars. Chapter 4 describes the algorithm of datatype validation base on parallel bit stream technology. Chapter 5 shows performance comparisons of the parallel bit stream technology base validation algorithm and the traditional validator. In Chapter 6, the results and future work are discussed.

Chapter 2

Background

2.1 Regular Expression and XML parsing

Regular expressions provide a concise and flexible means to specify and recognize strings of text. Most formalisms provide boolean "or", grouping and quantification operations to construct regular expressions. For "or" operation, a vertical bar separates alternatives. Parentheses are used define the scope and precedence of the operators in grouping. The most common quantifiers are the question mark, asterisk and plus sign. A question mark indicates there is zero or one of the preceding element. The asterisk indicates the preceding element can occur zero or more times, while the plus sign indicates there is one or more of the preceding element.

In XML regular expressions, a piece is an atom, possibly followed by a quantifier. A branch consists of zero or more pieces and a regular expression is composed from zero or more branches, separated by vertical bars.

There are two types of characters that are used in regular expressions: metacharacters and normal characters. A metacharacter has special meaning in regular expressions, but can be escaped to be used as normal character. Metacharacter includes "\","?","*" and so on.

XML DOM(Document Object Model) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of an XML document. According to the DOM, everything in an XML document is a node. It views an XML document as a tree-structure, which is called a node-tree. The nodes in the node tree have a hierarchical relationship to each other. Relationships are described

as parent, child and sibling.

SAX(Simple API for XML) is an event-based sequential access parser API for XML documents. It provides a mechanism for reading data from an XML document that is an alternative to DOM. Where the DOM operates on the document as a whole, SAX parsers operate on each piece of the XML document sequentially. Compared with DOM, SAX has its benefits such as minimum memory requirement and speed. And drawbacks, such as in XML validation, it requires access to the full document.

One approach of XML paring is called pull parsing. It treats the document as a series of items which are read in sequence using the iterator design pattern. A pull parser creates an iterator that sequentially visits the various elements, attributes, and data in an XML document. Code that uses this iterator can test the current item (to tell, for example, whether it is a start or end element, or text), and inspect its attributes (local name, namespace, values of XML attributes, value of text, etc.), and can also move the iterator to the next item. The code can thus extract information from the document as it traverses it. The recursive-descent approach tends to lend itself to keeping data as typed local variables in the code doing the parsing, while SAX, for instance, typically requires a parser to manually maintain intermediate data within a stack of elements which are parent elements of the element being parsed. Pull-parsing code can be more straightforward to understand and maintain than SAX parsing code. Examples of pull parsers include StAX in the Java programming language, XMLReader in PHP and System.Xml.XmlReader in the .NET Framework.

2.2 Basic of XML Validation

XML validation is the process of checking a document written in XML to confirm that it is both well formed and also valid in that it follows a defined structure. A valid document respects the rules dictated by a particular DTD or XML schema document. The requirements of the schema include such constraints as:

- Elements and attributes may be included, and their structure
- The structure as specified by a regular expression syntax
- How character data is to be interpreted

2.2.1 Document Type Definitions

DTDs(Document Type Definitions) define the legal building blocks of an XML document, which include elements, attributes, entities, PCDATA, CDATA, etc. A DTD uses a different syntax from XML. DTD are limited when it comes to the actual types of data they can include. A valid XML document contains only elements that are defined in the DTD. There are also limitations on ordering child elements. Here is an example DTD.

```
<!DOCTYPE RECIPE_COLLECTIONS [

<!ELEMENT collection (description,recipe*)>

<!ELEMENT description ANY>

<!ELEMENT recipe (title,ingredient*,preparation,comment?,nutrition)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>

<!ELEMENT preparation (step*)>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition protein CDATA #REQUIRED
                    carbohydrates CDATA #REQUIRED
                    fat CDATA #REQUIRED
                    calories CDATA #REQUIRED
```

```
alcohol CDATA #IMPLIED>


]>
```

In the DTD, a typical element type defination is as the following.

```
<!ELEMENT recipe (title,ingredient*,preparation,comment?,nutrition)>
```

This defines an element and its possible content. In this case, it defines that the "recipe" element contains five elements:"title,ingredient,preparation,comment,nutrition", of which, the child element "ingredient" can occur zero or any times, and the child element "comment" can occur zero or one time. Each element defined has a content model: a description of the element's expected contents.

There are several keywords and expressions specify an element's content:

- *EMPTY* specifies that the defined element allows no content.
- *ANY* specifies that the defined element allows any content, with no restriction.
- *(#PCDATA)* specifies that the defined element allows only parsed character data.
- Or a mixed content, which means that the content may include at least one text element and zero or more named elements.
- Or an element content, which means that there must be no text elements in the content. Element content consists of:
  - A content particle, which can be either the name of an element declared in the DTD, or a sequence list or choice list. There may follows an optional quantifier.
    - a sequence list indicates that it is an ordered list of one or more content particles. All the content particles must follow a relative order.
    - a choice list means an exclusive list of two or more content particles, only one of these content particles may appear in the content of the defined element.
  - A quantifier is a single character that immediately follows the specified item to which it applies, to indicate the number of successive occurrences of these items.
    - + for specifying that there must be at least one occurrences of the item.

- * for specifying that any number of occurrences of the item is allowed.
- ? for specifying that there must not be more than one occurrence of the item.

Also, there are attribute list declarations such as following.

```
<!ATTLIST nutrition protein CDATA #REQUIRED
                    carbohydrates CDATA #REQUIRED
                    fat CDATA #REQUIRED
                    calories CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>
```

This defines the list of all possible attributes associated with the given element type, it consists of the declared name of the attribute, its data type, and its default value. In this case, the element type "nutrition" has an attribute named "protein", and four other attributes of the "CDATA" data type. Four of them are required and the other is implied, which means that this attribute is not required.

Here are some attribute types supported by XML:

- *CDATA* means characters data and indicates that the value of the attribute can be any textual value.
- *ENTITY* means the effective value of the attribute can only be the name of an unparsed external entity.
- *ID* means the effective value of the attribute must be a valid identifier.

A default value can define whether an attribute must occur (#REQUIRED) or not (#IMPLIED), or if it has a fixed value(#FIXED), or what value should be used as a default value.

2.2.2 XML Schema

XML Schema is an XML-based alternative to DTD. An XML schema describes the structure of an XML document, which consists of constraints on the structure and content of documents. XML Schema are much more powerful than DTDs, because XML Schema support data types, and they are in XML syntax and also they are extensible.

Here is an example of XML Schema:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note" type="xs:string" minOccurs="0"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

</xs:schema>
```

A simple element is defined like:

```
<xs:element name="orderperson" type="xs:string"/>
```

In this case, the data type of the element "orderperson" is string.

A complex element is an XML element that contains other elements or attributes. A complex element is defined like:

```
<xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
```

The element type "shipto" is a complex element, it contains four child elements, which are "name", "address", "city" and "country".

A complex element can also have attributes, where an attribute defination is like:

```
<xs:attribute name="orderid" type="xs:string" use="required"/>
```

It specifies the name, data type of the attribute, and it also indicates that the attribute is required in the element.

There are also some other XML schema languages in widespread use such as RELAX NG and Schematron.

2.3 Regular Expression and XML Validation

Regular expressions provide a concise and flexible means to specify and recognize strings of text. Each character in a regular expression is either understood to be a metacharacter which has a special meaning, or a regular character with its literal meaning. Together, they can be used to identify textual material of a given pattern. The characters in regular expression are divided into two types, literal characters and special characters. Literal characters are just

simple textual characters while there are 11 special characters which have special meanings, such as "?", which indicates a option. There are also operations in regular expressions such as "or", grouping and quantification. A vertical bar separates alternatives, which means pattern matches one of the alternatives. For example $a|b$ can match "a" or "b". Grouping specifies that this part of regular expression can be applied to a operator together. For example, *(ab)* can be referred as a whole. And a quantifier after a token or a group specifies the quantity of the part of the regular expression. The most common quantifiers are the question mark *?*, the asterisk ***, and the plus sign *+*:

- *?* indicates there is zero or one of the preceding element.
- *** indicates there is zero or more of the preceding element.
- *+* indicates there is one or more of the preceding element.

Since content model and data type validation are two major parts of XML validation, and both of which can be referred to a regular expression matching problem. A content model specified using a regular expression over element names while basic regular expressions are over literal characters. DTD analogues of regular expression syntax are given below[2]:

RE Syntax	DTD Syntax	Meaning
ε	EMPTY	no element content is allowed
ab	a,b	both a and b must occur, in order specified
a | b	a | b	one(and only one) of a or b must occur
a*	a*	zero or more occurrences of a must occur
a+	a+	one or more occurrences of a must occur
a?	a?	zero or one occurrence of a must occur
	#PCDATA	content is text rather than an element

In the follow XML Schema example shows how a data type with a specific pattern is defined.

```
<simpleType name="better-us-zipcode">
  <restriction base="string">
    <pattern value="[0-9]{5}(-[0-9]{4})?"/>
  </restriction>
</simpleType>
```

Pattern is a constraint on the value space of a datatype which the literals must match a specific form. The value of pattern must be a regular expression. It's only used to validate whether an entire element matches a pattern or not, rather than for extracting matches from large blocks of data.

2.4 Related Work

Zhang et al [19] present a high-performance XML parsing and validation technique and develop a schema-specific parsing method which uses a two-stack push-down automaton(PDA) for single-pass parsing and validation without backtracking. Several efforts have been made to address the parsing and validation performance through the use of schema-specific (grammar-based) parsers.But these parsers are either time efficient, but encode many states as a result, or space efficient with backtracking. The new schema-specific parsing method described in their paper is the first successful attempt to achieve both time and space optimal. Performance measurements were taken against Xerces and Expat, two widely used runtime-based parsers. The throughput of their work is on average 10 times faster than Xerces parser with validation, and also 2 times faster than non-validating Expat. Performance measurements were also taken against some schema-specific parser, like gSOAP. The result showed that there is still 4 times faster.

Futher work presents a table-driven streaming XML parsing and searching technique called TDX. It speed-up XML parsing, validationg and searching by pre-recording the states of an XML parser. The parsing table combines parsing, validation and search into a single pass. It is pre-processed at compile time, so the looking up takes constant time. TDX implements a single pass predictive validating parser without bachtracking or function calling overheads.

Another related work by Kostoula et al [13, 15, 17] describes a system in which high-performance XML parsers are customized using parser generation and compilation techniques. Parsing is integrated with Schema-based validation and deserialization. In the paper, they analyze a variety of architectural considerations relating to the design of high-performance XML systems, and compared with an experimental prototype implementation known as XML Screamer. It compiles customized validating XML parsers from an XML Schema. On the tests reported in their paper, their XML Screamer can process XML at speeds of roughly 100 to 200 Mbytes per sec on the 4 GHz processors. Screamer is between

1.5 to 3.3 times faster than Expat, and between 3.2 to 5.3 times faster than nonvalidating Xerces. Besides, it delivers from 7.8 to 15.4 times the throughput of Xerces when both parsers are validating.

Perkins et al [16] present a method for generating efficient parsers by using the schema component model itself as the representation of the grammar. XML Schema grammar can be used during parsing to improve performance, but the expressiveness of XML Schema does not fit well to the generic intermediate representations associated with the traditional grammar=based parser generation. Their method enables the use of grammar-sensitive primitives and other forms of specialized an optimistic validation that increase parsing performance significantly.

There is also another related work by Lowe et al [14], which proposed an offline parser generation approach to enhance online processing performance for XML documents conforming to a given DTD. They presented an algorithm that maps DTDs to deterministic context-free grammars, and made them suitable for standard parser generators. They took the approach of specific parsing, which means that a parser applies to documents conforming to a given DTD only. They also compared their implementation to Xerces and expat.

One of the related works by Chiu and Lu [11] developed a framework for schema-specific parsing centered on an intermediate representation which abstracts the computational steps necessary to validate against a schema. They compared their implementation to Libxml2, expat and gSOAP. The result showed that the performance difference between expat and their work was about 7 times. Results suggest that their approach is significantly faster than non-schema-specific parsers.

Thompson and Tobin [18] give complete details on how to convert W3C XML Schema content models to Finite State Automata, including numeric exponents and wildcards. They also described Enforcing the Unique Particle Attribution constraint and implementing restriction checking in polynomial time using their FSAs.

Chapter 3

Content Model Validation

3.1 Problem Statement

Given an object XML document of a particular class and corresponding markup declarations that provide a grammar for a class of documents in the form of DTD or XML schema, the content model validation checks the object XML document against the element content model grammar that has been extracted from the markup declarations. The content model here is defined as the constraint for the element contents.

3.2 Algorithm Overview

As shown in figure 3.1, there are three stages in this algorithm. The first stage extracts the element content model grammar from the markup declarations. The second stage builds up the element name pool and also the content arrays. The third stage checks the arrays against the element content model grammar by applying parallel bit stream technology.

In the first stage, structures of element content model grammar are built by parsing the XML schema documents. The element content model grammar is built on content particles, which consist of names, choice lists of content particles, or sequence list of content particles. An element type should have a corresponding content model structure which indicates information above.

In the second stage, an element name alphabet is constructed. Each element type will have a unique general GID. All the GIDs are of identical bit width, which means for all the GIDs, the number of bits to represent them will be the same. During the process of the

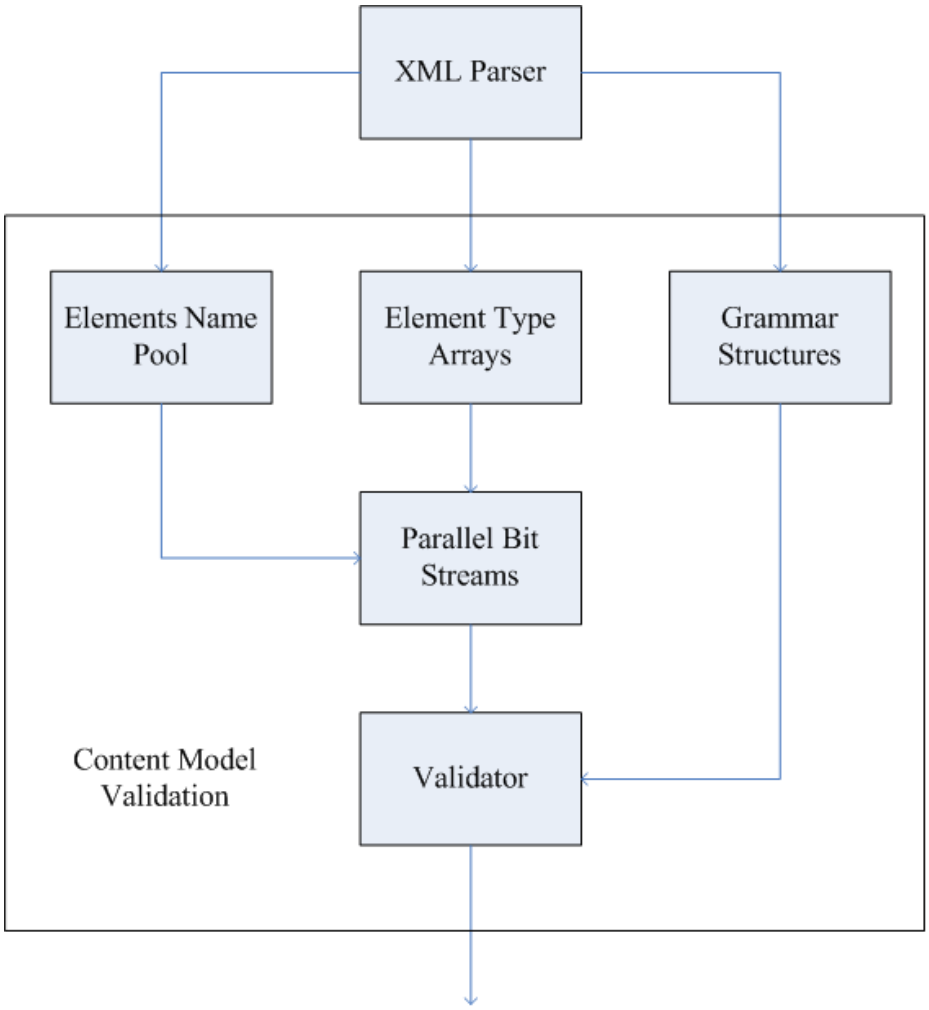

Figure 3.1: Architecture

parsing of the object document, element content arrays are build up. One array of a certain element type consists of the GIDs of all the appearances of the children of this element type, along with signs of every closing of this element type. So there is one array for each element type.

In the validation stage, each array will be checked against the corresponding element content model grammar. Firstly, the arrays of bytes are transformed to a set of 8 or 16 parallel bit streams, depending on the bit width of each GID. Based on the grammar, each GID of the children, choice lists and sequence lists are checked in parallel by using bitwise logic and shifting operations. If there is any invalid child, or misordering of the children, the error will be indicated.

Section 3.3 introduces the extraction of the element content model grammar and what information is in the grammar.

Section 3.4 presents how the alphabet is defined.

Section 3.5 indicates how the element type arrays are constructed.

Section 3.6 considers how the validation is carried out on the arrays against the grammar by employing the parallel bit stream technology.

3.3 Content Model Extraction

In DTD documents, an element definition consists of the ELEMENT keyword, the element name, and the content it can contain. For the element content model, we only focus on the content about other elements that the element can contain. While in XML schema documents, adding children to an element requires the use of complex types. To list one or more child elements, a sequence element is needed. For the case where there is only one child element from a list of alternatives, choice element is needed.

As shown in Table 3.3, by using the parallel bit streams, we can mark every element name start and end positions, and then locate each element instance in the document. Futhermore, by removing the first mark in the streams by only one bit-wise operation, we can easily located the next element name in the source file. It is how the syntactical items are located and gathered.

During the process of DTD or XML schema document parsing, the element content model is extracted and stored as element content model structures. Each structure consists of all possible children of the element, a sequence of children and choice alternatives of

Table 3.1: XML Example 1

```
<library>
  <book id="b0836217462" available="true">
    <isbn>
      0836217462
    </isbn>
    <title lang="en">
      Being a Dog Is a Full-Time Job
    </title>
    <author id="CMS">
      <name>
        Charles M Schulz
      </name>
      <born>
        1922-11-26
      </born>
      <dead>
        2000-02-12
      </dead>
    </author>
    <character id="PP">
      <name>
        Peppermint Patty
      </name>
      <born>1966-08-22</born>
      <qualification>
        bold, brash and tomboyish
      </qualification>
    </character>
    <character id="Snoopy">
      <name>
        Snoopy
      </name>
      <born>1950-10-04</born>
      <qualification>
        extroverted beagle
      </qualification>
    </character>
  </book>
</library>
```

Table 3.2: Schema Example 1

```
<xs:element name="library">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="book" maxOccurs="unbounded">
        <xs:complexType>
           <xs:sequence>
             <xs:element ref="isbn"/>
            <xs:element ref="title"/>
            <xs:element ref="author" minOccurs="0"
              maxOccurs="unbounded"/>
            <xs:element ref="character" minOccurs="0"
              maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute ref="id"/>
          <xs:attribute ref="available"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

children.

One DTD document or XML schema document can refer to XML documents of one class of application. So when processing multiple XML documents of the given class, the content model extraction only needs to be done once.

For the XML Schema in table 3.2, the grammar structure of element type “book” is like table 3.4. The element type “book” has a sequence of children, which consists of element type “isbn”, “title”, “author” and “character”. The type “isbn” and “title” must occur only once, while elements “author” and “character” should follow the “title” element, and they can occur from zero to unbounded times. The table doesn’t show the grammar structure of the element type “library”, which consists of a sequence of elements of “book”, and they can occur from zero to unbounded times.

Table 3.3: Content Model Gathering Example

Source Text	</address><city>Vancouver</city><country>
Start Tag Marks	___________1_____________________1_______
Element Name Start	__1________1_____________1_____1_______
Element Name End	_________1_____1_____________1________1

Table 3.4: Grammar Structure

book	
isbn	
title	
author	minOccurs="0" maxOccurs="unbounded"
character	minOccurs="0" maxOccurs="unbounded"

3.4 Element Name Alphabet

A system of notation is built to represent an element with a unique GID, which is called element name alphabet. The GIDs should have same bit width. Depending on the quantity of the kinds of different element type, the bit width of the GID is set as 8 or more bits. 8-bit GIDs can represent up to 256 different element types, while 16-bit GID is for up to over 40000 element types. For most cases, a document will contain less than 256 different element types, so only 8-bit GIDs are needed.

As for some XML documents with the same document type, the number of the kinds of different elements can be represented by less than 8 bits. Because in the validation stage, the bytes of the GIDs will be transformed into a set of bit streams, the fewer bits we need to represent the GID, the fewer number of bit streams we will need, and the fewer operations we will need to do each validation process.

The GIDs are allocated to the elements by the order the elements occur in the schema document, by which a map of the content model structures and the element GIDs are constructed. There will also be a GID representing the closing of a element.

For the object XML document shown in Table 3.1, the element name alphabet looks like Table 3.5. Every type of elements will get a corresponding unique GID, besides, there is a special type called element closing, which has a GID as zero. There are 10 types of elements in the object XML document, so the GIDs assigned to the types are from 1 to 10. The element closing type indicates where one element is closing.

Table 3.5: Element Name Pool

Element Name Pool	
Element Type	ID
library	1
book	2
isbn	3
title	4
author	5
character	6
name	7
born	8
dead	9
qualification	10
element closing	0

3.5 Content Array

The content arrays are constructed while parsing the object XML document. Each element type has a corresponding content array for one object document. When parsing the content of element A, if element B occurs, the GID of element B will be appended to content array of element A. When the element is closing, the GID of current element closing will be appended to the content array. The content array of element A only contains A's direct children.

The other elements should also have one content array for each element. Each content array contains the information of the content of the element for the entire document.

The content arrays of the example XML document in table 3.1 are shown in table 3.6. For the element type "book", it's a sequence of its children. First GID 3 indicates element type "isbn", and follows a element of type "title" and so on. The last position of each array is an GID 0, which indicates the element closing positions of each element type.

Table 3.6: Content Array

library	2	0						
book	3	4	5	6	6	0		
author	7	8	9	0				
character	7	8	10	10	0			

3.6 Regular Expression Transformation

To validate the content model against the grammar with parallel bit streams, we need to generate the bitwise operation code from the regular expression of the content model grammar. In the example above, from the content model grammar of the element "book" and the element name pool, we can get the regular expression of the element "book", which is:

```
2(34(5)*(6)*)0
```

In the regular expression, each GID except "0" represents a element type, while GID "0" represents the end of the current element.

To input the regular expression into the Regular Expression bitwise code generator, it will generate the code of the bitwise validation function of the corresponding element.

The XML representation for a model group schema component includes "all", "choice" and "sequence" element information items. The "choice" and "sequence" items can be represented using regular expression easily, however the "all" item allows the elements to occur in any order, which is not easy for regular expression to represent, so the "all" item validation is not included in the validation algorithm for now.

3.7 Validation

In the validation stage, the content arrays are validated against the content model structures one by one. Assuming the bit width of the GIDs is 8, each array is transformed in a set of 8 parallel bit streams, each stream represents one bit position of the 8 bits of the GIDs. They are called basic streams. As shown in table 3.7.

For each element type, related element types and their GIDs are already known according to the grammar. Element type streams will be generated by applying bitwise operations on the basic streams. An element type stream is to identify a GID position at which the element type occurs. A stream for the closing GID is also generated.

From the regular expression of the grammar of the object element, the bitwise operations are applied on the basic bit streams, and the match stream is generated. In the example above, there are two element occurrences but only the first one is a match, because one child of the second occurrence of the element "book" is missing. Then the "xor" operation is applied on the match stream and element_occurrence stream, to find the errors.

Table 3.7: Basic Streams and Match Streams

book	3	4	5	6	6	0	3	5	6	0
basic8	0	0	0	0	0	0	0	0	0	0
basic7	0	0	0	0	0	0	0	0	0	0
basic6	1	1	1	1	1	1	1	1	1	1
basic5	1	1	1	1	1	1	1	1	1	1
basic4	0	0	0	0	0	0	0	0	0	0
basic3	0	1	1	1	1	0	0	1	1	0
basic2	1	0	0	1	1	0	1	0	1	0
basic1	1	0	1	0	0	0	1	1	0	0
element_occurrence	0	0	0	0	0	1	0	0	0	1
match	0	0	0	0	0	1	0	0	0	0
error	0	0	0	0	0	0	0	0	0	1

Chapter 4

Data Type Validation

4.1 Problem Statement

Given an object XML document of a particular class and corresponding markup declarations that provide a grammar for a class of documents in the form of DTD or XML schema, the data type validation checks the content of the elements of the object XML document against the data type constraint of the element.

4.2 Algorithm Overview

There are three stages in this algorithm. The first stage parses the DTD or XML schema document, extract the element content data type constrain of each class of elements. The next stage of the algorithm extracts the content data of each class of elements in a corresponding buffer, to prepare for the transformation to parallel bit streams. The last stage validates the content data streams by using parallel bit stream technology.

In the first stage, the element content data type constraints can be extracted in the process of XML schema documents in chapter 3. We will discuss the definition of datatype in section 4.3. The elements with an identical type of content data will be assigned to one group.

In the second stage, element content data will be extracted from the object document, and byte streams of the content data of different element group will be formed. The data from different element instances of the same group will be consecutive in the streams.

In the last stage, the byte streams are first transformed to a set of 8 parallel bit streams.

Then the content data are validated against the data type constraints in parallel. When an invalid data is found, an error will be indicated.

Section 4.3 introduces the data type system of XML Schema.

Section 4.4 indicates how the content data is extracted, and how the data streams is built during the parsing of the instance file.

Section 4.5 presents how the constraint of the datatypes is translated into regular expressions.

Section 4.6 introduces that how the validation of datatypes is carried out on the data streams against the definition by employing the parallel bit stream technology.

4.3 Data Type System

Datatypes are part of the specification of XML Schema language. They define facilities for datatype definitions that are used in XML Schema. In this specification, a datatype consistes of three properties, a value space, a lexical space and a small collection of functions, relations and procedures associated with the datatype.

The value space of a datatype is the set of values for the datatype. The relations of identity and equality are required for the value spaces. The validation of the value space of the datatypes is not covered in the algorithm.

The lexical space of a datatype is the prescribed set of strings of the datatype. Functions, relations and procedures associated with the datatype is the lexical mapping. The validation of lexical space and lexical mapping is the object of the algorithm.

Atomic datatypes are the types whose value spaces contain only atomic values. Boolean, dateTime and double are all atomic datatypes. One of the relations of the atomic datatypes is list. List datatypes are the values consist of finite-length sequences of atomic values. The lexical space is composed of space-separated literals of the atomic type. Another lexical mapping is called union. Union types may be defined as “ordered unions” of atomic datatypes of union types.

4.4 Data Extraction

The data extraction stage pulls out the content of particular datatypes, and contributes a data stream for each datatype. For each datatype, there is a corresponding data stream that

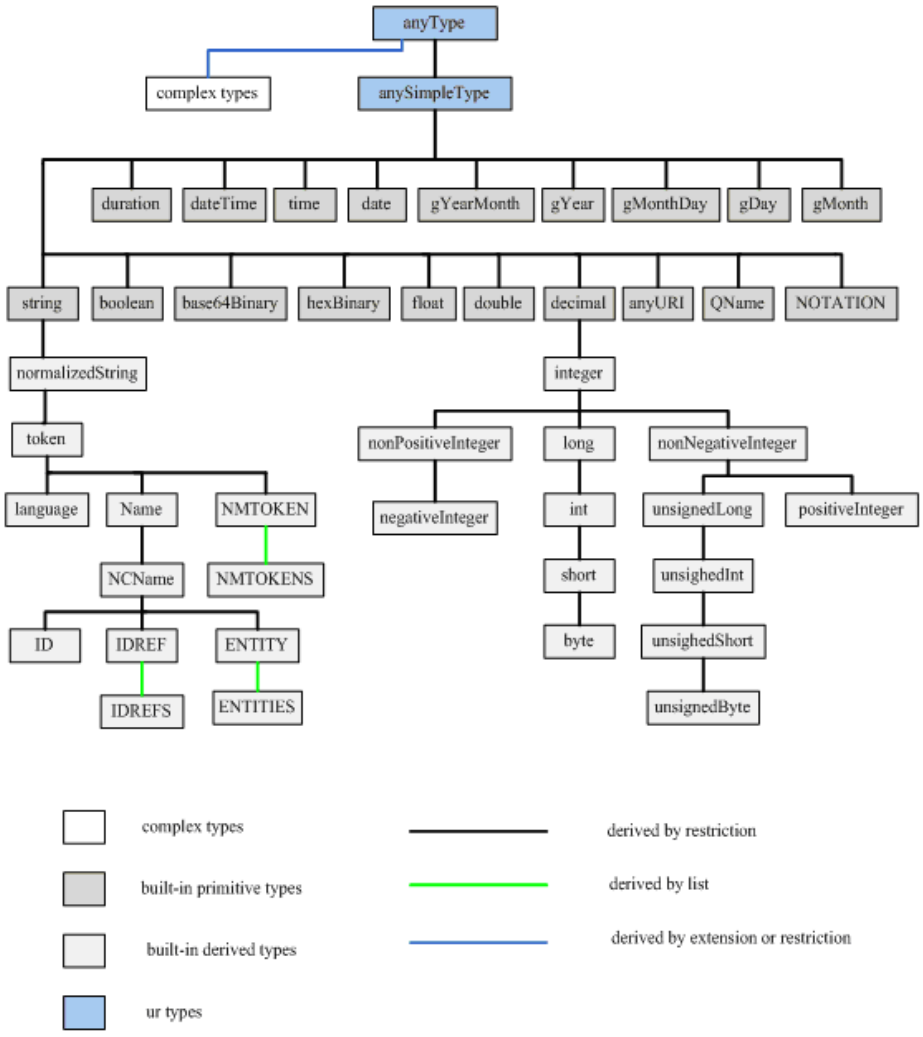

Figure 4.1: XML datatypes

consists of the content data of the whole object document of that datatype, and splitting characters to separate data of each occurrence.

The elements will be classified by the content datatype during the parsing of the XML Schema document. When parsing the object document, content data of a same element class will be appended into a stream of that type consecutively. In Table 4.1, a slice of a xml document, the data type of the content of the element "textureCoordinates" is "doublelist" as defined in the schema in Table 4.2, which is a list of doubles split by white spaces. The data is extracted into a buffer, which contains all the data of the element "textureCoordinates", split by identical symbols of one byte, to indicate the end of the data content of each occurrence of the element. In this case, the splitting character is "#", which will not appear in any valid data of type "doublelist".

Table 4.1: XML Example 2

```
<app:target uri="#PolyID16_1737_484245_123631">
   <app:TexCoordList>
      <app:textureCoordinates ring="PolyID16_1737_484245_123631_0">
152.196947454962 1.3123359580052 150.425219300984
   </app:textureCoordinates>
   </app:TexCoordList>
</app:target>
<app:target uri="#PolyID99_1141_449345_378934">
   <app:TexCoordList>
      <app:textureCoordinates ring="PolyID99_1141_449345_378934_0">
0.757565797553053 -0.105958583378923 0.757314870865753
   </app:textureCoordinates>
   </app:TexCoordList>
</app:target>
```

The datatype buffer of the example in table 4.1 are shown in table 4.3. Every occurrence of the data of type "doublelist" is extracted into a buffer in order, and the splitting character "#" is put in between data of different element occurrences.

4.5 Data Type and Regular Expressions

Most of the datatypes defined by Schema can be represent by regular expressions, as described in Appendix C. And there are two ways that other datatypes can be derived by

Table 4.2: XML Schema Example 2

```
<xs:element name="textureCoordinates" type="doublelist"/>

<xs:simpleType name="doublelist">
  <xs:list itemType="xs:double"/>
</xs:simpleType>
```

Table 4.3: Data Buffer of Doublelist

1	5	2	.	1	9	6	9	4	7	4	5	4	9	6
2		1	.	3	1	2	3	3	5	9	5	8	0	0
5	2		1	5	0	.	4	2	5	2	1	9	3	0
0	9	8	4	#	0	.	7	5	7	5	6	5	7	9
7	5	5	3	0	5	3		-	0	.	1	0	5	9
5	8	5	8	3	3	7	8	9	2	3		0	.	7
5	7	3	1	4	8	7	0	8	6	5	7	5	3	

built-in datatypes: “List” and “Union”. List datatypes are those having values, each of which consists of a finite-length sequence of values of an atomic datatype. Union datatypes are union of one or more other datatypes. The list and union logic can be fulfilled by regular expressions, so as long as the atomic datatypes can be represented by regular expressions, list and union datatypes of them can be represented as well.

In this case, the regular expression of the datatype “doublelist” is as follow:

```
^[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?([ ]+[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?)*$
```

However, there are some datatypes and some value constraining facets which can not be represented by regular expressions easily in the data type system of XML Schema, such as the length of the data value, maximum and minimum value of the data value.

4.6 Validation

To validate the datatypes against the grammar with parallel bit stream technology, the bitwise functions for each datatype are generated based on the regular expressions. The regular expressions are input into the regular expression bitwise code generator, to get the bitwise validation functions of all the datatype occurs in the instance file.

Also the data streams are transformed into 8 parallel bit streams, each stream represents one bit position of each byte of the data. The validation function is applied on the corresponding basic bit streams, every match is marked in a bit stream, at the position of the end of each occurrence of the data. Then the “xor” operation is applied on the match stream and splitting character stream, to find any mismatch.

Chapter 5

Performance Studies

In this chapter, performance studies of the implementation of the algorithm on different XML schemas and the comparison against the well-known XML parser and validator Xerces are shown. Also the difference in performance of the implementations with different parameters such as the size of the datatype buffer are also studied. The instance documents in this chapter are all generated by schema based XML file generators, the size of the instance files are over 10 Mbytes. All the experiments are carried out on 64-bit CPU with 3.4 Ghz and SSE4.1/4.2 and AVX instruction sets.

5.1 Performance Study 1

The instance files in this section are all based on the schema in Appendix A. The schema shows that only one datatype needs to be validated, which is "doublelist", so for the implementation, only one data buffer is needed.

Table 5.1: Measurements

instance file	1	2	3	4	5	6	7	8
data proportion	9.7%	25%	35%	36%	51%	60%	83%	92%
element occurrence per Kbyte	145.67	109.50	104.19	51.70	79.26	64.55	29.07	11.49
symbol table(cycle/byte)	7.66	6.46	5.80	5.53	4.75	3.92	1.92	1.17
datatype validation(cycle/byte)	1.19	2.17	2.52	2.49	2.95	3.36	3.78	3.80
datatype gathering(cycle/byte)	1.24	1.28	1.81	1.91	1.69	1.84	1.30	0.75
content model validation(cycle/byte)	2.39	1.47	1.94	2.04	1.75	1.71	1.46	1.28
content model gathering(cycle/byte)	2.05	1.16	1.44	1.12	0.89	0.88	0.34	0.06

Table 5.1 provides benchmark data for parallel validation on instance files with different characteristics as data proportion and element proportion. The performance of each components of parallel validation are measured in term of cycle per byte. Each column in Table 5.1 represents an XML instance.

Performance Analyses

The over all performance achievable with parallel validation ranges from 6.34 to 9.17 cycle-per-byte, which are from 11.04 to 15.33 instance-per-byte accordingly, including content model and datatype validation.

Symbol Table Performance

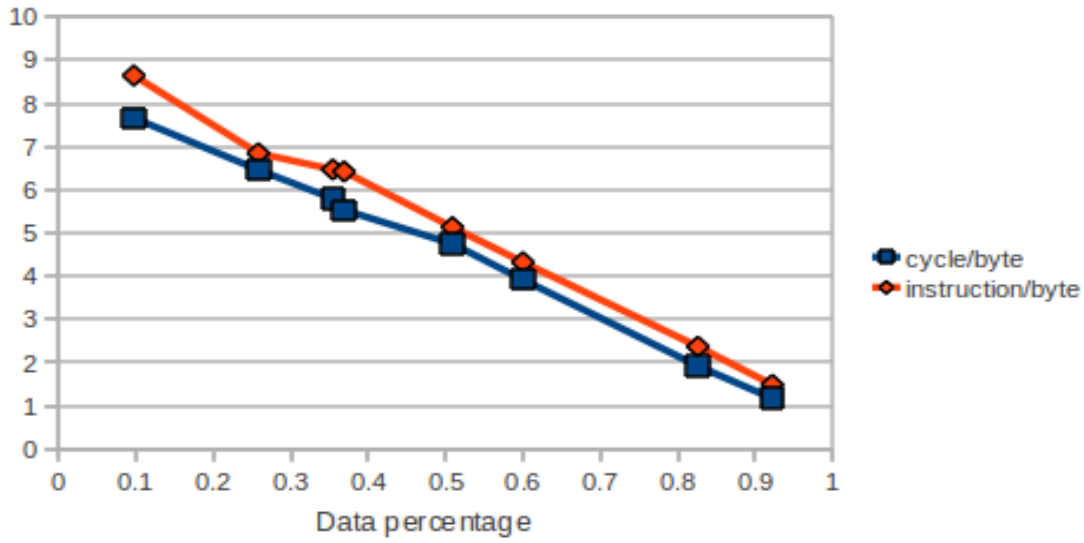

Figure 5.1: Symbol Table Performance

The symbol table contains the element name and GID informations, which is used to implement element name pool used for both content model validation and datatype validation. As shown in Figure 5.1, the overhead of symbol table reaches a peak when the percentage of the data gets to the lowest point of about 9%, which is less than 8 cycle-per-byte. Then the overhead of symbol table module drops as the percentage of the content data increases.

The reason is mainly that as the percentage of the data increases, the number of both the occurrences of content data and element instances decreases, each time there is an occurrence of those, there will be a memory access, hash function will be applied. So in this case, the performance of symbol table module increases, while the percentage of the content data increases.

As shown in Table 5.1, as the data proportion increases from 9.7% to 92%, the number element occurrence per Kbyte decreases from 145.67 to 11.49, while the overhead of symbol table module decreases from 7.66 cycle-per-byte to 1.17 cycle-per-byte. And from Figure 5.1, we can tell that the relation of the overhead of symbol and the data proportion is linear, and the data proportion has reached 90%, almost the maximum percentage of a document that content data can be, so the maximum overhead of symbol table is about 9 cycle-per-byte.

The overhead of symbol table should also be affected by the number of element types in the instance file, since the more element types the instance file has, the more overhead it will have for hashing and storing the information, so the performance of symbol table will be different for other cases.

Datatype Validation Performance

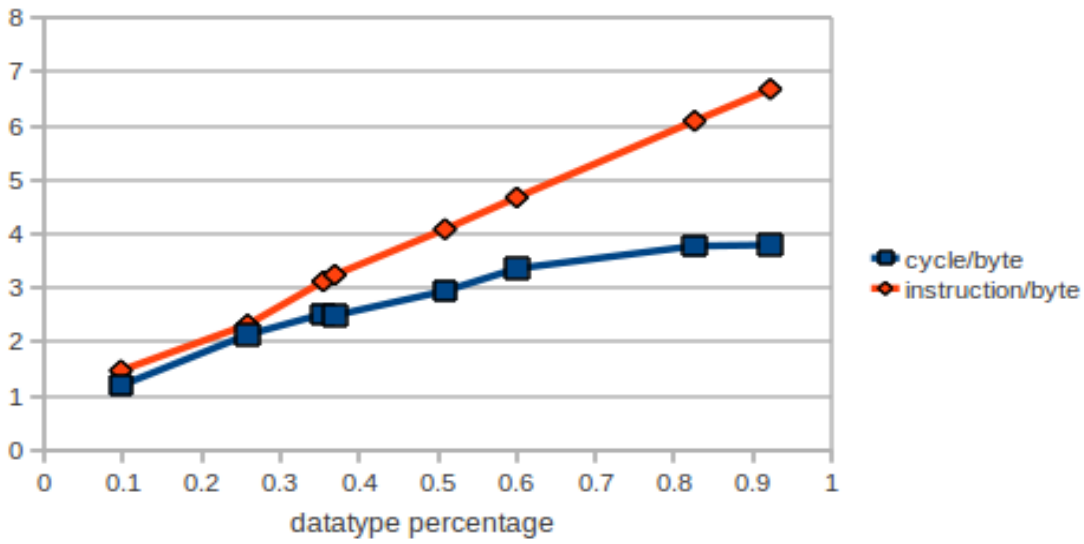

Figure 5.2: Datatype Validation Performance

The performance of the datatype validation component is mainly affected by the data proportion of the instance file, as shown in Figure 5.2, the overhead of the datatype validation increases as the percentage of the datatype in the instance file increases, and reaches a peak when the proportion of the datatype is about 90% of the instance file. The peak over head of the datatype is under 4 cycle-per-byte, or 7 instruction-per-byte. The overall overhead of datatype validation component includes two parts, the overhead of data gathering and data validation. The overhead of data validation is dependent on the number of bit-wise operations of the validation function, so the overhead of data validation is more stable in this case. While the overhead of the data gathering is dependent on the data proportion and the number of the occurrences of content data. Because there is at least one memory copy operation each occurrence of the content data, so when the data proportion and the number of the occurrences are higher, the overhead of the whole datatype validation module is higher.

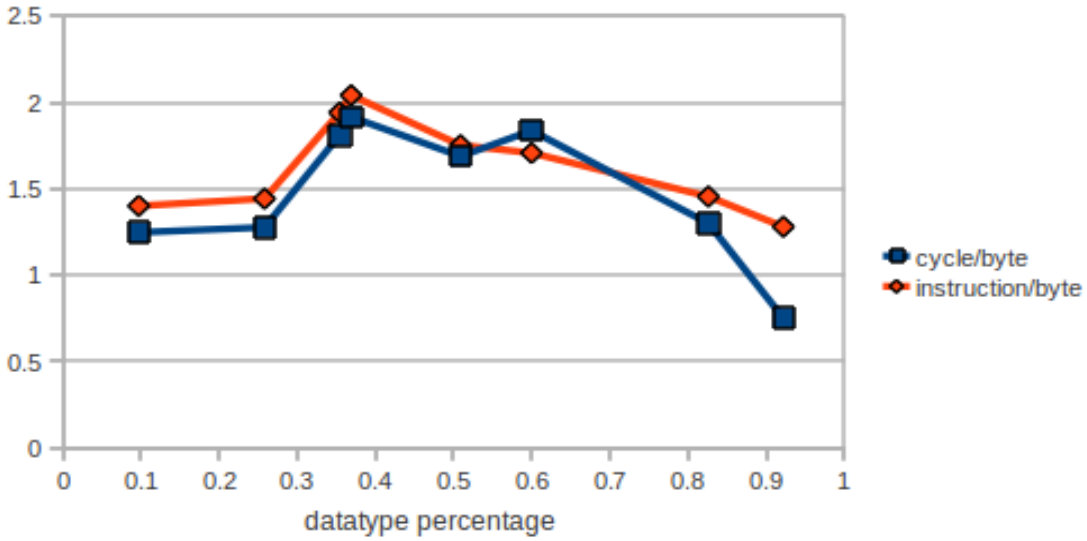

Figure 5.3: Datatype Gathering Performance

The gathering of the data is a component of the datatype validation, which is also affected by the data proportion of the instance file. As shown in Figure 5.3, the overhead of data gathering varies as the percentage of the datatype increases, reaching a peak of about 2 cycle-per-byte, or over 2 instance-per-byte when the 40 percent of the instance file is of

the datatype.

Content Model Validation

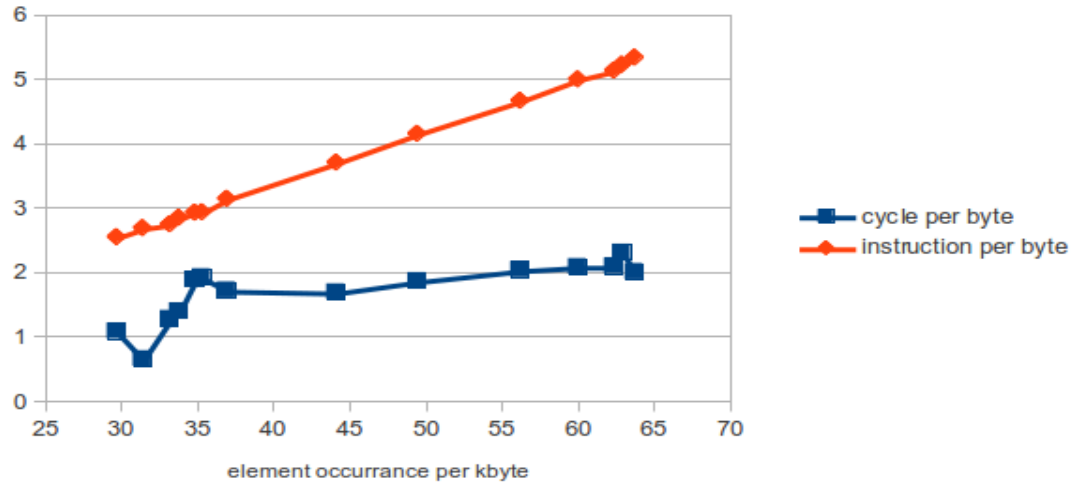

Figure 5.4: Content Model Performance1

As shown in Figure 5.4, the overhead of content model validation reaches a peak when the percentage of the datatype is about 9%, which is less than 3 cycle-per-byte. And when the data is of over 90% of the instance file, the overhead of content model is the lowest, less than half cycle-per-byte. The overhead of the content model validation module includes two parts, first of which is the overhead of the gathering of content model, second is the overhead of content model validation. The overhead of content model validation is dependent on the average number of occurrences of element instances and the number of operations of the validation functions. The number of the operations in the validation functions vary for different element types. The overhead of content model gathering, however, is dependent mainly on the average number of occurrences of element instances, because each occurrence will lead to a hashing and memory access operation, and compared to the overhead of the validation functions, the overhead of the gathering dominates. So the most contribution of the overhead of the content model validation module is the gathering of the content model.

The overhead of content model is not only affected by the percentage of the datatype,

but also by the number of element per Kbyte and the average length of the element names. The number of elements varies from 11 to 145, and the overhead of content model changes in the range of 0.16 to 2.38 cycle-per-byte.

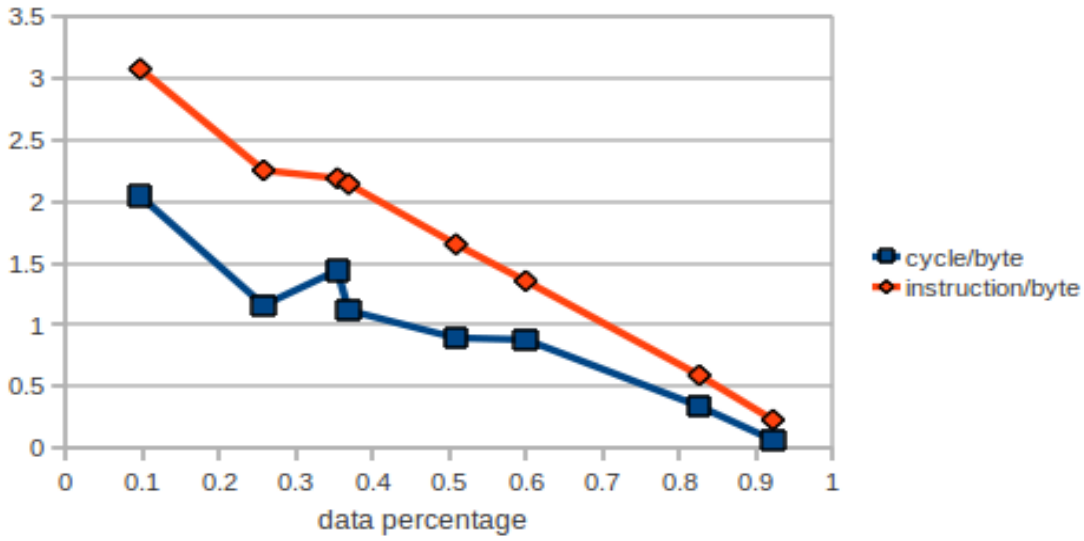

Figure 5.5: Content Model Performance2

Because that we have already addressed that the most contribution of the overhead of the content model validation module is the content model gathering. The range of the overhead falls in the range of 0.06 to 2.04 cycle-per-byte.

Performance Summary

As shown in Figure 5.6, the performance of the validation module, without the symbol module, doesn't change very much. It ranges from 3 to 5 cycle-per-byte. The over all overhead including symbol table falls into a range from 4 to 12 cycle-per-byte.

Compared with the overhead of the validation only, the overhead of the symbol table in some cases can be the most contribution. Because as we discussed, as the change of the overhead of the symbol table is more than the change of the overhead of the validation modules, while the characteristic of the instance file changes, so after some point, the overhead of the symbol table will dominate in the overall performance.

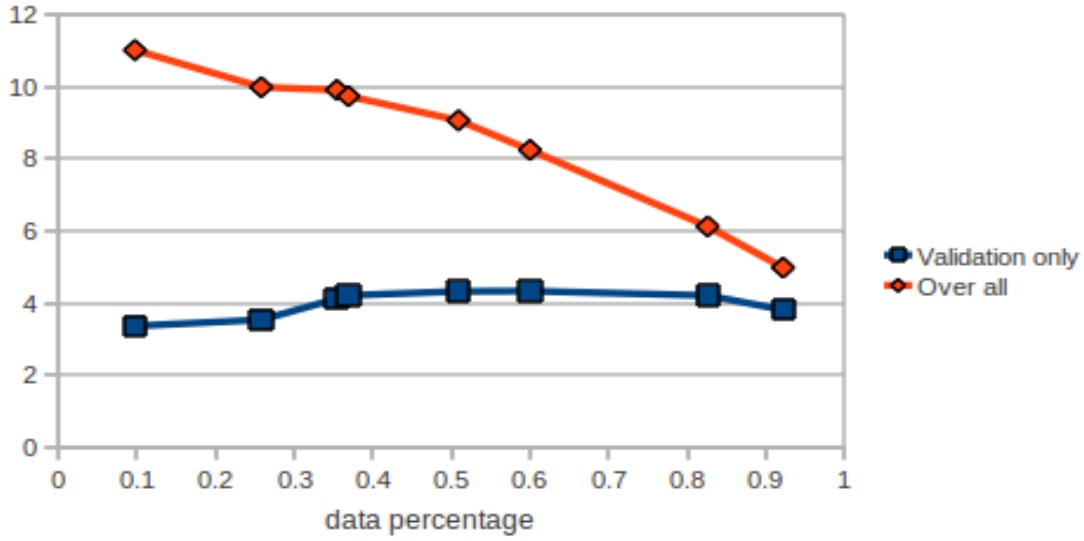

Figure 5.6: Over All Performance1

5.2 Performance Study 2

The instance files in this section are all based on the schema in Appendix B. The schema shows that there are there are four datatypes needed to be validated. Besides, there are three complex element types, including content model of "sequence" and "choice" rules.

The instance files in this section are more complex than the files of section 5.1, there are 4 different datatypes need to be validated, and the rules of content model constraint is also more complex.

Datatype Gathering

As shown in Figure 5.7, the overhead of datatype gathering increases as the data occurrence per Kbyte increases.

So the performance of datatype gathering is mainly dependent on the occurrence of data per Kbyte of the instance file, as the range of the occurrence of data per Kbyte changes from 0 to 65, the overhead of datatype gathering falls into a range of 1.14 to 5.30 cycle-per-byte, or 1.50 to 9.33 instruction-per-byte. Because each time there is an occurrence of the content data, there is at least one memory copy operation, when the length of the data is long or

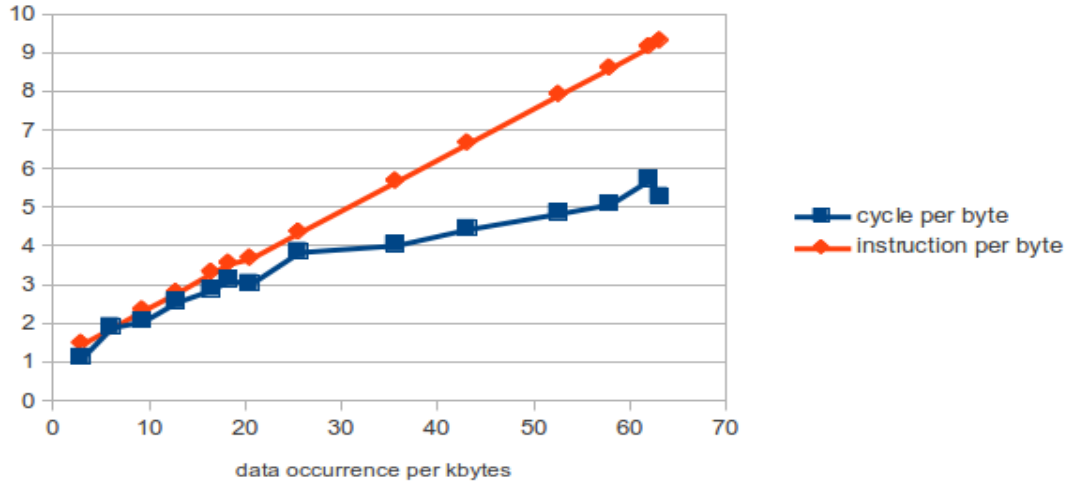

Figure 5.7: Datatype Gathering Performance

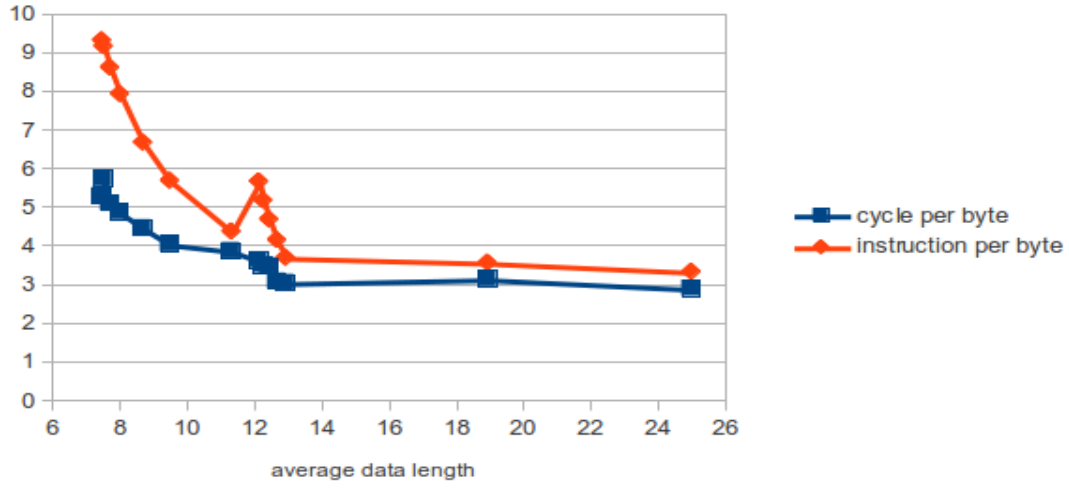

Figure 5.8: Datatype Gathering Performance 2

it falls to the boundary of the buffers, there can be multiple memory copy operations. And the overhead of the memory copy dominates in the overhead of the datatype gathering, so the performance of datatype gathering is mostly dependent on the average number of occurrences data per Kbyte of the instance file.

The difference between the files in this section and in section 5.1 is that, there are multiple datatypes need to be validated in the files in this section. For datatype gathering module, multiple datatypes will lead to overhead of datatype buffer entry searching, cursor locating and so on. So the overhead of data gathering module in this section is more than the sample case in section 5.1, and the result in this section is more common in XML documents.

As shown in Figure 5.8, the overhead of datatype gathering is less dependent on the average data length of the instance file. Because the overhead of datatype gathering dominates by the overhead of memory copy operations. And the overhead of memory copy operations dependents less on the length of the data copied, so for the same amount of total data, the shorter average length of each occurrence of the data it is, the more copy operations it will need and the more overhead there will be for datatype gathering.

Content Model Gathering

Figure 5.9 indicates that the performance of content model gathering is mainly dependent on the occurrence of elements per Kbyte of the instance file. Because when we gather the content model information, the element names are transformed into GIDs, the performance is less dependent on the average length of element names. As shown in the figure, the overhead of content model gathering module ranges from 2 cycle-per-byte to less than 6 cycle-per-byte, which is from less than 1 instruction-per-byte to less than 3 instruction-per-byte.

The content model gathering module is the same as in section 5.1, the only difference is the number of the element types that need to be validated. Since the overhead of content model gathering is mainly dependent on the number of element instances, the figures in both sections show the same result.

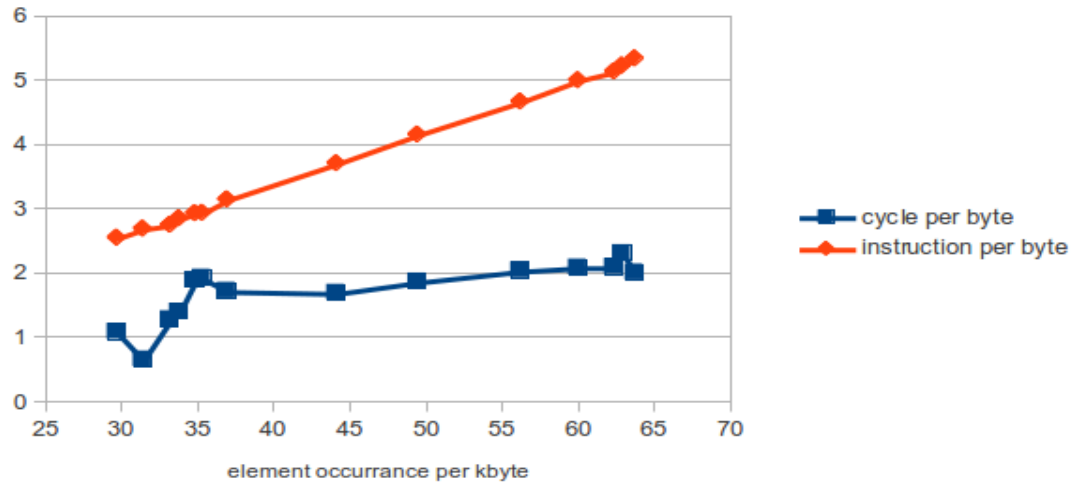

Figure 5.9: Content Model Gathering Performance

Datatype Validation

In this section, the performance of datatype validation module is discussed, the overhead of datatype gathering is not considered.

The performance of datatype validation is dependent on data occurrence, average data length and data bytes per Kbyte of instance file. Because the number of operations for the validation of each data type varies, the performance of over all datatype validation depends on the proportion of each type of data as well. As shown in Figure 5.10, The overhead of datatype validation falls into a range from 0.03 to 2.39 cycle-per-byte, or 0.17 to 5.19 instruction-per-byte.

It is different from the performance study 1 in section 5.1, because the current instance file is based on a schema which has more than one data type to be validated, the performance of datatype validation is also dependent the proportion of each different datatypes. However, the number of operations of the validation function for different datatypes dependent the regular expressions of the datatypes. Usually the more complex the regular expression is, the more operations there are. But since the operations are applied on the register parallel, the difference will be reduced.

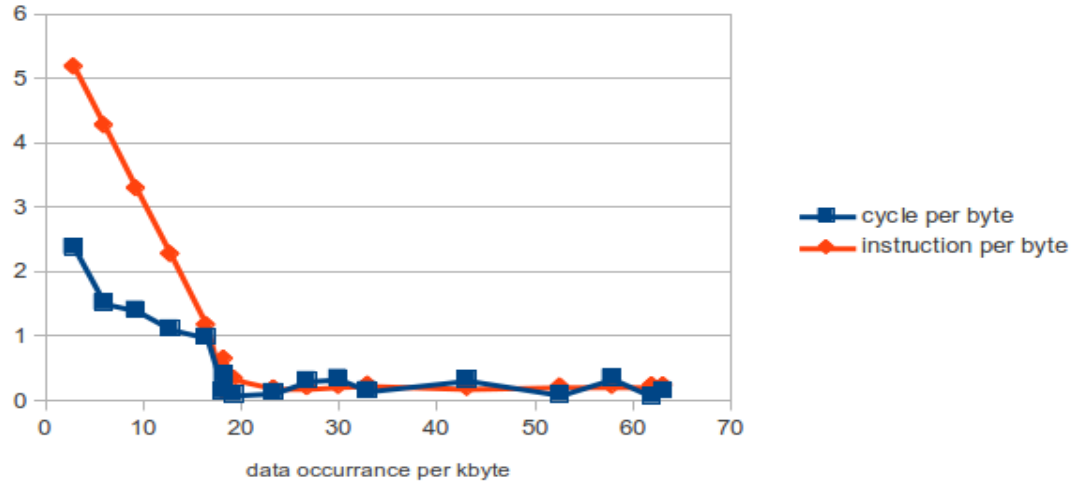

Figure 5.10: Datatype Validation Performance 1

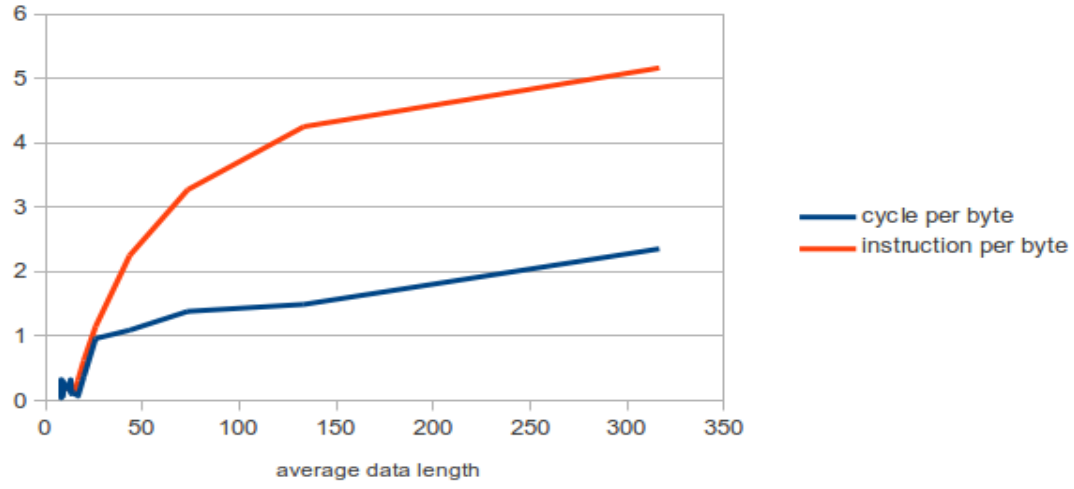

Figure 5.11: Datatype Validation Performance 2

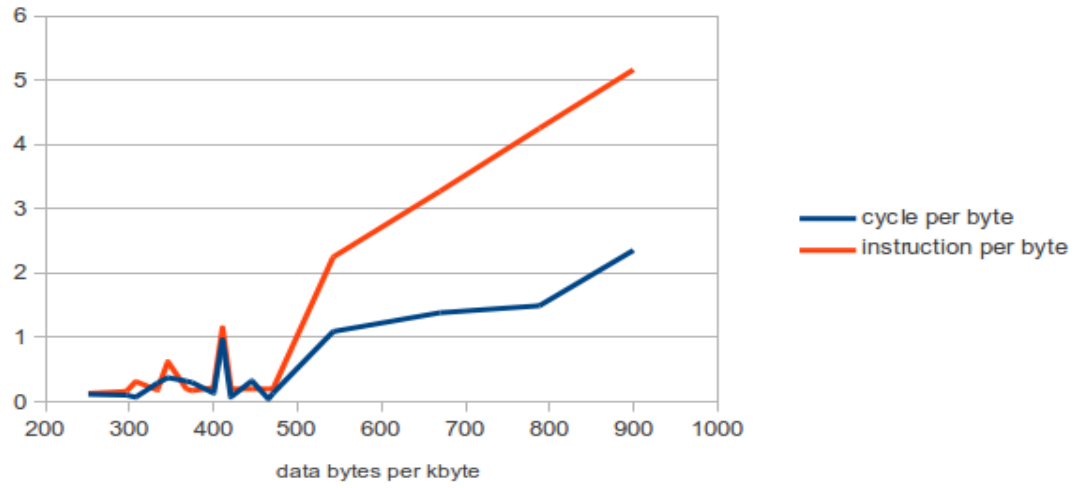

Figure 5.12: Datatype Validation Performance 3

As shown in Figure 5.12, the overhead of datatype validation increases when the data proportion increases, before the overhead of datatype validation reaches more than 1 cycle-per-byte, the affect of the change of data proportion to the performance of the datatype validation module is very small, which is lest than 1 cycle-per-byte.

Content Model Validation

In this section, the performance of content model validation module is discussed, the overhead of content model gathering is not included.

The performance of content model validation varies with the number of occurrence of element per Kbyte of the instance file. Because the number of operations to validate different element types are not the same, the performance of content model validation is dependent on the proportion of each element type as well. As the occurrence changes in the range of 4.64 to 63.66, the overhead of content model validation falls into a range of 0.16 to 1.18 cycle-per-byte, or 0.33 to 3.02 instruction-per-byte.

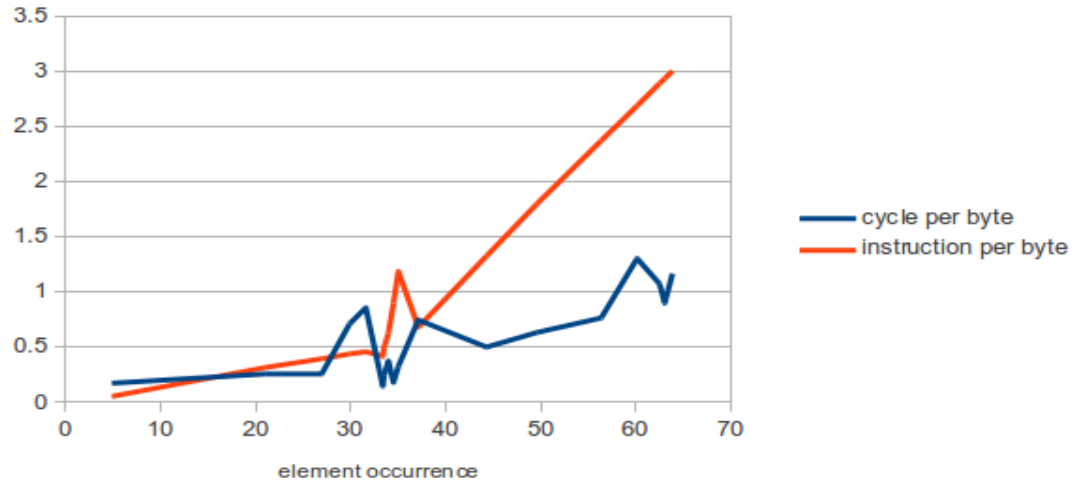

Figure 5.13: Content Model Validation Performance

Over All Validation

The over all performance of validation is the subtotal of both the content model validation and datatype validation, since the characteristics of the instance files are different, in term of the percentage of the data bytes and element occurrence, the performance varies in a range from 0.19 to 2.58 cycle-per-byte, or 0.60 to 5.26 instruction-per-byte. As shown in Figure 5.14.

5.3 Performance Study 3

The instance files in this section are all based on the schema in Appendix B. The schema shows that there are there are four datatypes needed to be validated. Besides, there are three complex element types, including content model of “sequence” and “choice” rules.

Performance Against Xerces

As shown in Figure 5.15, as the element occurrence per Kbyte increases, the overhead of the validation of Xerces rises rapidly. The performance of the validation of Xerces is tested

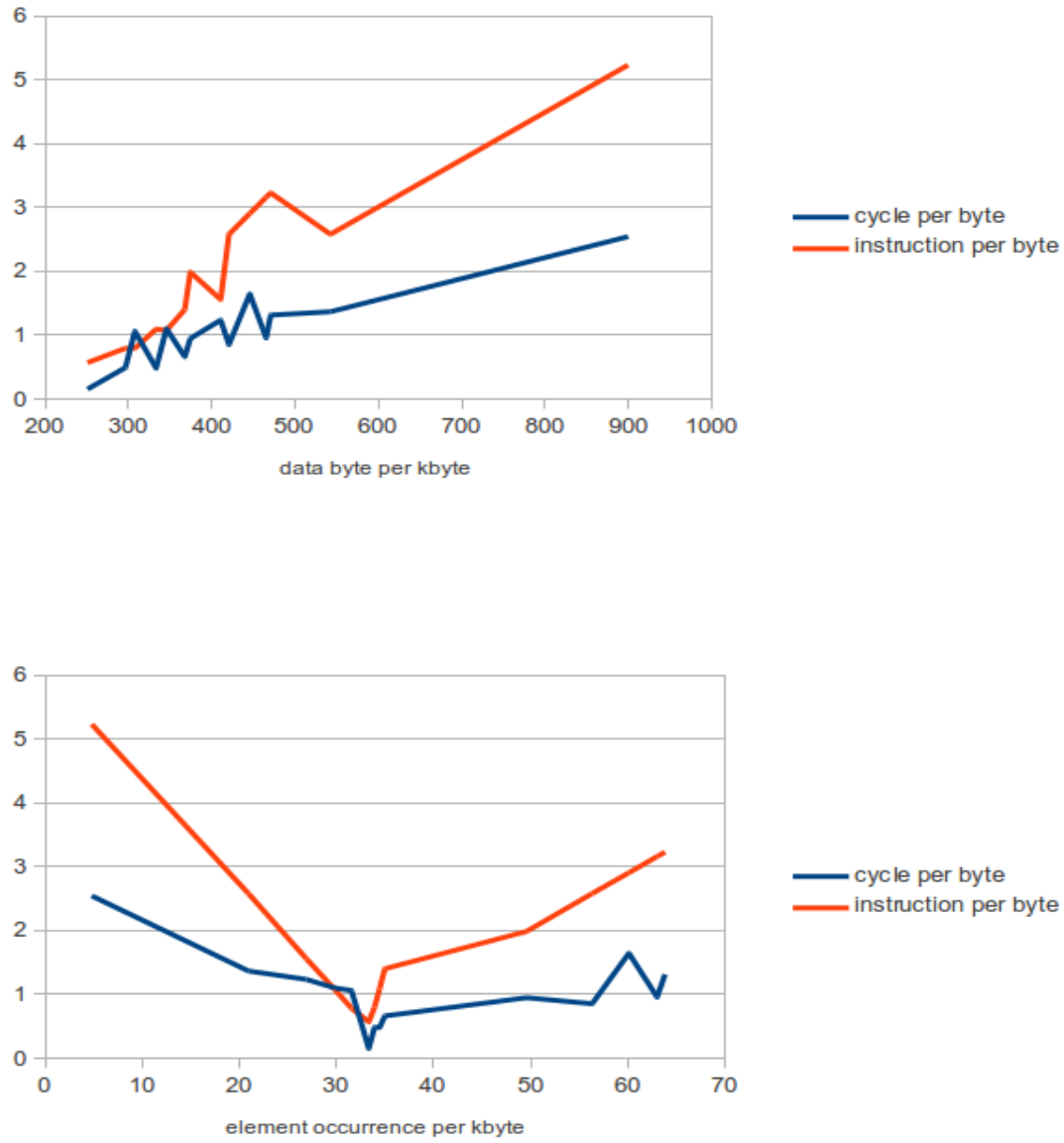

Figure 5.14: Over All Validation Performance

Table 5.2: Performance Against Xerces

element per Kbyte	parallel validation(cycle/byte)	Xerces(cycle/byte)
8.4360001279	3.1309240841	213.265894733
13.6562448835	2.3989508048	171.294562504
21.0198433557	3.5931549873	120.109194821
25.72279118	3.838526008	91.2817089821
29.0875104267	3.7435517285	71.4113897988
31.0111394734	4.8541471414	57.2968200868
33.2440301963	5.2872566644	40.2126389571
36.519440708	4.0315018027	77.0690224924
42.752459889	5.1375744752	147.899139108
47.2608134928	5.2561934806	199.522025888
52.8446273297	5.7556525993	267.437901581
55.8190281212	6.2307075233	295.704017855
57.6546050727	7.2438551494	314.967140622
58.7472846131	7.3811013552	332.991834564

by calculating the difference of the performance of Xerces with the validation and without the validation.

As we can see from the table, of all the test results, Xerces needs at least more than 40 cycle-per-byte to validate the instance file, while our implementation only need 5 cycle-per-byte, there is a 7 times difference between our implementation and Xerces. Furthermore, for some case, there will be an approximately 60 times speed up against Xerces.

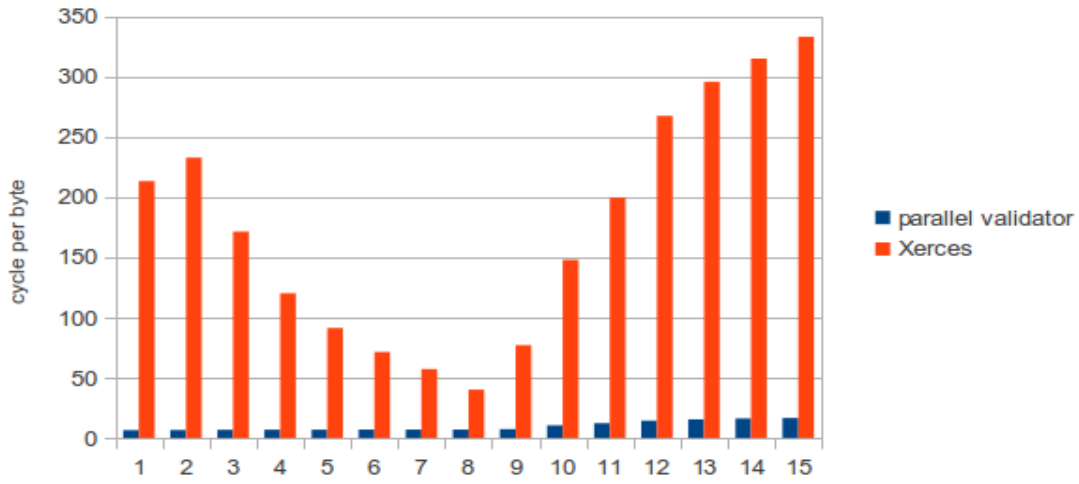

Figure 5.15: Performance Against Xerces

Performance Against icXML

Table 5.3: Performance Against icXML

element per Kbyte	parallel validation(cycle/byte)	icXML validation(cycle/byte)	icXML
8.4360001279	3.1309240841	214.6861036	232.258652352
13.6562448835	2.3989508048	185.891746442	207.337024897
21.0198433557	3.5931549873	123.7482789	151.895853424
25.72279118	3.838526008	88.9758646706	122.227056428
29.0875104267	3.7435517285	62.9855909601	99.7191369029
31.0111394734	4.8541471414	49.1089104569	87.7510078353
33.2440301963	5.2872566644	30.1368451346	70.9301644372
36.519440708	4.0315018027	61.2728551829	103.969180588
42.752459889	5.1375744752	118.93790525	164.24458079
47.2608134928	5.2561934806	152.678688094	201.388552872
52.8446273297	5.7556525993	211.862372468	262.476126648
55.8190281212	6.2307075233	229.996432294	282.466941365
57.6546050727	7.2438551494	255.893780412	309.38315187
58.7472846131	7.3811013552	263.846181264	319.365940862

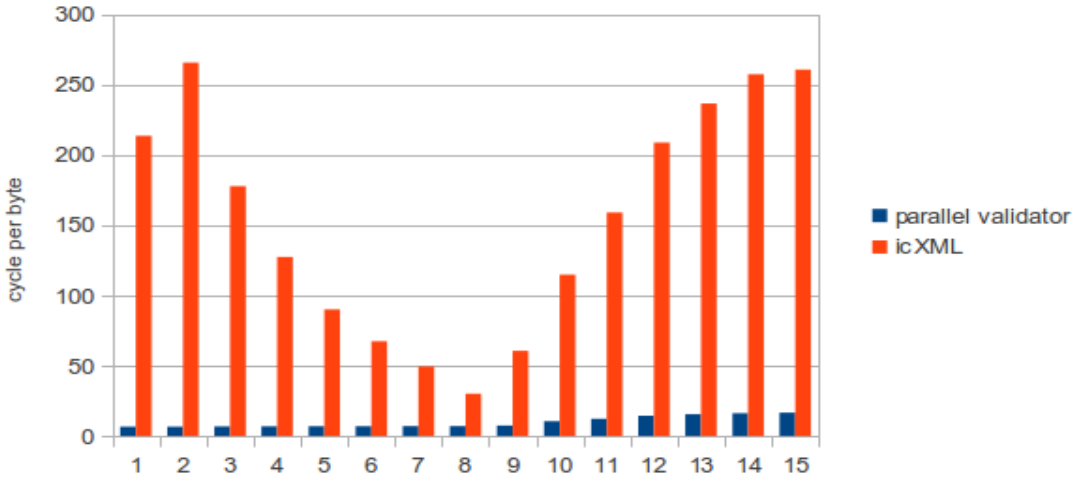

Figure 5.16: Performance Against icXML

As shown in Figure 5.16, as the element occurrences per Kbyte increases, the overhead of the validation of icXML varies. The performance of the validation of icXML is tested by calculating the difference of the performance of icXML with and without the validation.

As shown in the table, even icXML shows much improvement in most of the cases against Xerces, there is still at least 5 time difference between our implementation and icXML.

Performance Against Other Validation Methods

One of the related work by Margaret [13, 15, 17] shows that the best performance of validating is from 21.45 to 43.08 cycle per byte on their benchmarks. While our work shows that the performance of validating including symbol table processing is from 8.40 to 27.57 cycle per byte on our benchmarks.

Branch Miss Rate Analysis

Branch misprediction occurs when CPU mispredicts the next instruction to process in branch prediction, which is aimed at speeding up execution. Branch miss rate is the percentage of the number of branch misprediction of the total branch prediction. While the program is being executed, there are conditional branches. CPU has a branch predictor that tries to guess which way a branch will go before this is known for sure. When the CPU predicts the branch wrong, there will be more over head to re-execute the instructions in the right branch. So if the branch miss rate is lower, more speed-up will be gain.

The data gathering module has much more branches than the other modules, so in this section, the relation between the length of the data segment and the branch miss rate of the data gathering module is analyzed.

Figure 5.17 shows that, as the length of the data segment increases from 1 Block_Size to 20 Block_Size, the branch miss rate dropped in every test cases, which will increase the performance, as shown in Figure 5.18.

From the result, we can tell that the branch miss rate decrease as the size of the segment increase from 1 block to 8 blocks then to 40 blocks. However, the performance difference is not very much between size of 8 blocks and size of 40 blocks cases, so the segment of 8 blocks is good enough.

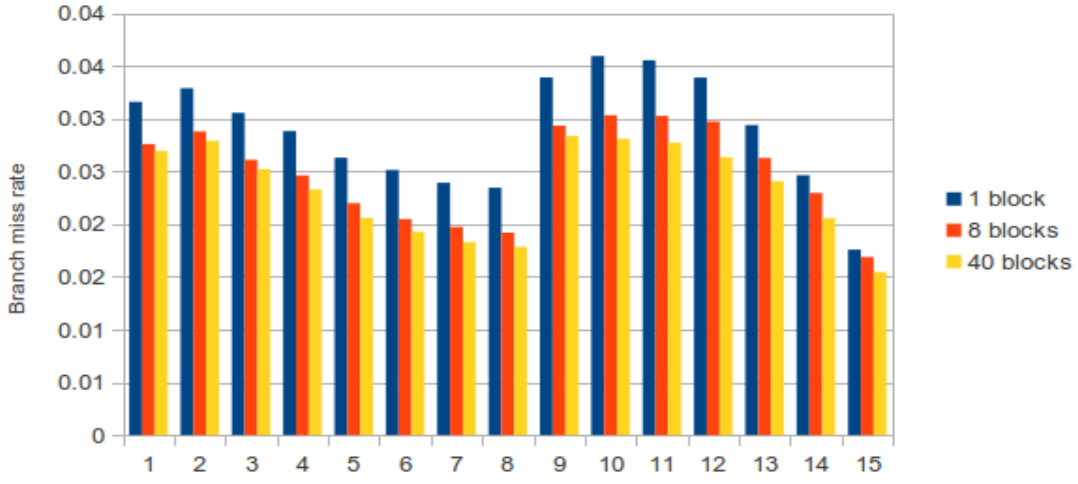

Figure 5.17: Branch Miss Rate Analysis

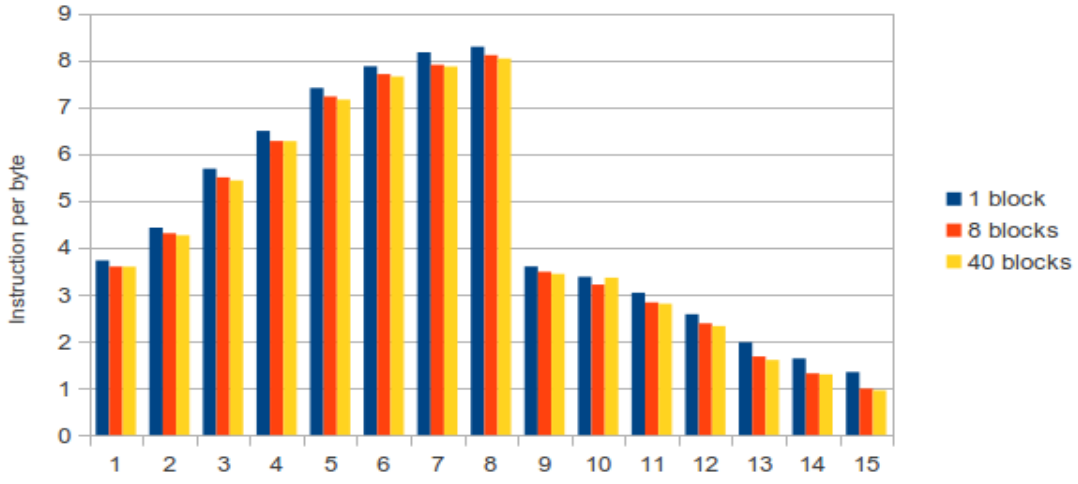

Figure 5.18: Branch Miss Rate Analysis 2

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, a new algorithm of XML validation based on parallel bit stream technology is proposed. Instead of the traditional byte at a time validation method, this algorithm converts the content model and content data into parallel bit streams, then validate them parallel with bitwise operations. This algorithm takes advantage of the regular expression to bitwise operations technology, to make both content model and content data validation into pattern matching problems.

As demonstrated, the performance of the algorithm can achieve at least 5 times speed-ups over the traditional XML parser Xerces. The speed-up can be much more as the characteristic of the instance file changed.

6.2 Future Work

One of the future work of this algorithm will be to fulfill more modules of XML validation, such as attribute validation. The future work also includes to implement a schema/DTD parser to translate the grammars into regular expressions and then into bitwise operations automatically.

The datatype validation module needs to be improved to process the data types and value constraining facets which can not be represented by regular expression easily. The content model validation module also needs to be improved to be able to process "all" content model item.

One of the features that the system could not handle by now is the content model rule "all". The "all" element allows the child elements to appear in any order and each child element can occur zero or one time. The reason that this feature is not supported by the system is that the "all" rule can not be represented by one sample regular expression. But there are alternative methods. One of them is also by using regular expressions, but by using multiple of them. For each child element, one regular expression is needed for validating that there is at least one instance of this child element type if needed, another regular expression will make sure that there is no more than one occurrence.

Another future work is to apply this algorithm on existing XML parsers, such as Xerces and icXML, to improve the performance of those parsers on XML validation.

Appendix A

Validation Problem #1

This schema has named has one data type, and four nontrivial content models.

A.1 Schema Listing

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="records">
   <xs:complexType>
   <xs:sequence>
      <xs:element name="shiporder" maxOccurs="unbounded">
         <xs:complexType>
         <xs:sequence>
            <xs:element name="orderperson" type="xs:string"/>
            <xs:element name="shipto">
               <xs:complexType>
               <xs:sequence>
                  <xs:element name="name" type="xs:string"/>
                  <xs:element name="address" type="xs:string"/>
                  <xs:element name="city" type="xs:string"/>
                  <xs:element name="country" type="xs:string"/>
               </xs:sequence>
```

```
               </xs:complexType>
            </xs:element>
            <xs:element name="item" maxOccurs="unbounded">
               <xs:complexType>
               <xs:sequence>
                  <xs:element name="title" type="xs:string"/>
                  <xs:element name="note" type="xs:string"
                   minOccurs="0"/>
                  <xs:element name="quantity" type="
                   xs:positiveInteger"/>
                  <xs:element name="price" type="doublelist"/>
               </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
         <xs:attribute name="orderid" type="xs:string"
          use="required"/>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
   </xs:complexType>
</xs:element>
</xs:schema>

<xs:simpleType name="doublelist">
   <xs:list itemType="xs:double"/>
</xs:simpleType>
```

records	1
shiporder	2
orderperson	3
shipto	4
name	5
address	6
city	7
country	8
item	9
title	10
note	11
quantity	12
price	13

A.2 Element Names and GIDs

A.3 Regular Expressions for Content Models

The content model of four element types which have child elements needs to be validated. GID "0" is used to represent end tag of each element.

The element type "record" has a sequence of "shiporder" elements, the maximum number of the occurrences is unbounded, so the regular expression of the content model of the element type "record" is:

```
1(2)*0
```

The element type "shiporder" has a sequence of the group of the element "orderperson", "shipto" and "item", they must be in order, and the maximum number of the occurrences of element type "shipto" and "item" is unbounded, so the regular expression of the content model of the element type "shiporder" is :

```
2((3)(4)+(9)+)0
```

The element type "shipto" has a sequence of the group of the element "name", "address", "city" and "country", they must occur only once and in order, so the regular expression of the content model of the element type "shipto" is :

```
4((5)(6)(7)(8))0
```

The element type “item” has a sequence of the group of the element “title”, “note”, “quantity” and “price”, which must occur in order. While the “note” element can occur zero or one time. The regular expression of the the content model of the element type “item” is:

```
9((a)(b)?(c)(d))0
```

Here the Character “a” to “d” indicate GID “10” to “13”.

A.4 Regular Expressions for Data Types

There is only one data type in this schema need to be validated, which is from the element type “price”, the data type is “doublelist”, which is a list of the built-in data type “double” in XML Schema. The character “#” is used as the splitter.

The regular expression of the data type “doublelist” is:

```
(^|#)[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?([ ][-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?)*#
```

Appendix B

Validation Problem #2

This schema has four nontrivial content models, and four data types.

B.1 Schema Listing

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="records">
   <xs:complexType>
   <xs:sequence>
      <xs:element ref="purchaseOrder" minOccurs="0"
       maxOccurs="unbounded"/>
   </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="description">
   <xs:complexType>
   <xs:sequence>
      <xs:element name="producer" type="xs:string"/>
      <xs:element name="distributor" type="xs:string"/>
```

```
      <xs:element name="retailer" type="xs:string"/>
      <xs:element name="phone" type="phone_number"
       maxOccurs="unbounded"/>
      <xs:element name="p" type="doublelist"
       maxOccurs="unbounded"/>
   </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="payment">
   <xs:complexType>
   <xs:choice>
      <xs:element name="creditCard" type="CC"/>
      <xs:element name="debitCard" type="DC"/>
   </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="purchaseOrder">
   <xs:complexType>
   <xs:sequence>
      <xs:element ref="description" minOccurs="0"/>
      <xs:element ref="payment" minOccurs="0"
       maxOccurs="unbounded"/>
   </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:simpleType name="phone_number">
   <xs:restriction base="xs:string">
      <xs:pattern value="\d{3}-\d{3}-\d{4}"/>
   </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="CC">
   <xs:restriction base="xs:string">
      <xs:pattern value="\d{16}"/>
   </xs:restriction>
</xs:simpleType>

<xs:simpleType name="DC">
   <xs:restriction base="xs:string">
      <xs:pattern value="\d{13}"/>
   </xs:restriction>
</xs:simpleType>

<xs:simpleType name="doublelist">
   <xs:list itemType="xs:double"/>
</xs:simpleType>

</xs:schema>
```

B.2 Element Names and GIDs

records	1
description	2
producer	3
distributor	4
retailer	5
phone	6
p	7
payment	8
creditCard	9
debitCard	10
purchaseOrder	11

B.3 Regular Expressions for Content Models

There are four element types which has child element, which are "records", "description", "purchaseOrder" and "payment". Here the character "a" and "b" indicate GID "10" and "11".

The element type "records" has a sequence of element of type "purchaseOrder", the maximum number of the occurrences is unbounded, so the regular expression for the content model of element type "records" is:

```
1(b)+0
```

The element type "description" has a sequence of the group of the element type "producer", "distributor", "retailer", "phone" and "p". The maximum number of occurrences of the element type "phone" and "p" are unbounded. So the regular expression for the content model of element type "description" is:

```
2((3)(4)(5)(6)+(7)+)0
```

The element type "payment" has a choice between the two element type "creditCard" and "debitCard", so the regular expression for the content model of element type "payment" is:

```
8((9)|(a))0
```

The element type "purchaseOrder" has a sequence of the group of element type "description" and "payment", in which the maximum number of the occurrences of the element "payment" is unbounded. The regular expression for the content model of element type "purchaseOrder" is:

```
b((2)(8)+)0
```

B.4 Regular Expressions for Data Types

There are four data type definitions in the schema, which are of element type "phone", "p", "creditCard" and "debitCard". The character "#" is used as the splitter.

The data type of "phone" is defined as a patterned string, the regular expression of the data type "phone_number" is:

```
(^|#)[1-9]{3}-[1-9]{3}-[1-9]{4}#
```

The data type of "p" is "doublelist", which is a list of built-in type "double" in XML Schema. The regular expression of the data type "doublelist" is:

```
(^|#)[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?([ ][-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?)*#
```

The data type of "creditCard" is a patterned string, which has a length of 16. The regular expression of the data type "CC" is:

```
(^|#)[1-9]{16}#
```

The data type of "debitCard" is a patterned string, which has a length of 13. The regular expression of the data type "CC" is:

```
(^|#)[1-9]{13}#
```

Appendix C

Regular Expressions of Built-in XML Schema Datatypes

C.1 string

The string datatype represents character strings.

C.2 normalizedString

The normalizedString represents white space normalized strings. It is the set of strings that do not contain the carriage return, line feed, nor tab characters.

The regular expression of normalizedString type is :

```
[^\xD\xA\x9]+
```

C.3 boolean

The boolean type can have the following legal literals {true, false, 1, 0 }.

The regular expression of boolean type is:

```
true|false|1|0
```

C.4 decimal

The decimal is a finite-length sequence of digits, separated by a period as a decimal. The "+" sign is prohibited, the decimal point is required.

The regular expression of decimal type is:

```
[-]?[0-9]+\.[0-9]*
```

C.5 integer

The regular expression of integer type is:

```
[-]?[0-9]+
```

C.6 nonPositiveInteger

The regular expression of nonPositiveInteger type is:

```
0\|-[0-9]+
```

C.7 negativeInteger

The regular expression of negativeInteger type is:

```
-[0-9]+
```

C.8 long

The long type is derived from integer by setting the value of maxlInclusive to be 9223372036854775807 and minInclusive to be -9223372036854775808.

C.9 int

The int type is derived from long by setting the value of maxlInclusive to be 2147483647 and minInclusive to be -2147483648.

C.10 short

The short type is derived from int by setting the value of maxlInclusive to be 32767 and minInclusive to be -32768.

C.11 byte

The byte type is derived from short by setting the value of maxlInclusive to be 127 and minInclusive to be -128.

C.12 nonNegativeInteger

The regular expression of nonNegativeInteger type is:

```
[+-](0|[+]?[0-9]+)
```

C.13 unsignedLong

The unsignedLong is derived from nonNegativeInteger by setting the value of maxlInclusive to be 18446744073709551615.

C.14 unsignedInt

The unsignedInt is derived from unsignedLong by setting the value of maxlInclusive to be 4294967295.

C.15 unsignedShort

The unsignedShort type is derived from unsignedInt by setting the value of maxlInclusive to be 65535.

C.16 unsignedByte

The unsignedByte type is derived from unsignedShort by setting the value of maxlInclusive to be 255.

C.17 positiveInteger

The positiveInteger type is derived from nonNegativeInteger by setting the value of minInclusive to be 1.

C.18 float

The value of float consists of the values m * 2ˆe, where m is an integer whose absolute value is less than 2ˆ24, and e is an integer between -149 and 104.Float also includes not-a-number value, which can be represented as INF, -INF or NaN.

The regular expression of float type is:

```
[-+]?[0-9]*[\.]?[0-9]+([eE][-+][0-9]+)?
```

C.19 double

The double type is similar as the float type, except the m is less than 2ˆ53, and e is between -1075 and 970.

The regular expression of double type is:

```
[-+]?[0-9]*[\.]?[0-9]+([eE][-+][0-9]+)?
```

C.20 duration

The duration data type is used to specify a time passed, which is specified in the form of "PnYnMnDTnHnMnS" where "P" and "T" are required. The regular expression of duration type is:

```
[-+]P([0-9]+Y)?([0-9]+M)?([0-9]+D)?T([0-9]+H)?([0-9]+M)?
([0-9]+(\.[0-9]+)?+S)?
```

C.21 dateTime

DataTime values include integer-valued year, month, day, hour and minute properties, a decimal-valued second property, a boolean timezoned property, and also one decimal-valued timeOnTimeLine property.

The regular expression of dateTime type is:

```
(-)?[0-9]{4}[0-9]*-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]+
\.?[0-9]*[+-][0-9]{2}:[0-9]{2}Z?
```

C.22 time

The time type represents an instant of time of a day. The regular expression of time type is:

```
[0-9]{2}:[0-9]{2}(-[0-9]{2}:[0-9]{2})?
```

C.23 date

The value space of date consists of top-open intervals of exactly on day in length on the timelines of dateTime. It includes year, month and day properties and an optional timezone-valued timezone property.

The regular expression of date type is:

```
[0-9]{4}[0-9]*-[0-9]{2}-[0-9]{2}(-[0-9]{2}:[0-9]{2})?
```

C.24 gYearMonth

The gYearMonth type represents a gregorian month in a gregorian year. The regular expression of gYearMonth type is:

```
(-)?[0-9]{4}-[0-9]{2}
```

C.25 gYear

The gYear type represents a gregorian year. The regular expression of gYear type is:

```
(-)?[0-9]{4}
```

C.26 gMonthDay

The gMonthDay type represents a gregorian date of a year. The regular expression of gMonthDay type is:

```
[0-9]{2}-[0-9]{2}
```

C.27 gDay

The gDay type represents a gregorian day. The regular expression of gDay type is:

```
[0-9]{2}
```

C.28 gMonth

The gMonth type represents a gregorian month. The regular expression of gMonth type is:

```
[0-9]{2}
```

C.29 hexBinary

The hexBinary type represents hex-encoded binary data. The regular expression of hexBinary type is:

```
[0-9A-F]+
```

C.30 base64Binary

The base64Binary type represents Base64-encoded binary data.

C.31 anyURI

The anyURI type represents a Uniform Resource Identifier Reference. A URI is a sequence of characters, used to identify a name of a web resource. The regular expression of anyURI type is:

```
(http|ftp|mailto|crid|file):[a-zA-Z0-9._]*
```

C.32 QName

The QName type represents XML qualified names.

C.33 NOTATION

The NOTATION type represents the NOTATION attribute type.

Bibliography

[1] Document type definition. http://en.wikipedia.org/wiki/Document_type_definition.

[2] Dtd and regular expression. http://www.dcs.bbk.ac.uk/~ptw/teaching/dtd-new/notes.html.

[3] Extensible markup language (xml) 1.1. http://www.w3.org/TR/2004/REC-xml11-20040204/#NT-doctypedecl.

[4] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[5] Regular expression. http://en.wikipedia.org/wiki/Regular_expression.

[6] Xml schema part 1: Structures second edition. http://www.w3.org/TR/xmlschema-1/.

[7] Xml schema part 2: Datatypes second edition. http://www.w3.org/TR/xmlschema-2/.

[8] Robert D. Cameron, Ehsan Amiri, Kenneth S. Herdy, Dan Lin, Thomas C. Shermer, and Fred Popowich. Parallel scanning with bitstream addition: An xml case study. In *Euro-Par (2)*, volume 6853 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2011.

[9] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. High performance xml parsing using parallel bit stream technology. In *CASCON*, page 17. IBM, 2008.

[10] Robert D. Cameron and Dan Lin. Architectural support for swar text processing with parallel bit streams: the inductive doubling principle. In *ASPLOS*, pages 337–348. ACM, 2009.

[11] Kenneth Chiu and Wei Lu. A compiler-based approach to schema-specific xml parsing, 2004.

[12] Buhwan Jeong, Jungyub Woo, Boonserm Kulvatunyou, and Hyunbo Cho. Jess-based web interface for xml document validation. *Expert Syst. Appl.*, 36:683–689, 2009.

[13] Margaret G. Kostoula, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. Xml screamer: an integrated approach to high performance xml parsing, validation and deserialization. In *WWW '06 Proceedings of the 15th international conference on World Wide Web*, pages 93–102, 2008.

[14] Welf M. Lowe, Markus L. Noga, and Thilo S. Gaul. Foundations of fast communication via xml. *Annals of Software Engineering*, 13:357–379, 2002.

[15] Morris Matsa, Eric Perkins, Abraham Heifets, Margaret G. Kostoulas, Daniel Silva, Noah Mendelsohn, and Michelle Leger. A high-performance interpretive approach to schema-directed parsing. In *WWW '07 Proceedings of the 16th international conference on World Wide Web*, pages 1093–1114, 2007.

[16] E. Perkins, M. Matsa, M. G. Kostoulas, A. Heifets, and N. Mendelsohn. Generation of efficient parsers through direct compilation of xml schema grammars. *IBM Systems Journal*, 45:225, 2006.

[17] Eric Perkins, Morris Matsa, Margaret Gaitatzes Kostoulas, Abraham Heifets, and Noah Mendelsohn. Generation of efficient parsers through direct compilation of xml schema grammars. *IBM Systems Journal*, 45:225–244, 2006.

[18] Henry S. Thompson and Richard Tobin. Using finite state automata to implement w3c xml schema content model validation and restriction checking. In *XML Europe*, 2003.

[19] Wei Zhang. High-performance xml parsing and validation with permutation phrase grammar parsers. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 286–294, 2008.

[20] Wei Zhang. *Efficient Xml Stream Processing And Searching.* PhD thesis, The Florida State University, 2012.