EMBEDDING PARALLEL BIT STREAM TECHNOLOGY INTO EXPAT

by

Qiang Zhang

B.Sc., Vancouver Island University, 2007

a Project submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the School
of
Computing Science

APPROVAL

Name:	Qiang Zhang
Degree:	Master of Science
Title of Project:	Embedding Parallel Bit Stream Technology into Expat

Examining Committee: Dr. Tamara Smyth
Chair

Dr. Rob Cameron, Senior Supervisor

Dr. Anoop Sarkar, Supervisor

Dr. Thomas Shermer, SFU Examiner

Date Approved: June 1, 2010

Abstract

Parallel bit stream technology is a novel approach to interpret byte stream data and exploit data level parallelism by employing single-instruction multiple-data (SIMD) instructions. Parabix, an XML parser embedded with parallel bit stream technology, performs much better than traditional XML parsers that process XML documents in byte-at-a-time fashion. The project attempts to enhance the performance of Expat, an traditional XML parser, by embedding parallel bit stream technology into Expat. Most byte-at-a-time loops are identified and then replaced with bit stream operations. A performance case study is conducted by comparing the performance result between the original Expat and the parallel bit stream version.

Keywords: XML, SIMD, Expat, Parabix, Byte-at-a-time, Parallel bit stream technology.

Acknowledgments

I would like to take this opportunity to express my sincere gratitude to all these people giving me help while my graduate study. I cannot achieve so much without their help.

First, I would like to thank my senior supervisor Dr. Robert D. Cameron. There is no doubt that my project could not be done without him. He inspires the project with his own work. His experience, advice and encouragement keep me moving forward. As a project student, he even offered me partial financial support. His easygoing personality makes him the nicest supervisor.

Second, I would like to thank my project committee, Dr. Tamara Smyth, Dr. Anoop Sarkar, and Dr. Thomas Shermer for their time and assistance.

Third, I would also like to extend my gratitude to Dan Lin, another graduate student of Dr. Cameron. I gained much insight about Parabix from the continuous discussion with her.

Finally, I want to thank for my parents. So many years, they have stood on my side and believed in me with all their love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Extensible Markup Language (XML) is a simple, human-readable textual data format derived from SGML (ISO 8897). The initial XML draft is released in November 1996 [3]. There are two current versions of XML: XML1.0 defined in 1998 [5], XML1.1 defined in 2004 [4]. XML1.1 is a minor update to XML1.0 with a few changes for character sets and encoding.

SGML's initial design goal is to handle large-scale electronic publishing. XML as a subset of SGML is specifically made for the simplicity and generality of using it over the Internet [5]. Also, a great number of XML-base languages is developed, including GML and XHTML. XML-based format has been used in most office-productivity software like Microsoft Office, Open Office, and Apple's iWork [11].

Document-oriented and data-oriented XML are popularly used to distinguish among XML documents. Document-oriented XML are used for publishing information, whereas data-oriented XML are used for exchanging database records [21]. Data-oriented XML have a higher markup density[1] than document-oriented XML [16].

XML documents cannot be directly interpreted by machine, an XML parser is need to extract information from XML documents. XML parsers can be distinguished in two categories: tree-based and stream-based. Tree-based parsers build a tree representation of XML documents. Users can interact with XML documents by navigating through the node of the tree. Stream-based parsers are event-driven stream-oriented parses. Users interact with XML documents by registering callback functions for events. Usually, stream-based

[1]Markup density is the percentage of the number of markup item in the total file size.

parsers consume less memory since they do not need to build the tree representation.

Expat as a stream-based parser has gain its popularity in open-source projects because of its light weight overhead. Expat like the most traditional parsers processes byte-at-a-time. Depending on what the current byte is, it starts up appropriate callback functions. Parabix is a XML parser with a new approach. Instead of parsing byte-at-a-time, Parabix first transfers byte stream into eight parallel bit streams. Each bit stream is comprised of one bit of each byte in the byte stream. Then single-instruction multi-data (SIMD) instructions are applied to the bit streams. 32, 64 or 128 bytes can be processed at a time with bit stream operations [2], depending on the size of the register.

One case study shows that Parabix run at least 2 times faster than Expat [16]. This project focuses on whether it is possible to improve Expat performance with parallel bit stream technology so that the current open-source projects can enjoy the improvement without modifying their existing code base.

The rest of the project report is constructed in the following way. Chapter 2 briefly introduces XML production rules (XML syntax). Chapter 3 explains the structure and usage of Expat and Parabix. The strategy used to embed parallel bit stream technology into Expat is well explained in Chapter 4. Chapter 5 outlines the performance study. The last chapter concludes the whole report.

Chapter 2

XML Basics

An XML document encoded in Unicode has two major part: document content and document type definition (DTD). Document content is comprised by text item and markup item. A set of XML production rules defined in [5] is used to produce markup item, including start tag, end tag, empty tag, reference, comments, and CDATA. All markup items except reference should start with a left angle bracket and ends with a right angle bracket. Start tag, end tag, and empty tag should have exact one tag name, and 0 or more attribute name/value pairs except end tag. Reference starts with an ampersand and ends with a semicolon. Reference is either character reference or entity reference. Character reference has decimal and hex-decimal representation. The following is simple examples of markup item:

- Start tag: $< startTagName\ attributeName =$ "$attributeValue$" $anotherAttributeName =$ "$anotherAttributealue$" $>$
- Empty tag: $< emptyTagName/ >$
- End tag: $< /endTagName >$
- Entity reference: $\&entityReference;$
- Character reference in decimal representation: $\&\#67;$
- Character reference in hex-decimal representation: $\&\#x1ab;$
- Comments: $<! - -comment - - >$

- CDATA: $<![CDATA[cddata]]>$

An XML document is well-formed if it meets all production rules. A valid XML document has to be well-formed. XML parsers has to report a fatal error and stops parsing if the XML document is not valid. Document type definition (DTD) defines the format of markup item. XML parsers can be categorized as validating parsers or non-validating parsers. Validating parsers not only check for the well-formedness but also report violation of DTD at user option. Non-validating parsers only check for the well-formedness. Expat is a non-validating parser, whereas Parabix is a validating parser.

The XML knowledge introduced here is really limited and is specifically selected for this project report. The XML1.0 specification is way more complex than what is introduced here. If there are questions, please refer to the XML1.0 specification at [5].

Chapter 3

Parsers

3.1 Expat

3.1.1 What is Expat?

Expat is a library, written in C, for parsing XML documents. It was first created by James Clark in 1998, then the project was handed over to a group led by Clark Cooper and Fred Drake [8]. The latest version, Expat-2.0.1, is released at 5, June, 2007 [27]. As one of the earliest open-source XML parsers, also because of its light-weight fast parsing, Expat is very popular in open-source projects like Mozilla, OpenOffice, and Apache [1][6][23]. Also, a number of programming languages (PHP, Python, Perl, Ruby, Objective-C, etc) binds Expat as one of their XML parsers through wrapper functions [27].

3.1.2 How to use Expat?

Expat is a stream-oriented event-driven XML parser. It closely follows the requirement of a non-validating parser specified by XML1.0 specification. The violation of the well-formedness constraints is reported, but the constraints defined by document type declarations (DTD) are ignored by the parser.

Expat allows user to interact with an XML document via callback functions (function pointers) that are written by the user. First, the user declares a parser and registers callback functions. Second, a buffer with XML data is fed into the parser. The buffer can either contain the whole XML document, or a part of the XML document in the case of the XML document that is too large to fit into the memory at once. Then, Expat examines byte-at-a-time to look for a markup item. The callback functions are called when a markup item is found. The parser keeps processing in this fashion until it reaches the last byte of the buffer.

Expat has a rich number of callback functions and options to handle different aspects of XML like start tags, character data, external entities and so on. The complete set can be found in the expat.h file of the source code [22]. It could be overwhelming to understand them all for those who are new to XML and Expat. Here are four fundamental functions that help the user to do 80% their requests [20].

1. `XML_Parser XML_ParserCreate(const XML_Char* encoding)`

   - A newly created parser is returned to user.
   - US-ASCII, UTF-8, UTF-16 and ISO-8859-1 are the four built-in encodings.

2.
   ```
   XML_SetElementHandler(XML_Parser p,
                         XML_StartElementHandler start,
                         XML_EndElementHandler end)
   ```

   - Set start and end tag callback functions that are written by user. Once Expat processes a full start or end tag, and collects the tag name and the attribute name/value pairs, the start and end tag callback functions are executed.

3.
   ```
   XML_SetCharacterDataHandler(XML_Parser p,
                               XML_CharacterDataHandler charhndl)
   ```

   - Set the character data callback function.
   - Reference and newline are treated as separated character data from the rest. For example, in <charData> firstPart&lt;thirdPart\n[1]forthPart<charData>, the character data callback function is executed 4 times with "firstPart", "&lt;", "thirdPart", "\n", and "forthPart" in this order.

4. `int XML_Parse(XML_Parser p, const char *s, int len, int isFinal)`

[1]A new line character

- Parse a buffer of XML data. The string "s" is an array of "len" bytes, but without a null terminator. The "isFinal" flag is used to determine whether it is the last piece of the XML document. "len" can be any none negative integer, so a large file can be broken down to smaller pieces in the case of limited memory.

3.1.3 How does Expat Perform Compare to other Parsers?

At the time Expat released, there are other open-source XML parsers available for download. What distinguishes Expat from the others? Since performance is one of major goals that Expat struggles for, does Expat achieve its purpose? In [18], Clark Cooper tests Expat against six other Parsers: C-expat, RXP, Java-XP, Java-XML4J, Perl[2], Python[3]. After the six parsers are fed in the exact same XML data, an identical statistical report should be generated with the following information [19]:

- The number of times each element occurs
- The number of one-level-up parents for the element
- The number of one-level-down parents for the element
- The number of a particular attribute for the element
- The number of character data that has at least one non-whitespace characters
- Whether the element is always empty

Below is an example of XML documents and the statistical report in next page.

```
<foo>
<bar>This is a test
</bar>
<bar>
<alpha><bar>Surprise</bar>
</alpha>
<ref id="me" xxx="there"/>
</bar></foo>
```

[2]Perl uses XML::Parser that is built on top of Expat.

[3]Python uses Pyexpat that is built on top of Expat

```
================
foo: 1
   Children:
      bar                        2
================
bar: 3
Had 22 bytes of character data
   Parents:
      alpha                      1
      foo                        2
   Children:
      alpha                      1
      ref                        1
                             =====
                                 2
================
alpha: 1
   Parents:
      bar                        1
   Children:
      bar                        1
================
ref: 1
Always empty
   Parents:
      bar                        1
   Attributes:
      id                         1
      xxx                        1
```

The performance is measured based on the actual time used to generate the statistical report. The measurement is only taken when the parser has loaded in physical memory. Five XML

	REC	chrmed	med	chrbig	big
Size(bytes)	159339	8993821	1264240	3417181	50052472
Markup Density	34%	6%	33%	2%	33%

Table 3.1: File's Characteristics

documents are used as the test cases: REC-xml-19980210.xml and the four expanded version of REC-xml-19980210.xml. chrmed.xml and chrbig.xml are only text contents repeated 8 and 32 times respectively, whereas med.xml and big.xml are the root content repeated 8 and 32 times [18]. Table 3.1 outlines the actual size in bytes and the markup density of each file.

The six parsers are written in three programming languages: C, Java, Scripting. If the coding quality is in the same level, C version should be the fastest; scripting one should be the slowest; Java should be in the middle. So, the question becomes whether the testing result confirms the expectation. Figure 3.1 clearly shows that C version has the absolute advantage over the others for all five files.

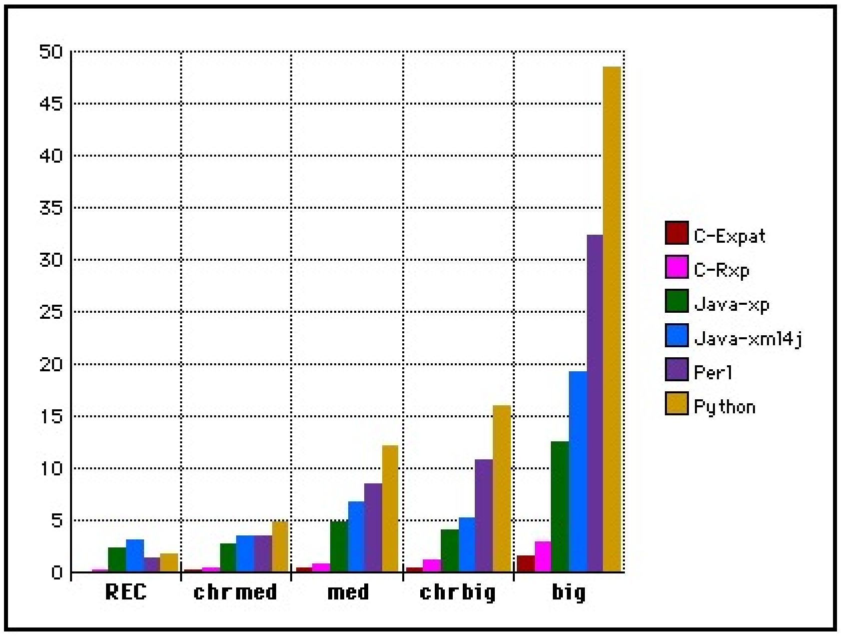

Figure 3.1: Comparison of Six XML Parsers Processing Each Test File [18]

3.2 Parabix

3.2.1 What is Parabix?

Parabix is an open-source XML parser written in C++ by Robert D. Cameron. It dramatically improves parser's performance by employing the SIMD (single-instruction multiple-data) instructions. The SIMD instructions exploit data level parallelism. It is the simplest way of parallelism and has becoming the most common way because hardware support for SIMD is available in modern-day processors like Intel Pentium III, Pentium 4 PCs, and Apple/Motorola G-4 machines [26]. In figure 3.2, SISD (single-instruction single-data) takes exact one instruction and one line of data to produce one output, whereas SIMD takes one instruction and multi-line of data to produce multi-line of result. The main advantage of SIMD is to process multi data at the same time. If one instruction is need to apply to 64 sets of data, the instruction has to be run 64 times in the SISD model, but the instruction needs to be run only once in the SIMD model if the register can contain all data.

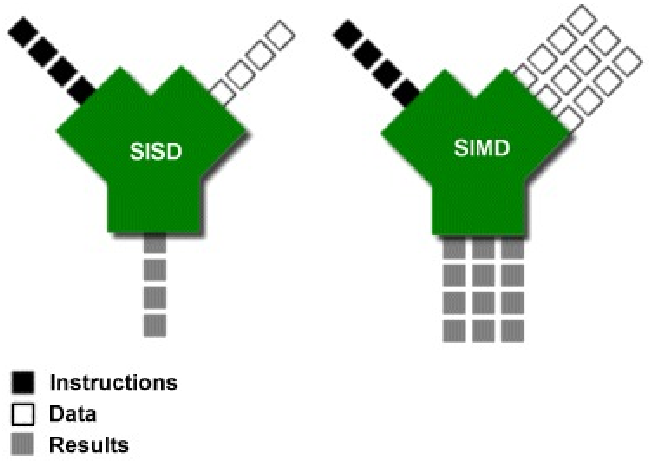

Figure 3.2: SISD VS SIMD [28]

Traditional XML parsers process XML documents in a byte-at-a-time fashion, whereas Parabix employs the SIMD technology to advance as many as 128 bytes with a single instruction [16].

3.2.2 How to use Parabix?

User can make use of Parabix through the Parabix interface that can be found in [17]. The interface can be divided into four categories based on their functionality. The following lists the most representative functions and explanation for each category:

1. `static Parser_Interface * ParserFactory(const char * filename)`

   - Create a Parabix parser object and return it to user.
   - To parse a XML document, all Parabix need is the file. Parabix automatically detects the encoding scheme based on the byte-order mark[4]. If there is no byte-order mark, UTF-8 encoding is implied.

2. `void Parse_prolog()`
   `void Parse_DocumentContent()`

   - Parse the XML documents. Parabix provides validating and non-validating mode for parsing.
   - Parse_prolog() has to be called before Parse_DocumentContent() in the case of DTD existence.

3. `XML_version get_version()`
   `XML_standalone standalone_status()`

   - Get information about the XML documents. For example, XML version, byte-order mark, encoding, etc.

4. `void StartTag_action(unsigned char * item, int lgth)`
   `void Text_action(unsigned char * item, int lgth, bool more)`
   `void Prolog_action(unsigned char * item, int lgth)`

   - Expat allows user to play with the XML documents through callback functions, whereas Parabix enables user interact with the XML documents with action routines.

[4]An unicode character that signals the endianness of the file. If byte-order mark exists, it has to be the first four bytes of the file.

- “item” is pointed at the first character after the opening markup, and “lgth” indicates the length of the markup content that ends at the character before the closing markup.
- User has to write their codes inside the action routines before parsing so that the action routines can take effect.

3.2.3 What is Parabix’s Internal Structure?

Parabix is broke down into 8 modules in figure 3.3. Now let’s define the functionality of each module. The XML interface module separates the XML document content from

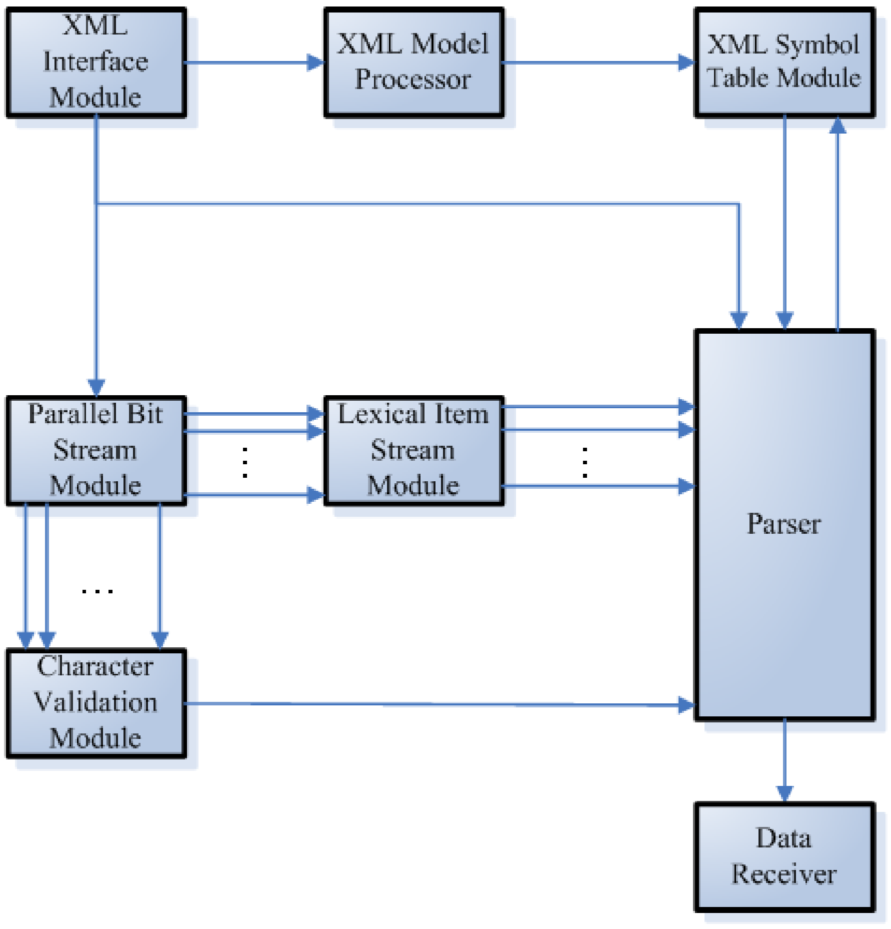

Figure 3.3: Parabix Architecture [16]

the XML declaration and DTD (document type definition). The XML model processor processes the DTD and stores the validation constraints in the XML symbol table module. The XML document content is passed to the parallel bit stream module to produce a set of eight parallel bit streams. The eight parallel bit streams is a new representation of the byte stream. Each bit of eight bits in a byte is extracted and stored in one bit stream. Then the parallel bit stream is passed into the character validation module to detect invalid encoding like incomplete UTF-8 sequence, and illegal characters, like #xFFFE, specified in XML1.0 specification. Also, the parallel bit streams is passed into the lexical item stream module to determine the position of XML markups [16]. Figure 3.4 is an example of the output produced by the lexical item stream module. The LAngle stream marks one at all occurrence of left angle brackets.

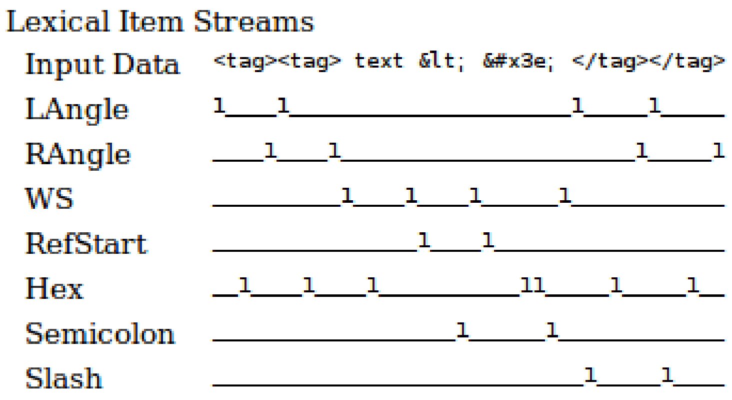

Figure 3.4: Lexical Item Stream [14]

Then the position of all left angle brackets can be calculated from the LAngle stream. The parser takes the lexical item streams along with the original byte stream to process the XML document content.The parser first uses bit scan operations[5] to detect the position of the next Markup item starting character ("<", "&" or "]" for potential of CD data end). Then, it switches to the byte stream to determine what the exact markup item[6] is. The markup item is parsed with maximum use of bit scan operations since they are far more

[5]Bit scan operations take one bit stream and one position as input, and return the position of the occurrence of one in the bit stream after the position from input

[6]The markup item can be a start tag, an end tag, a processing instruction, CD data section, reference or an CD data end.

efficient than byte-at-a-time parsing [16]. Now let's walk through $< startTag > textData < /startTag >$ to show how the parser works [7].

1. The first MarkupStart position 0 is found via bit scan operations in the MarkupStart stream. Update the current position with 0.
2. Use byte tests to determine it is a start tag. Increase the current position by 1 to escape $<$.
3. Found the start tag name length by scanning the NameFollow stream[8]. Parse the start stag name, escape $>$, and update the current position with 10.
4. The second MarkupStart position 18 is found and the current position updates to 18.
5. Determine it is and end tag and increase the current position to 20.
6. Scan the end tag name and parse it. Escape $>$, and update the current position with 29.
7. End of the XML document content and stop.

While the parser processes the content, the XML symbol table module is involved to confirm the content following the validation constraints specified in DTD. In the case of non-validating model, the XML symbol table module is only used for checking the well-formedness of the name [16].

3.2.4 How does Parabix perform?

The evaluation of Parabix is conducted by comparing three C/C++ based event-driven, stream-oriented XML parsers: Parabix, Xerces, and Expat [16]. Since Expat does not verify the validation constraints defined in DTD, the validation feature of Parabix and Xerces are disabled. The driver used in the performance test is collecting a XML document statistical report [16]:

[7] The example is far simper from a real XML document, so the parser is more complicated but the general idea is the same.

[8] The NameFollow stream is one bit stream that have marked one at all occurrence of a set of characters (spaces, left angle brackets, question marks, etc) that indicates it is the end of a name.

File Name	dewiki.xml	jawiki.xml	roads.gml	po.xml	soap.xml
File Size (kB)	66240	7343	11584	76450	2717
Markup Density	0.07	0.13	0.57	0.76	0.87

Table 3.2: XML Document Characteristics [16]

1. The number of occurrence of each type of markup items. For example, the number of start tags, the number of empty tags, and the number of comments.
2. The average length of each type of markup items.

The performance test is experimented on a 2.1 GHz Intel Core 2 Duo processor desktop machine with 2 GB memory in Ubuntu 7.10 [16]. The XML documents used in the experiment is listing in table 3.2. dewiki.xml and jawiki.xml are the document-oriented XML documents. The rest of three are the data-oriented XML documents.

The paper [16] compares the performance based on: instructions completed, processor cycles, conditional branches, branch mis-predictions, and caches misses. Figure 3.5 shows the overall performance. Independent of markup density and files, Parabix performs at least two times faster than Expat, and five times faster than Xerces.

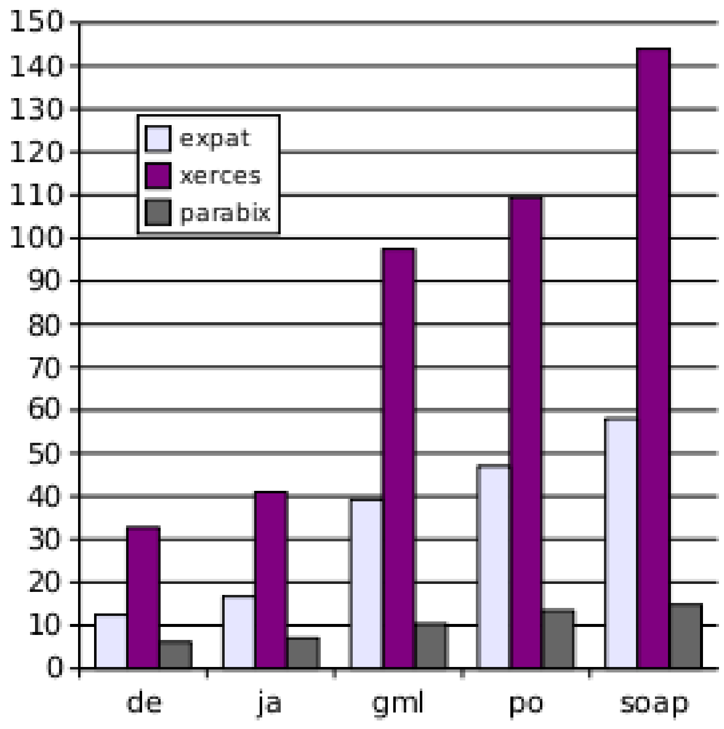

Figure 3.5: CPU Cycles Per Byte [16]

Chapter 4

Adapt Expat to Parallel Bit Stream Technology

Since Expat already gains popularity in open-source community, no open-source project is willing to switch from Expat to Parabix without seeing a speedup in their own project. It is understandable because of two reasons: 1. Expat performs relative well, so that the user is satisfied with the performance. 2. Switching requires modifying their existing code since Parabix and Expat have different interfaces. Modifying the existing code can be time and money consuming. Therefore, the likely solution is to employ Parabix with an Expat interface. Then, the user can enjoy the performance improvement with little changes to their existing code.

4.1 The General Ideal of Expat with Parallel Bit Stream Technology

It is difficult to build an identical Expat interface for Parabix because Expat and Parabix have quite different approaches about their internal design. However, it is possible to bring the parallel bit stream technology into Expat. Expat parses byte-at-a-time with loops. It means Expat advances at most one code unit[1] in byte stream every time, whereas Parabix is

[1]Code unit is the minimal bit combination that can represent a unit of encoded text for processing or interchange. The Unicode Standard uses 8-bit code units in the UTF-8 encoding form, 16-bit code units in the UTF-16 encoding form, and 32-bit code units in the UTF-32 encoding form [9].

able to advance as many as 128 code units at a time with bit scan operations that operate on lexical item streams. Generating the lexical item streams for 100K data do bring about 2 or 3 cycles per byte overhead [16], but the performance gain from advancing many code units at once can be greater than the overhead. Therefore, the idea of adapting the parallel bit stream technology has become substituting all byte-at-a-time loops with bit scan operations. Since XML DTD is a small part of the XML document, bit scan operations are only applied to parsing the document content to meet the minimum change to Expat requirement.

4.2 Generating Lexical Item Streams in Expat

Four modules from Figure 3.3 are need to generate the lexical item streams for a valid byte stream[2]: the XML interface module, the parallel bit stream module, the character validation module, and the lexical item stream module. The XML interface module is still an overkill because many functions are made particularly for the parser module. Therefore, the interface BitStreamScan is created. The BitStreamScan is responsible for three jobs:

1. Gather encoding and endianness of the XML document. This is done by examining the first four bytes of the XML data.

2. Allocate an 100K buffer (src_buffer) for containing the byte stream data.

3. Allocate and initialize an buffer (lexical_buffer) for each lexical item stream [3]. Since each byte in src_buffer is represented by one bit in lexical_buffer, the size of each lexical_buffer is 12.5K (100K/8).

In order to minimize the cost of generating the lexical item streams, the BitStreamScan interface should be declared only once in Expat. After the declaration, src_buffer and lexical_buffer are reused if the byte stream data is more than 100K. The interface declaration is made in the XML_Parse(XML_Parser parser, const char *s, int len, int isFinal) function of Expat. When the function is called the first time, the first four bytes of “s” is the first four bytes of the XML document, so it is used to calculated the encoding and the endianness. Then the BitStreamScan is declared and a flag (bitstream_create_flag) set to 1 to prevent the BitStreamScan from getting declared second time.

[2]A valid byte stream is a byte stream that does not contain illegal XML character and illegal byte sequence

[3]The lexical item streams can be found in bitlex.h [7]

The parallel bit stream module is mainly left unchanged, but the buffer strategy is not compatible with Expat because Parabix takes a file pointer that points to the file containing all XML data, whereas Expat can parse a part of XML data. The following changes is applied to the parallel bit stream module:

1. If the byte stream data from Expat is less than 1024 bytes but no more XML data, parse it anyway. Otherwise, wait for more data until the available XML data is more than 1024 bytes. this is to minimize cycle per byte overhead introduced by generating the lexical item streams.

2. Make sure the last byte in src_buf is a complete byte sequence. For example, the last three bytes of the byte stream is 0x5A, 0xD2, 0x81 in UTF-8 encoding. The 0x81 is not a compete byte sequence because 0xD2 indicates three bytes need to represent the Unicode but only two bytes are available. Therefore, the byte stream ends at 0x5A is copied into src_buf. The detail information about UTF-8, UTF-16 and UTF-32 can be found in [10] [24] [25] respectively.

The character validation module and the lexical item stream module is left unchanged.

4.3 Position the lexical item stream generator in Expat

By following Expat execution in GDB, the four-stage runtime is identified.

1. Create an Expat Parser and initialize options and callback functions.

2. Make a local copy of XML data.

3. Parse document type definitions and trigger the corresponding callback functions.

4. Process the document content and execute the corresponding callback functions.

Intuitively, the forth stage should be the candidate location for generating the lexical item streams. By looking through the source code [12], the function doContent(XML_Parser parser, int startTagLevel, const ENCODING *enc, const char *s, const char *end, const char **nextPtr, XML_Bool haveMore) is found. Algorithm 1 is the pseudocode of the function. The doContent(...) acts as a driver function to loop through the byte stream from “s” to “end”. The “haveMore” flag signals whether the current byte stream is the last

XML data. The XmlContentTok(...) returns a integer value indicating what the current token is and "next" points the first character after the token. Then the parser switches the corresponding token case to process the token, and "s" gets updated with the value in "next".

Algorithm 1 doContent Function Pseudocode

```
doContent(..., const ENCODING *enc, const char *s,
          const char *end, ..., XML_bool haveMore){
   ...
   for( ; ; ){
      int tok = XmlContentTok(enc, s, end, &next);
      switch(tok){
         case XML_TOK_NONE:
            if(haveMore) return XML_ERROR_NONE
            if(start tags one-to-one matches with end tags)
               return XML_ERROR_NONE
            return XML_ERROR
         case XML_TOK_PARTIAL:
            if(haveMore) return XML_ERROR_NONE;
            else         return XML_ERROR;
         case XML_TOK_ENTITY_REF:
            ...
         case XML_TOK_START_TAG_WITH_ATTS:
            ...
         case XML_TOK_END_TAG:
            ...
         case XML_TOK_DATA_CHARS:
            Execute the character data callback function.
         case XML_TOK_COMMENT:
            ...
      }
      s = next;
   }
}
```

This process keeps going until a fatal error is encountered or "end" is reached. The XML_TOK_NONE indicates that the current byte stream is size of zero. An occurrence of a partial markup item[4] is signaled by XML_TOK_PARTIAL. Only a full markup item[5] is processable. In the case of the XML_TOK_PARTIAL, the parser cannot keep processing, so it has to make a copy of the partial markup item and waits for more byte stream. All other tokens are self-explaining. The tokens in the pseudocode is only a subset of tokens from the doContent(...) function.

The lexical item streams are generated for parsing the document content, so it has to be in the doContent(...). Since the for loop iterates over the same data byte stream, it is definitely hurt the performance if the same lexical bit streams calculated repeatably. Therefore, the lexical bit streams should be positioned in the doContent(...) but right before the for loop.

4.4 Apply Bit Stream Operations in Expat

In order to apply bit stream operations, byte-at-a-time loops should be replaced with the bitstream_scan(...) function. The bitstream_scan(item, pos) takes an lexical item stream ("item") and an integer that indicates a position in the byte stream ("pos"), then returns an integer that indicates the byte stream position of the next lexical item starting from the "pos". For example, ... $< tag >< \backslash tag >$ .... Let LAngle represent the left angle bracket bit stream. Let the position for the first angle bracket be 10, then the second one is 15. The bitstream_scan(LAngle, 10) returns the integer 10. The bitstream_scan(LAngle, 11) returns 15.

$$CodeUnitPos = (CurrentBytePos - StartBytePos)/BytesPerCodeUnit \tag{4.1}$$

$$CurrentBytePos = StartBytePos + CodeUnitPos * BytesPerCodeUnit \tag{4.2}$$

The "pos" in the bitstream_scan(...) is a code unit position, but Expat only provides

[4]A partial markup item is a part of a full markup item. For example, $< partialStartTagattribute1 =$ "bbb. It only occurs at the end of the byte stream in a well-formed XML document.

[5]A full markup item is a well-formed markup item that has all XML data from the opening left angle bracket to the ending right angle bracket. For example, $< partialStartTagattribute1 =$ "bbb" $>$.

the byte position. The conversion is made based on the formula 4.1 and 4.2. The code unit base for UTF-8, UTF16 and UTF32 is 1 byte, 2 bytes, and 4 bytes respectively.The Expat-PosToParabixPos(...) and the ParabixPosToExpatPos(...) in algorithm 2 are the macro code for the conversion. The "input_ptr" points the start of the byte stream. The MINBPC(enc) is the code unit base from Expat. The reason of using macro is to shift the conversion cost to the compile time. Otherwise, the performance gain is less than the cost. After using the bitstream_scan(...), the return value has to be less than "buffer_limit_pos" (the total code units in the BitStreamScan buffer). If the value is bigger, it means the BitStreamScan only has a partial markup item at the end of the buffer. Then, the BitStreamScan hands the control back to Expat and asks for more XML data.

Algorithm 2 How Bitstream_scan works!

```
#define ExpatPosToParabixPos(enc, expat_pos, parabix_pos)
   parabix_pos = (expat_pos - input_ptr)/MINBPC(enc);

#define ParabixPosToExpatPos(enc, expat_pos, parabix_pos)
   expat_pos = (input_ptr + parabix_pos * MINBPC(enc));

ExpatPosToParabixPos(enc, ptr, current_pos_parabix);
next_pos_parabix=bitstream_scan(MarkupStart,current_pos_parabix);
if(next_pos_parabix >= global_bitstream->buffer_limit_pos)
   return XML_TOK_PARTIAL;
ParabixPosToExpatPos(enc, ptr, next_pos_parabix);
```

In Expat, most of byte-at-a-time scanning happens in scanning a token. While scanning a token, Expat checks for the well-formedness of the markup item. The goal is to modify the functions scanning the following tokens: start tags, end tags, entity references, character references, comments, CD data, and character data. All these functions are located in xmltok_impl.c [13]. The rest of this section is organized as explanation, the pseudocode of the Expat version, and the pseudocode of the bit scan operation version.

4.4.1 Byte-at-a-time Name Scan VS Bit Stream Name Scan

Expat does not have a specific function for processing XML name. It is directly located in functions involving XML names like scanning start tags, end tags and entity references. The general idea is extracted and written in algorithm 3. Since XML has different character sets for the first character of a name and the rest of the name. Expat checks whether the first character is legal, and checks the rest of characters byte-at-a-time in a loop.

Algorithm 3 Check a Name: Expat Version

```
checkName(...){
   if(at a illegal name start character)
      return XML_TOK_INVALID;
   move to the next byte
   for(;;){
      if(at a legal name character)
         move to the next byte
      else
         return XML_TOK_INVALID;
      if(at a white space)
         break;
   }
}
```

The CheckNameMarco(...) in algorithm 4 has a four-stage schema to verify the well-formedness of a name:

1. Find the "name_end" in the code unit position with the "NameFollow" bit stream. The "NameFollow" have all non-name character (like white space) marked.

2. Scan to the first "next_check" with the "NameStartCheck". The "NameStartCheck" have all non-ASCII-start-name character marked. Since most names are composed of ASCII characters, the "name_end" is reached in the most cases. If so, the CheckNameMarco(...) is done.

3. Scan to the first "next_check" with the "NameCharCheck". The "NameCharCheck" have all non-ASCII-name character marked. If the "name_end" is reached, a valid name found.

4. The CheckNameMarco(...) most unlikely gets in this stage. The current byte either a non-ASCII character or an illegal name character. If the later case is found, issue a fatal error and stop parsing. Otherwise, update to the next code unit point and scan for the next "NameCharCheck" until reaching the "name_end" or finding a illegal name character.

Algorithm 4 CheckNameMarco: Bit Stream Version

```
CheckNameMacro(enc, ptr, end, nextTokPtr){
   ExpatPosToParabixPos(enc, ptr, name_start);
   name_end = bitstream_scan(NameFollow, name_start);
   if(name_end >= global_bitstream->buffer_limit_pos)
      return XML_TOK_PARTIAL;
   next_check = bitstream_scan(NameStartCheck, name_start);
   if(next_check == name_start)
      if( a illegal start name character)
         return XML_TOK_INVAILID;
      next_check = next_check + 1;
   if(next_check < name_end){
      next_check = bitstream_scan(NameCharCheck, next_check);
      while(next_check != name_end){
          if( a illegal name character )
             return XML_TOK_INVALID;
          if( a valid non-ASCII character )
             next_check = next_check + 1;
          else
            return XML_TOK_INVALID;
          next_check = bitstream_scan(NameCharCheck, next_check);
      }
   }
   ParabixPosToExpatPos(enc, ptr, name_end);
}
```

The CheckNameMarco(...) is supposed to be more efficient than Expat version. First, a XML name rarely has a non-ASCII character. Second, a XML name mostly have more than three characters. Therefore, in the case of “startTagName”, Expat has to verify each byte (12 times in total), whereas the CheckNameMarco(...) only uses the bitstream_scan(...) one time.

4.4.2 Byte-at-a-time Character Data Scan VS Bit Stream Character Data Scan

Expat version in algorithm 5 starts up a markup item scanning operation (the scanLT(...) or scanRef(...)) if the first character is a markup item start (< or &). Otherwise, scan for character data until a markup item start or a CD data end (]] >) found. The running time of the while loop depends on the number of character data.

Algorithm 5 contentTok Function Pseudocode: Expat Version

```
static int PTRCALL
PREFIX(contentTok)(const ENCODING *enc, const char *ptr,
                   const char *end, const char **nextTokPtr)
{
   if( empty byte stream )
      return XML_TOK_NONE;
   switch(BYTE_TYPE(enc, ptr)){
      case ``<'':
         return PREFIX(scanLt)(...);
      case ``&'':
         return PREFIX(scanRef)(...);
      case illegal character or invalid encoding sequence:
            return XML_TOK_INVALID;
   }
   while(ptr != end){
      switch(BYTE_TYPE(enc, ptr)){
         case ``<'':
         case ``&'':
            return XML_TOK_CHAR;
         case ``]'':
            if(CD data end (``]]>'') occurred)
               return XML_TOK_INVALID;
         case illegal character or invalid encoding sequence:
               return XML_TOK_INVALID;;
         }
   }
   return XML_TOK_CHAR;
}
```

However, bit stream version in algorithm 6 accomplishes the same result by performing one time of the BIT_STREAM_SCAN(...)[6]. Since character data should count for a great percentage of a XML documents, the performance gain here should be significant.

[6]the process described in 2.

Algorithm 6 contentTok Function Pseudocode: Bit Stream Version

```
static int PTRCALL
PREFIX(contentTok)(const ENCODING *enc, const char *ptr,
                   const char *end, const char **nextTokPtr)
{
   if(empty byte stream)
      return XML_TOK_NONE;
   switch(BYTE_TYPE(enc, ptr)){
      case ``<'':
         return PREFIX(scanLt)(...);
      case ``&'':
         return PREFIX(scanRef)(...);
      case illegal character or invalid encoding sequence:
            return XML_TOK_INVALID;
   }
   BIT_STREAM_SCAN(ptr, MarkupStart);
   return XML_TOK_DATA_CHARS;
}
```

4.4.3 Byte-at-a-time Start Tag Scan VS Bit Stream Start Tag Scan

Expat in algorithm 7 first determines what the markup item is, and accordingly chooses one operation from the scanComment(...), scanCdataSection(...),, scanPi(...) or scanEndTag(...). Otherwise, an start tag is found. Then the start tag name is parsed in byte-at-a-time fashion. If an attribute is occurred, the scanAtts(...) is called. Instead, the bit stream version in algorithm 8 parses the name in one step. The performance gain here depends on the length and frequency of a start tag name.

Algorithm 7 ScanLt Function Pseudocode: Expat Version

```
static int PTRCALL
PREFIX(scanLt)(const ENCODING *enc, const char *ptr,
               cosnt char *end, const char **nextTokPtr){
   switch(BYTE_TYPE(enc, ptr)){
      case ``<!-'':
         return PREFIX(scanComment)(...);
      case ``<!['':
         PREFIX(scanCdataSection)(...);
      case ``<?'':
         return PREFIX(scanPi)(...);
      case ``</'':
         return PREFIX(scanEndTag)(...);
      case a valid name start character:
         ptr = ptr + MINBPC(enc);
   }
   while(not end of the byte stream){
      switch(BYTE_TYPE(enc, ptr)){
         case the occurrence of an attribute:
            return PREFIX(scanAtts)(enc, ptr, end, nextTokPtr);
         case an valid name character:
            ptr = ptr + MINBPC(enc);
         case ``>'':
            return XML_TOK_START_TAG_NO_ATTS;
         case ``/>'':
            return XML_TOK_EMPTY_ELEMENT_NO_ATTS;
         default:
            return XML_TOK_INVALID;
      }
   }
}
```

Algorithm 8 ScanLt Function Pseudocode: Bit Stream Version

```
static int PTRCALL
PREFIX(scanLt)(const ENCODING *enc, const char *ptr,
               cosnt char *end, const char **nextTokPtr){
   switch(BYTE_TYPE(enc, ptr)){
      case ``<!-'':
         return PREFIX(scanComment)(...);
      case ``<!['':
         PREFIX(scanCdataSection)(...);
      case ``<?'':
         return PREFIX(scanPi)(...);
      case ``</'':
         return PREFIX(scanEndTag)(...);
      case a valid name start character:
         ptr = ptr + MINBPC(enc);
         break;
   }
   CheckNameMarco(enc, ptr, end, nextTokPtr);
   if(the occurrence of an attribute)
      return PREFIX(scanAtts)(enc, ptr, end, nextTokPtr);
   if(at ``>'')
      return XML_TOK_START_TAG_NO_ATTS;
   if(at ``/>'')
      return XML_TOK_EMPTY_ELEMENT_NO_ATTS;
}
```

4.4.4 Byte-at-a-time Attribute Scan VS Bit Stream Attribute Scan

Once again, Expat version in algorithm 9 scans the attribute in byte-at-a-time according to the XML attribute production rule: processes the attribute name, looks for a equal sign, looks for the opening quote, scans the attribute value and processes any reference in the attribute value, and seeks for a matching closing quote. The scanning process repeats until the markup item end is found. In algorithm 10, the byte-at-a-time parsing of the attribute name and value is replaced with bit stream operations. Also, the optimization is made for the high frequency of the quote immediately after the attribute name and the opening quote immediately after the equal sign.

Algorithm 9 scanAtts Function Pseudocode: Expat Version

```
static int PTRCALL
PREFIX(scanAtts)(const ENCODING *enc, const char *ptr,
                 const char * end, const char **nextTokPtr){
   while(not end of the byte stream){
      switch (BYTE_TYPE(enc, ptr)){
        case a valid name case:
           ptr = ptr + MINBPC(enc);
        case a white space:
           skip all white space
        case the occurrence of an equal sign:
           skip all white space
           store the start quote
           for(;;){
               if(ptr matches with the start quote)
                 break;
               switch(BYTE_TYPE(enc, ptr)){
                 case illegal characters or encoding sequence:
                    return XML_TOK_INVALID;
                 case ``&'':
                    PREFIX(scanRef)(...);
                 case ``<'':
                    return XML_TOK_INVALID;
                 default:
                    ptr = ptr + MINBPC(enc);
               }
           }
           skip all white space
           switch(BYTE_TYPE(enc,ptr)){
              case ``>'':
                 return XML_TOK_START_TAG_WITH_ATTS;
              case ``/>'':
                 return XML_TOK_EMPTY_ELEMENT_WITH_ATTS;
              case an valid name start character
                 break;
              default:
                 return XML_TOK_INVALID;
           }
        default:
           break;
      }
   }
}
```

Algorithm 10 scanAtts Function Pseudocode: Bit Stream Version

```
static int PTRCALL
PREFIX(scanAtts)(const ENCODING *enc, const char *ptr,
                 const char * end, const char **nextTokPtr){
   do{
      CheckNameMarco(enc, ptr, end, nextTokPtr)
      if(at an equal sign){
         //optimize for = immediately after the attribute name
      }else{
         BIT_STREAM_SCAN(ptr, NonWS);
         if(not at an equal sign)
            return XML_TOK_INVALID;
      }
      if(at a start quote){
         store the start quote
      }else{
         BIT_STREAM_SCAN(ptr, NonWS);
         if(not at a quote)
            return XML_TOK_INVALID;
         else
            store the start quote
      }
      for (;;) {
         BIT_STREAM_SCAN(ptr, Quote);
         if (ptr matches the start quote)
            break;
         switch (BYTE_TYPE(enc,ptr)) {
            case ``&'':
               PREFIX(scanRef)(...);
               if(error in scanning reference)
                  return XML_TOK_INVALID;
            case ``<'':
              return XML_TOK_INVALID;
            default:
              ptr += MINBPC(enc);
         }
      }
      BIT_STREAM_SCAN(ptr, NonWS);
      if(at ``>'')
        return XML_TOK_START_TAG_WITH_ATTS;
      if(at ``/>'')
        return XML_TOK_EMPTY_ELEMENT_WITH_ATTS;
   }while(1);
}
```

4.4.5 Improvement on End Tags, References, CD Data, and Comments

Bit stream operations are applied the same way as described above with different markup item streams to end tags, references, CD data, and comments. The XML name of end tags is parsed with the “NameFollow”, as well as entity references. Character references are parsed with either the “NonDigit” or “NonHex” depending on decimal or hex representation of the character code point. The “NonDigit” has all non-decimal marked, whereas the “NonHex” has all non-hex marked. The while loop for finding the CD data end is replaced with the “CD_End_check”. In parsing comments, the while loop for finding a hyphen is replaced with the “Hyphen”.

Chapter 5

Performance Comparisons

5.1 Methodology

In Expat benchmark [18] and Parabix performance study [16], both of them use the XML document statistical report generator as their driver because the internal structure and API of parsers are different. The comparison base line is achieved by making the parser accomplished the same task. However, the two versions of Expat in this project have no change to the internal structure and API, so a driver that simply checks for the well-formedness of XML documents with no call back functions suits for the purpose. Furthermore, the longer the driver runs, the higher chance that an OS interrupt happens and corrupts the result.

The performance test collects three sets of data based on processor cycles per byte between original Expat and Expat with parallel bit stream technology: 1. The overall running time of each XML document. 2. The processor time spending on XmlContentTok(...). 3. The processor cycles for generating lexical item streams.

5.2 Test Platform

All tests are operated on Ubuntu 7.10 (Linux x86) with Intel Core 2 Duo processor 6400 desktop. The source code is complied with GCC 4.1.3 with O2 optimization. Since there is no significant performance difference between the original Expat compiled with gcc and the original Expat complied with g++, but Parabix is written in C++, both versions of Expat are compiled with g++. The detailed hardware and software information is in table 5.1.

BOM_Profiler (Binary Order of Magnitude Execution Time Profiler) is a lightweight

CPU Name	Intel Core 2 Duo processor 6400
System Bus MHz	1066
CPU MHz	2128
FPU	Integrated
CPU(s) enabled	4 core, 2 chip, 2 cores/chip
Primary Cache	32 KB I + 32 KB D on chip per core
Secondary Cache	2048 KB (I+D) on chip
Memory	2 GB DIMM
OS	Ubuntu 7.10 (Linux x86)
Complier	GCC 4.1.3
Base Pointers	64-bit
Peak Pointer	64-bit
Other Software	BOM_Profiler

Table 5.1: Hardware and Software Configuration

multi-platform execution time profiling utility based on processor cycle counters. It not only detects processor cycle pattern for processing a fixed small number of bytes[1], but also rules out the inaccuracy caused by OS interrupts and processor cycle overflow [15].

5.3 XML Document Characteristics

Seven XML documents are chosen for the experiment. dewiki.xml, jawiki.xml and arwiki.xml are the document-oriented documents, whereas roads.gml, po.xml, soap.xml and worst.xml are the data-oriented documents. The markup density [16] spreads from as low as 0.07 to as high as 1. The detail information can be found in table 5.2. None of these files has CDATA and comments but it does not matter much because they are rarely used in XML documents.

5.4 Experiment Results

Each test case executes five times before collecting data in order to escape from the overhead of bringing the driver and the whole XML documents into main memory. Then BOM_Profiler collects the processor cycle per byte pattern on the test case. An python

[1]1024 bytes are recommended.

File Name	dewiki.xml	jawiki.xml	arwiki.xml	roads.gml	po.xml	soap.xml	worst.xml
Empty Tag	5814	903	23139	6907	0	10000	0
Avg. Lgth of Empty Tag	9	10	10	11	0	25	0
Start Tag	197582	36538	661502	41454	2317055	80002	10000001
Avg. Lgth of Start Tag	9	9	9	37	12	16	3
Total Attribute	18806	3527	62745	55265	463396	19999	0
Attribute Name	18806	3527	62745	55265	463396	1999	0
Avg. Lgth of Attribute Name	8	8	8	6	7	5	0
Attribute Value	18806	3527	62745	55265	463396	19999	0
Avg. Lgth of Attribute Value	7	7	7	5	5	7	0
End Tag	197582	36538	661502	41454	2317055	80002	10000001
Avg. Lgth of End Tag	8	8	8	17	10	9	4
Text Item	1153459	153736	1833224	89827	4632574	170014	0
Avg. Lgth Of Text Item	50	39	47	21	4	2	0
Reference	796851	89949	555377	0	0	0	0
Avg. Lgth of Reference	4	4	4	0	0	0	0
File Size (kB)	66240	7343	113082	4175	76450	2717	68359
Markup Density	0.07	0.13	0.21	0.55	0.76	0.87	1.00

Table 5.2: XML Document Characteristics

File Name	dewiki.xml	jawiki.xml	roads.gml	po.xml	soap.xml
Cycles Per Byte	16.502	18.777	36.846	46.021	51.254

Table 5.3: Expat Performance on an 100K Buffer

program automatically picks the patterns consisting of 99% of the whole to avoid inaccuracy from OS interrupts and processor cycle overflow. This process is executed 20 times and the average is used as the final processor cycle per byte pattern.

Since the test platform is identical from [16], it would be interesting whether it is possible to re-generate Expat performance in figure 3.5. The original Expat with no callback functions is tested. An preloaded 100K buffer is fed into the XML_Parser(...) each time until the end of the whole XML document. BOM_Profiler starts right before the XML_Parser(...) and ends right after the function. The final result is in table 5.3. All cases except dewiki.xml run slightly faster than the figure 3.5. It is desirable because the figure 3.5 includes the cost of the callback functions for producing the statistical report.

In order to prove the speedup of Expat with parallel bit stream technology, 9 experiments in total are designed based on three criteria: which version (BitStreamScan or Expat), the size of the input buffer (1024 bytes, 99968 bytes or the whole document), the size of the BitStreamScan internal buffer (1K or 100K). The result is in table 5.4. Here is how to comprehend the table:

- The version is distinguished with the word "Bitstream" or "Expat". The size of the input immediately follows the version word. The size of the BitStreamScan internal buffer comes last except the 3 Expat experiments. For example, the "Bitstream_1024_1k" means the BitStreamScan version parses 1024-byte XML data every run with a 1K internal buffer. The "Expat_all" means the Expat version processes a buffer containing all XML data.

- The "Bit", "Tok" and "Total" are cycles per byte for the time of generating lexical item streams, the processing time of the XmlContentTok(...), and the runtime of the XML_Parse(...) respectively.

- The "Func$\uparrow$" is the speedup in cycles per byte from the XmlContentTok(...) compare to the Expat version configured with the same input size. Since the producing lexical

	Bitstream_1024_1k							Bitstream_1024_100k							Expat_1024	
	Cycles Per Byte					Percentage		Cycles Per Byte					Percentage		Cycles Per Byte	
File Name	Bit	Tok	Total	Func↑	Full↑	%Func↑	%Full↑	Bit	Tok	Total	Func↑	Full↑	%Func↑	%Full↑	Tok	Total
arwiki.xml	3.213	3.880	22.504	9.140	10.638	56%	32%	4.045	3.905	23.491	8.283	9.651	51%	29%	16.233	33.142
dewiki.xml	3.086	4.383	19.928	4.079	6.087	35%	23%	3.758	4.368	20.320	3.422	5.695	30%	22%	11.548	26.015
jawiki.xml	3.467	5.230	22.141	6.357	8.775	42%	28%	4.230	5.240	22.874	5.584	8.042	37%	26%	15.054	30.916
roads.gml	3.090	6.200	40.122	3.949	5.801	30%	13%	4.805	6.312	42.773	2.122	3.150	16%	7%	13.239	45.923
po.xml	2.938	14.195	59.818	3.591	3.817	17%	6%	4.941	14.411	62.081	1.372	1.554	7%	2%	20.724	63.635
soap.xml	2.954	15.241	65.963	4.071	5.119	18%	7%	5.332	15.278	69.382	1.656	1.700	7%	2%	22.266	71.082
worst.xml	2.931	31.191	90.277	-7.620	-2.725	-29%	-3%	5.073	31.301	92.167	-9.872	-4.615	-37%	-5%	26.502	87.552

	Bitstream_99968_1k							Bitstream_99968_100k							Expat_99968	
	Cycles Per Byte					Percentage		Cycles Per Byte					Percentage		Cycles Per Byte	
File Name	Bit	Tok	Total	Func↑	Full↑	%Func↑	%Full↑	Bit	Tok	Total	Func↑	Full↑	%Func↑	%Full↑	Tok	Total
akwiki.xml	3.639	4.007	21.653	8.274	10.881	52%	33%	3.396	3.898	21.522	8.626	11.012	54%	34%	15.920	32.534
dewiki.xml	3.489	3.795	20.286	4.415	5.626	38%	22%	3.344	3.801	19.954	4.554	5.958	39%	23%	11.699	25.912
jawiki.xml	3.802	5.064	22.066	5.953	7.749	40%	26%	3.542	4.995	21.511	6.282	8.304	42%	28%	14.819	29.815
roads.gml	3.283	6.031	39.471	3.444	5.247	27%	12%	3.206	6.158	40.320	3.394	4.398	27%	10%	12.758	44.718
po.xml	3.290	14.115	59.798	3.078	3.068	15%	5%	3.113	14.308	59.635	3.062	3.231	15%	5%	20.483	62.866
soap.xml	3.060	14.761	65.889	3.487	3.789	16%	5%	3.176	14.697	66.870	3.435	2.808	16%	4%	21.308	69.678
worst.xml	3.198	31.247	89.104	-8.168	-2.459	-31%	-3%	3.100	31.439	89.853	-8.262	-3.208	-31%	-4%	26.277	86.645

	Bitstream_all_1k							Bitstream_all_100k							Expat_all	
	Cycles Per Byte					Percentage		Cycles Per Byte					Percentage		Cycles Per Byte	
File Name	Bit	Tok	Total	Func↑	Full↑	%Func↑	%Full↑	Bit	Tok	Total	Func↑	Full↑	%Func↑	%Full↑	Tok	Total
akwiki.xml	3.621	3.976	16.694	8.333	10.054	52%	38%	3.544	3.918	16.598	8.468	10.150	53%	38%	15.930	26.748
dewiki.xml	3.490	4.268	15.519	3.933	5.881	34%	27%	3.494	4.100	15.348	4.097	6.052	35%	28%	11.691	21.400
jawiki.xml	3.817	5.125	18.792	5.929	7.790	40%	29%	3.678	4.975	18.434	6.218	8.148	42%	31%	14.871	26.582
roads.gml	3.349	6.080	35.458	3.942	5.557	29%	14%	3.344	6.189	35.355	3.838	5.660	29%	14%	13.371	41.015
po.xml	3.298	14.418	N/A	2.800	N/A	14%	N/A	3.296	14.291	55.542	2.929	N/A	14%	N/A	20.516	N/A
soap.xml	3.262	15.090	62.775	2.973	2.976	14%	5%	3.366	15.274	61.925	2.685	3.826	13%	6%	21.325	65.751
worst.xml	3.295	31.063	N/A	-8.103	N/A	-31%	N/A	3.279	31.396	N/A	-8.420	N/A	-32%	N/A	26.255	N/A

Table 5.4: Performance Result

item streams is pulled out of the XmlContentTok(...), the real XmlContentTok(...) time for the BitstreamScan version should be the sum of the "Bit" and "Tok". The "Full$\uparrow$" is the speedup of the XML_Parse(...). The "%Func$\uparrow$" and "%Full$\uparrow$" are the percentage of the improvement.

- The "N/A" indicates the data is not available.

5.5 Analysis

Generally speaking, the speedup of the XmlContentTok(...) (under %Func$\uparrow$) is from 13% to 56%, while the speedup of the BitStreamScan version (under %Full$\uparrow$) is from 4% to 38%. The only slowdown case is worst.xml. Actually, it is supposed to happen because the nature of the document. worst.xml is constructed by repeating "$< a >< /a >$", so the bit stream operations are forced back to byte-at-a-time operation. However, the cost associated with the bit stream operations, generating lexical item streams and the conversion between byte position and code unit position, becomes the absolute overhead that there is no way to recover. On the other hand, the original Expat is the optimistic solution because it is in byte-at-a-time fashion. Two points are worth mentioning here.

1. Even in this absolute nightmare for BitStreamScan, the slowdown is only about 3%.

2. This case should never happen.

One might question why the runtime of the "Expat_99968" is longer than the table 5.3 even though they have the same configuration. The slowdown comes from setting a timer before and after both the lexical item stream generating process and the XmlContentTok(...). Without these two timers, they are the same.

The generating lexical item streams takes about 3 or 4 cycles per byte, which is 1 cycle per byte greater than the data in [16]. The 1 byte increase is caused by adjusting the last byte of the internal buffer to be a complete sequence, and the BitstreamScan version generates three more lexical item streams than [16]. The Bitstream_1024_100K takes even longer because the 100K internal buffer has never fully used. With only 1024 bytes XML data available every time, the overhead of managing the 100K buffer is taking on by 1024 bytes instead of 99968 bytes. This configuration should never be used. Comparing Bitstream_99968_1k and Bitstream_99968_100k concludes that there is no difference between the 1K internal buffer

	BitStream_1024_1K						Expat_1024	
	Cycles Per Byte							
File Name	Bit	Tok	Total	Func↑	Full↑	Full↑ − Func↑	Tok	Total
arwiki.xml	3.214	3.883	22.219	6.820	6.615	-.205	13.917	28.834
dewiki.xml	3.104	4.381	19.739	1.040	.713	-.327	8.525	20.452
jawiki.xml	3.455	5.231	21.752	3.628	3.281	-.347	12.314	25.033
roads.gml	3.102	6.219	40.243	2.162	2.242	.80	11.483	42.485
po.xml	2.924	14.189	59.257	4.063	3.861	-.202	21.176	63.118
soap.xml	2.929	15.218	65.686	5.121	6.060	.939	23.268	71.746

Table 5.5: Performance Result with no Newline Character

and the 100K buffer for generating lexical item streams as long as the internal buffer can be fully used.

With the same internal buffer (1K or 100K), the speedup gradually increases as the size of the coming XML data changes from 1024 bytes to the whole XML documents. The original Expat has the same increase. This increase is not brought by the BitStreamScan, but is brought by Expat itself. Since Expat makes a copy of the XML data before any parsing, it is more efficient to make a whole copy of the document once than many copies of a part of the document. Also, partial markup item situation is eliminated as well.

Comparing the cycles per byte speedup from the XmlContentTok(...) (under Func↑) and the whole program (under Full↑), 2-cycle difference is observed. Since all modification are made in the XmlContentTok(...), one assume the speedup would only come from that function. However, the original Expat processes a three line character data with six passes, whereas the BitStreamScan version processes it with only one pass. The less passes reduces the function call and branching in the doContent(...), which leads to the 2 cycles improvement. The same files with all newline characters removed are re-tested and the result shows in table 5.5. The difference between Full↑ and Func↑ (under Full↑ − Func↑) is eliminated.

What factors have influence on the degree of the speedup? The speedup under %Full↑ suggests that the speedup has a negative relationship with the markup density. However, arwiki.xml does not follow the rule since it has a higher markup density and speedup. It is because dewiki.xml has more percentage of reference over file size than arwiki.xml, 4.8% for dewiki.xml and 2.0% for arwiki.xml. Compare to other average length in Table 5.2, the average length of reference is much smaller, so the performance gain from parsing reference is much smaller. Therefore, markup density is good as a general indicator, but the average length of markup item, text item and reference along with their percentage over file size is much accurate indicator of the degree of the speedup.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

Extensible Markup Language (XML) has been popular with recording publishing information and web services because of its textual human readable format, simplicity and generality. However, machine cannot directly understand XML documents. It requires a middle ware, an XML parser, to interact with XML documents. Since the size of XML documents can be large from a few bytes to gigabytes, the performance of XML parsers becomes more and more critical as the size of XML documents grows. Expat does perform better comparing to the other parsers, but it significantly slower than Parabix. One of the major advantages of Parabix is parallel bit stream technology.

It is a good idea to embed parallel bit stream technology into Expat so that it can perform better. The project looks into the source code of Expat and identifies where and how Expat processes XML document content. The byte-at-a-time parsing strategy of the original Expat is replaced with parallel bit stream technology.

The experiment result clearly shows that Expat with parallel bit stream technology gains speedup over the original Expat. In general, the degree of the speedup goes up as markup density decreases. More accurately, the degree of the speedup depends on XML document characteristics including the number of reference, and the average length of text item, tag and attribute name, and attribute value.

6.2 Future Work

Currently, parallel bit stream technology is only applied to XML document content. Later, it can be applied to document type definition (DTD) but the performance gain should be small because XML documents normally have no DTD or a fairly small number of DTD.

The parallel bit stream technology used in this project is version 1.0. Currently, the version 2.0 is under development. The overhead of generating lexical item streams is even smaller. Also, some well-formedness checking is embedded into lexical item streams. The version 2.0 should improve the performance more.

Bibliography

[1] Apache2.0. http://httpd.apache.org/docs/2.2/ja/developer/thread_safety.html.

[2] Welcome to the Development Home of Parabix - World's Fastest XML Software. http://parabix.costar.sfu.ca/.

[3] Development History. http://www.w3.org/XML/hist2002, November 1996.

[4] Extensible Markup Language (XML) 1.1 (Second Edition). http://www.w3.org/TR/xml11/, September 2006.

[5] Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/2008/REC-xml-20081126/, November 2008.

[6] FastParser. http://wiki.services.openoffice.org/wiki/FastParser, February 2008.

[7] bitlex.h. http://parabix.costar.sfu.ca/browser/trunk/src/bitlex.h, February 2010.

[8] Expat(XML). http://en.wikipedia.org/wiki/Expat_(XML), March 2010.

[9] Glossary of Unicode Terms. http://unicode.org/glossary/, January 2010.

[10] UTF-8. http://en.wikipedia.org/wiki/UTF-8, March 2010.

[11] XML. http://en.wikipedia.org/wiki/XML#cite_note-Cover_pages_list-4, April 2010.

[12] xmlparse.c. http://expat.cvs.sourceforge.net/viewvc/expat/expat/lib/xmlparse.c?view=markup, February 2010.

[13] xmltok_impl.c. http://expat.cvs.sourceforge.net/viewvc/expat/expat/lib/xmltok_impl.c?view=markup, February 2010.

[14] Ken Herdy Cameron, Rob and Ehsan Amiri. Parallel bit stream technology as a foundation for xml parsing performance. presented at international symposium on processing xml efficiently: Overcoming limits on space, time, or bandwidth, montral, canada, august 10, 2009. In *Proceedings of the International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth. Balisage Series on Markup Technologies*, volume 4 (2009) doi:10.4242/BalisageVol4.Cameron01.

[15] Robert D. Cameron. BOM_Profiler. http://parabix.costar.sfu.ca/.

[16] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. High performance xml parsing using parallel bit stream technology. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 222–235, New York, NY, USA, 2008. ACM.

[17] Robert D. Cameron and Dan Lin. engine.h. http://parabix.costar.sfu.ca/browser/trunk/src/engine.h.

[18] Clark Cooper. Benchmarking XML Parsers. http://www.xml.com/pub/a/Benchmark/article.html?page=1.

[19] Clark Cooper. Using The Perl XML::Parser Module. http://www.xml.com/pub/a/98/09/xml-perl.html.

[20] Clark Cooper. Using Expat. http://www.xml.com/pub/a/1999/09/expat/index.html, September 1999.

[21] Bob DuCharme. Documents vs. data,schemas vs. schemas. In *XML 2004*, Washington D.C., 2004.

[22] Kwaclaw Fdrake. Expat XML Parser. http://sourceforge.net/projects/expat/.

[23] Guha. XML in Mozilla. http://www.mozilla.org/rdf/doc/xml.html.

[24] F. Yergeau P. Hoffman. UTF-16, an encoding of ISO 10646. http://www.ietf.org/rfc/rfc2781.txt, February 2000.

[25] F. Yergeau P. Hoffman. UTF-16, an encoding of ISO 10646. http://unicode.org/faq/utf_bom.html#UTF32, February 2010.

[26] W.P. Petersen and P. Arbenz. Simd, single instruction multiple data. In *Introduction to Parallel Computing: A practical Guide with Examples in C*, page 85, the United States, 2004. Oxford Universitt Press.

[27] Sourceforge.net. The Expat XML Parser. http://expat.sourceforge.net/.

[28] Jon Stokes. SIMD architectures. http://arstechnica.com/old/content/2000/03/simd.ars, March 2000.