INTERNATIONAL JOURNAL OF PURE AND APPLIED RESEARCH IN ENGINEERING AND TECHNOLOGY

A PATH FOR HORIZING YOUR INNOVATIVE WORK

A REVIEW OF ACCELERATION OF XML PARSING THROUGH PREFETCHING

FARHANULLAH A KHAN[1], PROF. S D DESHPANDE[2]

1. M.E., PRMCEAM, Bandera.
2. Dept. of PG Studies, PRMCAM, Bandera.

Accepted Date: 26/02/2015; Published Date: 01/03/2015

Abstract: Extensible Markup Language (XML) has turn out to be a extensively used standard for data representation and exchange. However, its features also bring in significant overhead intimidating the performance of recent applications. Here, we present a study of XML parsing and settle on that memory-side data loading in the parsing step incurs a major performance overhead, as much as the computation does. XML parsing is the process of reading an XML document and providing an interface to the user application for accessing the document. In this paper we present a study on XML parsing through different classic prefetching algorithms. Without a parser, your code cannot be understood. Computers require instruction. An XML parser provides vital information to the program on how to read the file. Parsers come in multiple formats and styles. This paper is an overview of the various issue involved in XML parsing through different prefetching algorithms. Hence, we propose memory-side acceleration which incorporates of data prefetching techniques, and can be applied on top of computation-side acceleration to speed up the XML data parsing.

Keywords: XML Parsing, Acceleration

Corresponding Author: MR. FARHANULLAH A KHAN

Access Online On:

www.ijpret.com

How to Cite This Article:

Farhanullah A. Khan, IJPRET, 2015; Volume 3 (7): 154-161

PAPER-QR CODE

INTRODUCTION

EXTENSIBLE markup language (XML) is known for its language neutrality, application independency and flexibility, and has thus been adopted as the standard in data exchange and representation. Although XML is prevalent with its many benefits, due to its verbosity and descriptive nature, it has also introduced a heavy performance overhead [1], [2]. Generally, XML parsing is both memory and computation intensive. It consumes about 30 percent of processing time in many Web service applications [4], and has become a major performance bottleneck in database servers [5]. To improve the performance of XML processing, most existing schemes have promoted acceleration from the computation side. Therefore, as opposed to previous computation acceleration studies, we propose to accelerate XML parsing from the memory side with the incorporation of data prefetching techniques. Unlike computation-side acceleration, which has a strong dependency on the parsing model, memory-side acceleration is generic and can be applied regardless of the underlying parsing model.

2. The XML Parsing Process

We present the review of some previous work in Table2.1 on the last page.

XML parsing is a process that scans through the input XML documents, breaks them into small elements, and builds corresponding inner data representation. It is a pre-requisite for any processing of an XML document because an XML document has to be parsed before any other operations can be performed. However, XML parsing is also very expensive due to the high overhead incurred by both computation and memory access.

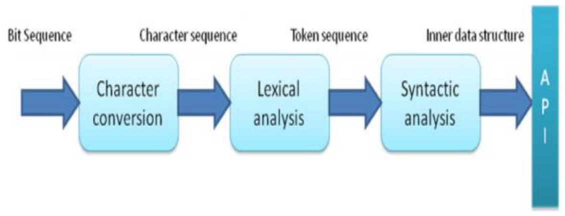

Fig. 1. XML parsing process.

Usually, XML data parsing consists of three steps: *character conversion, lexical analysis* and *syntactic analysis*, as shown in Figure 1[1].The first parsing step, *character conversion:* The first parsing step involves converting a bit sequence from an XML

document to the character sets the host programming language understands. For example, documents written in Western, Latin-style alphabets are usually created in UTF-8, while Java usually reads characters in UTF-16. In most cases, a UTF-8 character can be converted to UTF-16 by simply padding 8-bit leading zeros. For example, the parser converts "<" "a" ">" from "3C 61 3E" to "003C 0061 003E" in hexadecimal representation.

Lexical analysis: The second parsing step involves partitioning the character stream into subsequences called tokens. Major tokens include a start element, text, and an end element. A token can itself consist of multiple tokens. Each token is defined by a regular expression in the World Wide Web Consortium (W3C) XML specifications. For example, a start element consists of a "<", followed by an element name, zero or more attributes preceded by a space-like character, and a ">".partitions the character sets into subsequences called tokens, like *start element*, *text,* and *end element*. Each token is defined by a regular expression in the World Wide Web Consortium (W3C) XML specifications [8]. The third parsing step, *syntactic analysis*, verifies the structure of tokens by checking that they have been properly nested. It is usually implemented by *pushdown automaton* (PDA)[14]. After syntactic analysis, the PDA organizes tokens into different data representations available for subsequent accesses or modifications via various application programming interfaces (APIs) provided by different parsing models. The first two steps stay the same among different parsing models. However, the third step, *syntactic analysis:* exhibit variable behaviors when different parsing model is applied [6]. The third parsing step involves verifying the tokens' well-formed-ness, mainly by ensuring that they have properly nested tags. The pushdown automaton (PDA) the following transition rules:

1. The PDA initially pushes a "$" symbol to the stack.

2. If it finds a start element, the PDA pushes it to the stack.

3. If it finds an end element, the PDA checks whether it is equal to the top of the stack.

- If yes, the PDA pops the element from the stack. If the top element is "$", then the document is "well-formed." Done! Otherwise, the PDA continues to read the next element.

- If no, the document is not "well formed." Done!

3. XML Parsing Modelling

Most XML parsers can be classified into two broad categories, based on the types of API that they provide to the user applications for processing XML documents: event-driven parser and tree-based parser [1]. On one hand, event-driven parser simply parses the document and associates any tag it finds along the way with corresponding event, including the start and end of the document, finding a text node, finding child elements, and hitting a malformed element.

It transmits and parses XML info sets sequentially at runtime [12]. The parser itself does not store any information of the XML document, so that the application can just access partial data before parsing is completed. As a result, event-driven parser has an enviably small memory footprint and low latency, making it suitable for streaming or forward only applications. Event-driven model can be further divided into two classes: pull parser and push parser, according to the parser- application interaction. Simple API for XML (SAX) [7, 1] adopts the push model, which uses callback functions to report all the events from the parser to the application. In contrast, Sax [18] adopts the pull model, in which clients pull XML data when it is needed so that it can skip uninterested events. As shown in upper part of Figure

2, SAX parses the XML document and then pushes the XML information into application in terms of SAX events. On the other hand, tree-based parser reads the entire content of an XML document into memory and creates an in-memory tree structure to represents parent-child sibling information. Only after parsing is complete, constructed trees can be navigated freely and parsed arbitrarily for the duration of the document processing, which makes this parser suitable for massive and frequent updates. This flexibility, however, comes at a great cost of potentially large memory requirement and significant access delay, especially when large document is processed. Document Object Model (DOM) [8] is the official W3C standard for tree-based parser. As shown in bottom part of Figure 2, DOM parser processes XML data, creates an object-oriented hierarchical representation of the document and offers the full access to the XML data. In this study, we focus on the two most popular parsing models, namely, SAX and DOM [19].

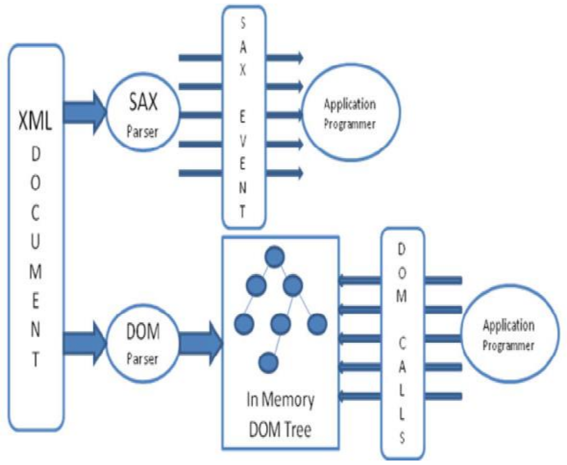

Figure 2: SAX and DOM Parsing Flow

4. Prefetching Techniques

Data prefetching has been proposed as a speculative technique to bridge the speed gap between CPU and memory subsystem [8,1].It alleviates the performance degradation from the long-latency memory accesses by predicting the memory access pattern of the application and speculatively prefetching data that would be used in future computation. Considering that the CPU memory performance gap is on the order of hundreds of processor clock cycles, prefetching is an attractive way to remove the affect of long latency memory accesses.

5. Classic Prefetching

Algorithm

Prefetching techniques has been well studied and lots of algorithms have been proposed. We list some classic prefetching algorithms below.

Sequential prefetching prefetchs the block or blocks that follow the current demanded block, and is fit for the programs with the consecutive memory access pattern [1]. As an improvement, *Sequential tagged prefetching* [1] issues a prefetch upon a cache miss as well as when a prefetched block is referenced for the first time, thus it requires an extra bit per block to mark the prefetch state. The *Sequential prefetching* family increases the performance on a broad range of applications at a low cost, however, at the expense of many useless prefetches.

Stride prefetching makes prefetch requests according to the observed strides that separate memory addresses flow. Conventional *stride prefetching* uses a record table indexed by the program counter (PC) that associates strides to the loads following this kind of memory access pattern [21]. If address *a* is referenced by a load that hits in the table, the matching entry indicates that the load is following a stride pattern, then prefetcher issues their quest for addresses *a+s*, where *s* is the associated stride. *Strem Prefetching* traces a sequence of nearby misses when their addresses follow the same positive or negative direction in a small memory region. In some design, there always exsites a streaming buffer to store the fetched data. *Correlating prefetching* predicts future addresses from tables that record the past memory program behavior .Usually, it generalizes the stride table by registering the stream of addresses associated either to the load PC or to an address that misses in the corresponding cache level.

6. Software Prefetching Vs. Hardware Prefetching.

According to how prefetching is implemented, it can be classified into two classes: software prefetching and hardware prefetching.

- Software Prefetching

Software prefetching need to introduce new prefetching instructions into the instruction set architecture (ISA), which could bring data at specified memory addresses into cache. It is assisted by compiler algorithms to insert software prefetching instructions into proper places of the source code. In the preprocessing stage, compiler gets the global information about memory data access pattern, locates those data-sets that are lean towards cache misses and calculates the positions to insert the prefetching instruction. In Intel® Pentium®

4 processor, it enables using the four prefetch instructions introduced with Streaming SIMD Extensions (SSE). These instructions are hints to bring a cache line of data in to various cache levels. Since software prefetching gets the assistance from compiler or programmer, it can acquire a globule map of data accesses, handle irregular access patterns and make more precise prefetching's. However, the insertion of the prefetching instruction is statically determined so software prefetching cannot adapt to the phase change of the application. Since new instructions need to be added, recompilation is required, so these do not benefit the scenarios where recompilation is inconvenient.

- Hardware Prefetching

Different from software prefetching that statically inserts prefetching instructions by compiler, hardware prefetching frees the need to expand instruction set architecture and frees the compiler from revising the source code of applications. It automatically determines the data accesses that might cause cache misses and then make prefetching requests. Its decision is based on the recorded history information so that it can adapt to the phase change of application. However, it must consume extra hardware resource and is unable to gain a complete picture of the whole memory pattern. Therefore, it does not suit for the case of irregular data access and short arrays for the penalty of history start-up. In our study, we focus on hardware prefetching for its advantage of no revise of the source code.

7. Conclusion

Different from previous research work which focused on computation acceleration of XML parsing, we studied the process of XML parsing and classic prefetching algorithms. We then proposed to make acceleration for XML parsing.

8. Future Work

The next step of this research project is to integrate memory-side and computations-side accelerators of XML parsing into a single core, and optimize its performance and power consumption. Then, integrate this core onto many-core architectures to act as a Data Exchange Frontend (DEF).

REFERENCES

1. Zhang W. and R. van Engelen," High-Performance XML Parsing and Validation with Permutation Phrase Grammer Parsers", *in International Conference on Web Service* (ICWS'08) IEEE*,* 2008.

2. Zacharia Fadika 1, Michael R. Head 2, Madhusudhan Govindaraju, "Parallel and Distributed Approach for Processing Large-Scale XML Datasets", *10th International Conference on Grid Computing, IEEE/ACM*, 2009.

3. Pan Y., W. Lu, Y. Zhang, and K. Chiu, "A Static Load-Balancing Scheme for Parallel XML Parsing on Multi-core CPUs", *in 7th International Symposium on Cluster Computing and the Grid*, *IEEE Brazil*, May 2007.

4. Kai Ning, Luoming Meng, "Design and Implementation of the DTD-based XML Parser", *in Proceedings of ICCT* 2003.

5. B. Naga malleshwar Rao, N. Samba Siva Rao, V. Khanaa, "Exploiting XML Dom for Restricted Access of Information", *in International Journal of Recent Trends in Engineering, Vol. 2, No. 4, November 2009.*

6. K. Chiu, T. Devadithya, W. Lu, and A. Slominski, K. Chiu, T. Devadithya, W. Lu, and A. Slominski, "A Binary XML for Scientific Applications," *Proc. First Int'l Conf. e-Science and Grid Computing*, 2005.

7. D. Callahan, K. Kennedy, and A. Portereld, “Software Prefetching,” *Proc. Fourth Int’l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 40-52, Apr. 1991.

8. A. J. Smith, “Sequential Program Prefetching in Memory Hierarchies”, *IEEE Trans. Computers,* vol. C-11, no. 12, pp. 7-21, Dec. 1978.

9. J. Fu and J. Patel, “Stride Directed Prefetching in Scalar Processors”, *Proc. 25th Ann. Int’l Symp. Microarchitecture (MICRO 25)*, 1992.

10. N. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, *Proc. Int’l Symp. Computer Architectures (ISCA)*, 1990.

11. S. Srinath and Y.N. Patt, “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, *Proc. Int’l Symp. High-Performance Computer Architecture (HPCA),* 2007.

12. S. Srinath and Y.N. Patt.

13. W.Y. Chen, S.A. Mahlke, P.P. Chang, and W.W. Hwu, “Data Access Microarchitectures for Superscalar Processors with Compiler- Assisted Data Prefetching”, *Proc. 24th Ann. Int’l Symp. Microarchitecture (Micro computing 24)*, 1991.

14. A. Lai, C. Fide, and B. Falsafi.