HPXA: A Highly Parallel XML Parser

Isaar Ahmad
Electrical Engineering
IIT Delhi
New Delhi, India
isaar.ahmad@gmail.com

Sanjog Patil
Electrical Engineering
IIT Delhi
New Delhi, India
jvl142701@ee.iitd.ac.in

Smruti R. Sarangi
Computer Science and Engineering
IIT Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

***Abstract*—State of the art XML parsing approaches read an XML file byte by byte, and use complex finite state machines to process each byte. In this paper, we propose a new parser, HPXA, which reads and processes 16 bytes at a time. We designed most of the components ab initio, to ensure that they can process multiple XML tokens and tags in parallel. We propose two basic elements – a sparse 1D array compactor, and a hardware unit called LTMAdder that takes its decisions based on adding the rows of a lower triangular matrix. We demonstrate that we are able to process 16 bytes in parallel with very few pipeline stalls for a suite of widely used XML benchmarks. Moreover, for a 28nm technology node, we can process XML data at 106 Gbps, which is roughly 6.5X faster than competing prior work.**

***Keywords*-XML parser, multibyte input, highly parallel parser**

I. Introduction

Extensible Markup Language(XML) is undoubtedly the de facto standard for transfer of structured information over the Internet. XML documents convey structured information in a self-explanatory manner while maintaining platform independence. XML documents need to be parsed into a tree based structure before they can be effectively accessed and modified by applications. This is one of the most performance sensitive operations while processing an XML document. The *parse tree* is then handed over to the application for subsequent processing. Given the large size of XML documents today, the performance requirements of XML parsers have increased, necessitating solutions in hardware. Some of these have found their way into commercial processors as dedicated accelerators such as the parser in IBM PowerEN [1].

Parsing broadly comprises of the following tasks: well-formedness checks on the XML document, extracting elements or tokens, and finally creating and storing the parse tree in memory. In this paper, we propose a novel XML parser called HPXA, which is 10-30X faster than state of the art prior work in terms of throughput (Gb/s). HPXA achieves this by reading and processing 16 bytes of data at a time; as opposed to prior proposals that mainly focus on reading the XML document byte by byte. However, this makes the process of parsing difficult, as a set of 16 bytes can contain up to 6 XML tags and also need not be aligned to XML tag boundaries. Moreover, a conventional stack based approach to construct a parse tree is hard to parallelize. We propose novel mechanisms to process up to 6 XML tags in one go, and subsequently show that it is possible to process large XML files and achieve a CPB (cycles per byte) that is close to the theoretical limit of 1/16 (0.0625). This is a substantial improvement over prior work as we shall see in the next section.

II. Background and Related Work

A. Brief Overview of XML Parsing

The quintessential method of XML parsing involves reading the file byte by byte, delineating character boundaries(for multi-byte characters), and using complex FSMs to identify characters marking the beginning and end of tags, attributes, and content. Incomplete tag values are temporarily buffered, and then transferred to a permanent location in memory (if needed). To create a parse tree, we push a start-tag on to a last-in first-out stack, and pop it out, when its corresponding end-tag is read.

B. Related Work

Year	Paper	CPB	Through-put(Gbps)	Frequency (MHz)
2004	ZuXA [2]	1	16*	2048
2006	Wie Lu et al. [3] †	27 [4]	0.68	2300
2009	SCBXP [5]	0.5	1.6-1.8	125
2010	XPA [4]	1	1.04	130
2012	PXP [6]	0.2505	3.992	125
2017	**HPXA**	**0.0625**	**106**	**833**(@28 nm)
* estimated based on CPB and frequency, † only in software				

TABLE I: Summary of prior work

ZuXA [2] employs BART-FSM, a programmable state machine technology to store a large number of state transition rules, and a novel hash function to quickly find the correct transition for a given input and current state. It operates sequentially, with a CPB of 1. XPA [4] is another proposal that uses the BART-FSM. In comparison, Wie Lu et al. [3] propose a parallel algorithm, where we first find roughly equal sized XML sub-trees, and then we parse each sub-tree separately.

Skeleton CAM based XML Parsing(SCBXP) [5] uses FIFO queues at the start to buffer short strings, which are checked for well-formedness and stored in a CAM. The CAM is used to skip well-formedness checks for strings that have already been verified to be XML compliant. Subsequently, the

978-3-9819263-0-9/DATE18/©2018 EDAA

characters are sent to multiple FSMs to parse the tree. The CPB is close to 0.5 because we still cannot process many tags in parallel. The Parallel Speculative DOM based Parser (PXP) [6] breaks an XML document into similar sized sub-documents. These are then parsed on parallel hardware units to create sub-trees, and then merged to form the final parse tree. Splitting the document in roughly equal sizes sometimes requires significant computational effort. The last line shows the numbers for HPXA (at least 6X higher throughput).

III. Design Details

A. Overview of the Parse Tree

The HPXA hardware is an in-order pipeline. It takes a generic XML file as input (16 bytes at a time), and stores the final parse tree in memory. If there is any syntax error in the file, then it promptly flags the error, and the process of parsing stops. The tree structure consists of nodes for each token extracted from the XML document. A sample XML document and a view of the corresponding tree are shown in Figure 1.

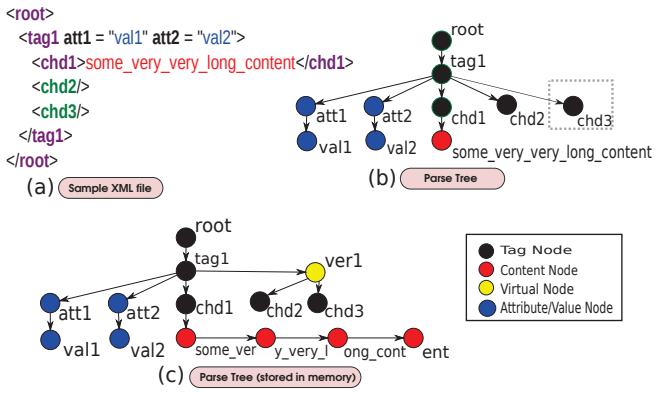

Fig. 1: Sample XML document and corresponding parse tree

Let us discuss the format of the output using the example in Figure 1. For now, assume that the enclosed content in dotted boxes is not a part of the XML file nor its associated parse tree. We presume one top level tag-pair that encapsulates all the rest of the tag-pairs. We refer to this tag pair as the *root* node of the tree. The *root* has one child tag, which is *tag1*. We refer to each such node as a tag node. *tag1* has two attributes, *att1* and *att2*, and two child tag-pairs, *chd1* and *chd2*. We create a node for each attribute, and each child tag-pair, and attach it as a child to the parent node. Let us now consider the first child, *chd1*, which has some content (no children).

We create a linked list of content nodes to store the content. Each content node in our example can store 8 characters, and thus we create a list of 3 content nodes to store the content for the tag node *chd1*. Using linked lists to store data helps us manage storage space efficiently. Let us now consider the content in the boxes with dotted lines. We add an extra child *chd3* to the node *tag1*. In principle, each tag node in an XML file can have any number of child tag-pairs. However, we need to have a limited amount of space in each tag node for child pointers. We assume a maximum of 4 nodes. We thus need to use a similar linked list based approach as we had done to store content. We create a virtual node, *ver1*, that is a child of *tag1*. All the remaining children of *tag1* will now be added to the sub-tree rooted at *ver1*.

We thus have the following kind of nodes in our tree: tag, content and virtual. This is a standard approach and is also a part of the DOM standard, and such virtual nodes are called *vnodes*. Any tree traversal needs to be aware of virtual nodes and not interpret them as regular nodes. Furthermore, it needs to adjust the depth information (distance from the root) correctly, and take virtual nodes into account. These are straight forward modifications.

Here, are the list of fields in each node (assigned a location in memory). A *token type* field specifies one of 6 types: start tag name, end tag name, attribute name, attribute value, empty tag and content. The *depth field* indicates the node's distance from the root node. The node's content is stored as an 8 byte packet in the *content* field. Pointers to the node's children are stored as a set of four pointers. As previously discussed, we may need to create linked lists of content or virtual nodes. Each node has fields containing the address of the *next content node* and the *next virtual node*, which contain valid memory addresses (if there is a need). We also have a 2-bit *HBT (head body tail)* field to interpret the nature of nodes. For example, if a tag node is a head, then it means that it is not a virtual node. However, if it is a body or tail, then it means that it is a virtual node. We use another set of HBT flags for content nodes also.

B. Overview of the Architecture

The Lexical Analysis Stage(LAS) verifies data at the byte-level and ensures the conformity of the characters to existing standards (see Figure 2). Additionally, it marks the location of the delimiter characters such as '<', and '>'. Subsequently, the Token Extractor Stage (TES) tokenizes the inputs and creates start tag, end tag, attribute name/value, and content tokens. After checking them, we create nodes (data structures in memory) in the Node Creation Stage. This stage provides its output to two stages: the Virtual Content Node Stage (VCNS) creates a linked list of content nodes as described in Section III-A, and the Tree Constructor starts building the XML parse tree. This stage embeds a Virtual Node Constructor Stage that creates virtual nodes.

C. Lexical Analysis Stage (LAS)

A simple approach to this task involves a sequential byte-by-byte traversal as shown in Figure 3. This approach marks all the positions containing delimiters, and this requires N steps for N characters. Now, in HPXA, we mark the location of each delimiter in the 16 byte array, A, using a 16-bit bit vector, where the i^{th} bit being set indicates that the byte $A[i]$ is the end of a delimiter character (see Figure 4). Note that the 1s in the bit vector are sparsely located (interspersed with 0s). We wish to create an array (Arr_pos (sorted position array) in Figure 4) that just contains the positions of these 1s. This

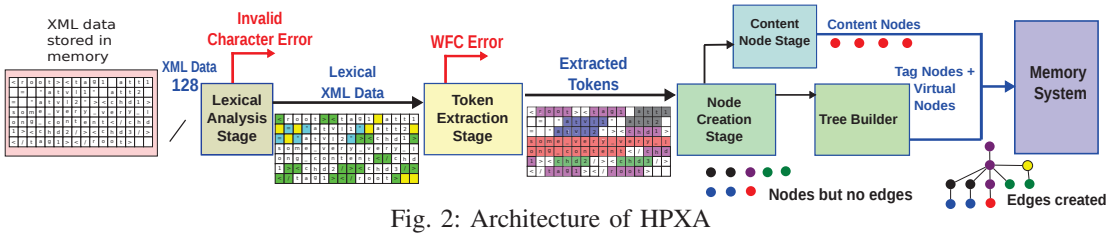

Fig. 2: Architecture of HPXA

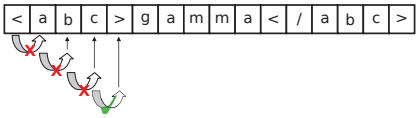

Fig. 3: Slow sequential search

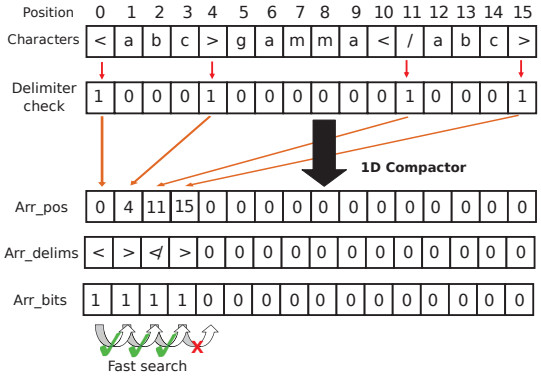

Fig. 4: Parallel delimiter search in HPXA

will help us locate the ends of tags very easily. We create a new piece of hardware to solve this problem, which we refer to as a *compactor*.

The compactor contains 4 units. We break the 16-byte set, into four consecutive blocks, where each block is assigned to an unit of the compactor. In each unit, we use a very small lookup table to populate a 2-entry 1D array with the positions of the last bytes of any tags that it may find. The next step is to merge the outputs of the 4 units. We use a tree structured hardware that combines the outputs of the units in a hierarchical fashion.

D. Token Extraction Stage

We can have a maximum of 6 tokens per cycle. Each token is processed by a token extractor unit. This unit looks up a pair of successive delimiter pairs in the arrays (output of LAS), and reads all the bytes between them. Let us refer to these group of bytes as *chunks*. At this point, we have extracted three kinds of chunks: (tag name + several attribute name/value pairs), end tags, and content. The end tags and the content chunks are valid tokens. Note that these chunks can easily be identified based on the values of the delimiters that enclose them. Now, there is a need to further break the chunk of bytes that contains the start tag name and attribute/value pairs. For doing this, we pass this chunk through a block of specialized hardware that starts out with marking the locations of the ' ', '=' and '"' characters. We then use a compactor to get their exact locations. Then, we check if the start tag name is valid using well-formedness rules extracted from the XML 1.0 specification. We then proceed to extract the attribute name/value pairs. It is easily possible to do that because we know the locations of their beginning and end (from the compactor). At this point, we check these new tokens for well-formedness, and try to mark duplicates. These new tokens are then merged with the previously generated tokens (end tag and content) to form a single token stream.

To merge these streams, we do the following. Before splitting we record the relative order of the chunks using a counter. The other end-tag and content chunks are buffered till all later chunks containing start tags are fully tokenized. We then use this counter to reorder the tokens.

E. Node Creation Stage(NCS)

This stage receives the following tokens: start tag, end tag, content, empty tag, attribute name/values. Based on the token types, we create a data packet for the corresponding node. We have a stream of nodes now, which need to be assigned a depth level. The *depth level* is defined as the distance of a node from the root of the parse tree (root has a depth of 0). We have a depth processor (one for each of the 6 possible nodes arriving in a cycle). We create a lower triangular matrix, R, where row i corresponds to the i^{th} node in the node stream (the earliest node is numbered 1). The i^{th} column for a start and end tag are assigned value V_i, which can take either of three values: +1 in case of start tag, -1 in case of end tag, and 0 as the default value. For attribute and content nodes we assign 1 to R_{ii}, and the rest of the columns are assigned 0. The depth of any node, i, is the sum of the existing depth of the latest node in the last batch plus $\sum_{j=0}^{j\leq i} R_{ij}$.

F. Parse Tree Builder

HPXA aims to construct the tree in a single pass, and in a depth-first manner. Once the memory location and depth of a node have been calculated (at the end of the Node Creation Stage), the final task required to complete tree construction is to create an edge between every node and its parent. It consists of the following operations: (1) identify the parent, (2) add the current node's memory address in the parent's node data structure. A traditional one-node-at-a-time approach uses a last-in-first-out stack. When we encounter a start tag we push the memory address of its node on the stack, and when we encounter an end tag we pop the entry. The top of the stack is always the parent of any tag nodes that we read. This approach needs to be parallelized. We assume a stack with 8 entries. In every cycle, we will have up to 6 nodes coming,

and all of them might need to access (peek, push, and pop) the stack. Each node might find its parent either in the current batch of nodes, or at the top of the stack. For each node, we have a *node processor* (logic unit), whose task is to find its parent. From the point of view of the stack, it needs to find its new state that it will reach at the end of the current set of operations. We have 8 *stack processors* to find the next state for the stack (one for each stack entry).

1) Node Processor: It searches for the latest node with a depth $(i-1)$ (i is the depth of the current node). It scans the previous nodes in the current batch of nodes, and then scans the stack (top down). Then a priority encoder is used to find the matching parent node. This process gives us the id, location, and memory address of the parent. Note that this operation is conducted in parallel for 6 nodes at a time.

2) Stack Processor: The i^{th} stack processor looks at all the nodes that are above the current node (towards the stack top), and all the nodes in the current batch of nodes. There are three choices: (1) either the node is popped, (2) it remains as is, or (3) its content changes. If there is any entry in the new batch of nodes with a depth lower than the depth(i) of this entry, then the node will get popped (situation (1) or (3)). Else, we have situation (2). We differentiate between (1) and (3) as follows. If the depth of the last node in the current batch of nodes is lower than i, then the node is popped(situation 1), otherwise we have situation (3). Now for situation (3) the node that will occupy this level is the latest node at depth level i in the current batch of nodes. For situation (3) we create a 8x6 matrix (8 entries in the stack, and max. 6 nodes per cycle), A. A_{ij} is 1 if node j has a depth level i. From each row, we select the j^{th} node such that $A_{ij} = 1$, and j is maximized. We have a similar 8x8 matrix for the pop operations.

3) Creation of Edges and Virtual Nodes: We have 14 edge processors – one for each stack level, and one for each node in the current batch of nodes. Each edge processor collates all the children of a node, and creates tree edges. It uses a compactor to identify and linearly arrange a node's children. The pointers to the children are arranged as an array in each node. In the case that a node has more than 4 children, the edge processor immediately creates a new virtual node, and adds the child nodes to the subtree of the newly created virtual node.

We use a small coalescing write buffer (16-entry, 64 byte line size, fully assoc.) to reduce the bandwidth to memory. Once all the 4 nodes in a 64 byte line are fully written (all their fields are updated), the block is ready to be written to memory. We assume a multibanked SRAM memory that can accept 2 blocks per cycle.

IV. Evaluation

A. Setup

The code for HPXA was written in Verilog (version 16.20), and was compiled and synthesized on the Cadence RC compiler for the UMC 28nm technology node. The final design had 25 pipeline stages. We used a standard set of 9 XML benchmarks from the Univ. of Washington XML data repository [7]. These files have up to 5 nesting levels, and significantly vary in terms of size, the number and nature of attributes/tags. We dumped their hex-code in a file arranged as 16 byte chunks, and provided this file as an input to our parser. We gave a global clock signal as input along with the reset and start signals. The active low reset signal resets all the pipeline registers, FSM state and all previously active buffers. The start signal lets HPXA know when to start reading the XML data and start the parsing operation. To calculate the CPB values, the number of global clock cycles is calculated from the time the start signal goes high till all the tokens are extracted.

We ran all our experiments on a system with four Intel i5 cores running at 3.3 GHz, 8 GB Ram, and Ubuntu Linux 14.04.

B. Results

Using the Cadence RTL Compiler, the area for HPXA was calculated to be $33066\mu m^2$, with a maximum clock frequency of 833 MHz for a 28 nm technology node.

Benchmark	File Size	Element count	Depth	Cycles taken	CPB
region	787 B	21	3	87	0.1016
nation	4.47 KB	126	3	316	0.0689
Ubid	19.7 KB	342	5	1300	0.0639
321gone	23.8 KB	311	5	1563	0.0637
Yahoo	24.6 KB	342	5	1611	0.0637
supplier	28.5 KB	801	3	1859	0.0635
Ebay	34.6 KB	156	5	2253	0.0633
reed	277 KB	10546	4	17,751	0.0625
UWM	2.3 MB	66729	5	146,126	0.0625

TABLE II: CPB values for benchmarks

Table II shows the performance of the HPXA accelerator for different benchmarks. We observe that with increasing file sizes (beyond 200 KB), the CPB value stabilizes to 0.0625 (1/16), and there are almost no stalls in the pipeline (can only be caused by the memory system).

V. Conclusion

In this paper, we designed HPXA, which is a highly parallel XML parser. Unlike prior work, it can process 16 bytes of data at one time. As a result our mean CPB is at least 4X more than that of competing work and our XML processing throughput is 6-25 times higher.

References

[1] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel, "Hardware acceleration in the ibm poweren processor: Architecture and performance," in *PACT*, 2012.

[2] J. Van Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson, "Xml accelerator engine," in *Workshop on High Performance XML Processing*, 2004.

[3] W. Lu, K. Chiu, and Y. Pan, "A parallel approach to xml parsing," in *ICGC*, 2006.

[4] Z. Dai, N. Ni, and J. Zhu, "A 1 cycle-per-byte xml parsing accelerator," in *FPGA*, 2010.

[5] F. El-Hassan and D. Ionescu, "Scbxp: An efficient hardware-based xml parsing technique," in *SPL*, 2009.

[6] M. Jianliang, S. Zhang, T. Hu, M. Wu, and T. Chen, "Parallel speculative dom-based xml parser," in *HPCC-ICESS*, 2012.

[7] "Uw xml data repository," http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html, [Online; accessed 29-August-2017].