A 1 Cycle-Per-Byte XML Parsing Accelerator

Zefu Dai, Nick Ni, Jianwen Zhu
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada, M5S 3G4
{zdai|yni|jzhu}@eecg.toronto.edu

ABSTRACT
Extensible Markup Language (XML) is playing an increasing important role in web services and database systems. However, the task of XML parsing is often the bottleneck, and as a result, the target of acceleration using custom hardware or multicore CPUs. In this paper, we detail the design of the first complete field programmable gate array (FPGA) accelerator capable of XML well-formed checking, schema validation, and tree construction at a throughput of 1 cycle per byte (CPB). This is a significant advancement from 40 CPB, the best previous reported commercial result. We demonstrate our design on a Xilinx Virtex-5 board, which successfully saturates a 1 Gbps Ethernet link.

Categories and Subject Descriptors
B.4.1 [**Input/output and Data Communications**]: Data Communication Devices – *processor.* I.7.2 [**Document and Text Processing**]: Document Preparation – Markup languages

General Terms
Performance, Design, Experimentation, Languages.

Keywords
XML Parsing, Schema Validation, Tree Construction, DOM, Bloom Filter, BART, String Comparison, Ethernet.

1. INTRODUCTION
Extensible Markup Language (XML) has become a standard for data representation and exchange. It is prevalent in a wide variety of applications like web services, database systems, content-based routing, and scientific applications, thanks to its platform-independence, interoperability and flexibility. As a result, XML processing has become an important workload for web servers, database servers, etc. However, XML parsing consumes a significant portion of execution time of web servers, and has become a threat to database performance [5].

XML parsing consists of three major tasks: *well-formed checking*, which checks the document against syntactic rules, *schema validation*, which checks the document against semantic rules, and *tree construction*, which builds the in-memory data structure for further processing. To characterize the performance of XML parsers, the metric of cycle per byte (CPB) is often used. Similar to cycle per instruction (CPI) found in computer architecture, CPB counts the average number of cycles used to process each byte of XML document. Since it is independent of the clock frequency, whose scaling can be arguably enjoyed by all platforms, it is a preferred figure of merit for achieved parallelism of a design.

Current commercial software XML parsers, such as libxml, Xerces and XML4C, can only achieve a best processing rate of 40 CPB on tree construction and 70 CPB on schema validation [4][5][11][23]. A large array of research results have been reported, which often exploit the SIMD instruction set extension of CPUs, or multicore CPUs to speed up XML processing in software [9][13][14][16]. However, their results are often incomplete, e.g. with result only on well-formed checking. While the leading IT companies such as IBM, Intel, HP and Dell offer hardware-accelerated solutions to different XML processing tasks, neither performance metric nor design detail was revealed. The latest commercial result of a full ASIC-based XML accelerator, presumably with highest performance, achieves well-formed checking of 10 CPB, schema validation of 40 CPB, and tree construction of 20 CPB [18].

In this paper, we present a high performance XML Parsing Accelerator (XPA) capable of performing all thee tasks at 1 CPB. More specifically, we make the following contributions: First, we identify recurring computational idioms in XML processing, and devise corresponding *hardware structures* to achieve efficiency. Second, we devise a *speculative pipeline structure* such that tree construction can be initiated before validated. Third, we devise a *skewed pipeline structure* in which it achieves high throughput under the common case where the XML document being parsed is correct, and stalls the pipeline for long latency operations only under non-common cases. Last but not the least, we detail the design of a complete hardware accelerator, which to the best of our knowledge, has not been found in the literature. Although our design has employed many techniques reported elsewhere in other contexts, we believe a synthesis of these techniques to achieve a record performance milestone is valuable to the community by itself.

We believe our contributions are particularly relevant to FPGAs in addition to the fact that our design is demonstrated on an FPGA platform. First, we took advantage of the availability of on-chip memory resources and bandwidth, as well as the availability of network IOs and intellectual properties. Second, as web services evolve at a fast rate, FPGAs present an inherit advantage over ASICs due to its field programmability. Our results show that by architectural and design innovations, FPGAs implementation can outperform existing ASICs. Combined with the fact that web services belong to the low volume infrastructure market where FPGAs have the economic advantage, we hope our contributions

FPGA'10, February 21-23, 2010, Monterey, California, USA.

make a case that XML processing is a promising area for FPGAs to win more sockets and expand more market.

The rest of the paper is organized as follows. In Section 2, we review the related work. In Section 3, we describe some background information about XML. In Section 4, we describe our key ideas. In Section 5, we detail our design. In Section 6, we discuss our experimental result.

2. Related Work

Two styles of XML parsers are involved, depending on if an in-memory data structure is constructed for later "random access". The popular style builds the Document Object Model (DOM) [3] tree, a standard data structure for web processing. The less popular, but faster style, called Simple API for XML (SAX), relies on the fact that the XML can be processed by later stages in the same order they are transmitted. It is therefore less flexible and of limited use.

The software community reported many implementations of XML parsers, with varying styles and compromises. In 2004, Zhang *et al*. developed the VTD-XML (Virtual Token Descriptor) parser [10]. They employ the concept of binary XML to avoid performance bottleneck of XML parsing, and achieves a performance of 20 to 27 CPB on tree construction and schema validation. However, binary XML is not an industry standard and their parsed data can't be used by other XML applications directly. In 2006, Lu *et al*. presented a parallel approach to XML parsing [9]. Their technique uses a light weight XML parser to build a skeleton of the XML document in a first pass parsing to guide the partition of the document into chunks that can be processed independently on different threads. Using this technique, the parser achieves tree construction performance of 30 CPB on a 4-core processor. However, the extra skeleton building process, done sequentially, may become a performance bottleneck. In 2006, Kostoulas *et al*. presented a schema-based XML parsing technique named XML Screamer [14], which improves the performance by schema-dependent compilation and tight integration across layers of software. The parser achieves a performance of 22 to 43 CPB on SAX parsing and schema validation. However, for each different type of XML documents, a new parser needs to be generated. In 2008, Cameron *et al.* developed an open-source non-validating XML parser Parabix (parallel bit streams for XML) which exploits the SIMD capabilities of modern-day commodity processors to process multiple characters at the same time, achieving performance of 6 to 15 CPB on SAX parsing [13]. However, no in-memory tree data was built and schema validation was not implemented.

In the hardware community, Lunteren *et al*. proposed in 2004 an approach to build an efficient and scalable general purpose state machine for accelerating XML processing [4]. However, no full system was demonstrated. In 2007, Moscola *et al.* presented a technique to automatically map regular expressions directly onto FPGA hardware and implemented a simple XML parser for demonstration [7]. Their technique could be useful but not sufficient to solve all problems since XML syntax rule is not a regular language. In addition, hardware recompilation is required each time it is applied to a different type of XML documents. In 2008, Krishnamoorthy presented a hardware XML parser [6], which constraints on the length and types of tokens. In 2009, Leventhal *et al.* presented an ASIC-based XML Accelerator, which achieves performance of 20 CPB on tree construction and 40 CPB on schema validation [18]. In addition, there are a number of commercial products provided by the leading IT companies, such as IBM's WebSphere DataPower XML accelerator XA35 [26], however neither performance metric nor design detail was revealed.

The achieved performance of previous work, along with our proposed design, is summarized in Table 1. ('?' means the data is not reported, and '-' means not implemented).

Table 1. XML parser performance (CPB) comparison.

Techniques	Well-formed checking	Tree construction	Schema Validation
Zhang [10]	?	20-27	20-27
Lu [9]	?	27	33
Kostoulas [14]	22-43	-	22-43
Cameron [13]	6-15	-	-
Leventhal [18]	10	20	40
MIT-libxml [24]	?	64	71
XPA	1	1	1

3. Background

XML parsing consists of three major tasks: well-formed checking, schema validation and in-memory data construction. Other XML applications including XSLT, XPATH, XQuery are based on the results of these 3 basic tasks.

3.1 Well-formed Checking

The task of Well-formed Checking is to perform syntax checking on XML documents to ensure that it conforms to XML syntax rules provided in XML specifications [1]. A sample XML document is shown in Figure 1.

The content of the document is organized in a tree structure with a unique root. Each element is delimited by an opening ('<>') and a closing tag ('</>') and may contain multiple attributes delimited by a space.

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!-- this is an example xml document -->
<University>
  <Department name = "ECE">
    <Students>
      <freshman>310</freshman>
      <sophomore>298</sophomore>
      <junior>213</junior>
      <senior>178</senior>
      <graduate>86</graduate>
      …
    </Students>
    <Professors>
      <professor name="Mike" field="network"/>
      …
    </Professors>
  </Department>
  …
</University>
```

Figure 1. A sample XML document. *"University" is the unique root element. "Department" is an element name which contains attribute name called "name" and attribute content "ECE". Text between the opening and closing tag of an element is called the content of the element, which can be either child elements or simply plain text.*

A well-formed checker scans characters of an XML document, checks if the characters are valid, extracts tokens from scanned

characters and perform syntax checking on the extracted tokens. Syntax rules include a) the opening tag of an element must match its closing tag; b) an attribute name must be unique within its parent element; c) element tags must be properly nested.

3.2 Schema Validation

Due to the flexibility of user-defined markups in XML, servers commonly only accept specific type of XML documents that conforms to set of rules described in certain formats: DTD (Data Type Definition) or its successor XSD (XML Schema Definition) [2]. An example of XSD file, which itself is an XML file, is shown in Figure 2.

A schema validator needs to interpret XSD files and to apply the rules to the tokens extracted by WFC processor. The challenge of schema validation is to select the correct rule to apply to each token out of a set of candidates as well as the token content validation against the selected rules.

```
<?xml version =“1.0”?>
<xs:schema xmlns:xs=“http://www.w3.org/XMLSchema”>
  <xs:element name=“University”>
    <xs:complexType>
      <xs:element name=“Department”  minOccurs=“2” >
        <xs:complexType>
          <xs:sequence>
            <xs:element name=“Students”>
                <xs:complexType>
                  <xs:all>
                    <xs:element name=“freshman” type=“xs:string” />
                    <xs:element name=“sophomore” type=“xs:string” />
                    <xs:element name=“junior” type=“xs:string” />
                    <xs:element name=“senior” type=“xs:string” />
                    <xs:element name=“graduate” type=“xs:string” />
                  </xs:all>
                </xs:complexType>
              </xs:element>
            <xs:element name=“Professors” type=“professorType”/>
          </xs:sequency>
        </xs:complexType>
      </xs:element>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2. A sample XML Schema Definition (XSD) file. *Element “University” is defined as complexType that is only allowed to have “Department” as its child. “Department” requires “Students” and “Professors” in the order. Finally, “Students” may contain “freshman” to “Graduate” in any order.*

3.3 In-memory Data Construction

Given that the size of an XML file can be very large, the DOM representation, which captures the parental relationship between elements and attributes, or nodes, must be stored in DRAM. Such tree data structure requires extra headers with pointers to connect parent, sibling and child nodes. Not only does this require extra memory footprint, but also non-uniform memory access caused by updating previously written memory locations to connect a new node to rest of the tree. Such accesses might cause DRAM page crossing and degrade performance.

4. Key Ideas

In this section, we first identified several recurring computational idioms (fondly referred to as dwarfs in recent literature [21]). Not surprisingly, in the context of XML processing, these idioms are all related to the processing of strings. Isolating these idioms allow us devise or choose efficient hardware structures to implement them. We then describe the key architectural decisions by refining a familiar, baseline architecture, which ultimately leads to the 1 CPB performance target.

4.1 Recurring Idioms

4.1.1 One-to-one String Match

This idiom tests if a subject string equals to a reference string.

Example 1. *During well-formed checking, the syntax rules require that opening and closing tags of an element be matched, the root element must be unique within the document and elements should be properly nested. This implies that the opening tag of each element needs to be compared with the root element, and each closing tag needs to be compared with the last opening tag.*

Due to the fact that the reference string is known at time of input, the commencement of matching task need not wait until the subject string is present in its entirety. Instead, the matching can be executed in a streaming fashion. This not only achieves the best latency, but also scales well on strings with large, variable length due to its minimal requirement of storage.

4.1.2 One-to-many String Membership Test

This idiom *tests* if a subject strings equals to any member of a set of reference strings.

Example 2. *There are rules in both well-formed checking and schema validation that require an element/attribute name or its value to be unique within a certain range. This is equivalent to ask if an incoming element/attribute name matches with one of the previously seen names.*

In general, performing such tests require string comparison of all reference strings, which can be prohibitively expensive. However, the number of full comparisons can be reduced if one can filter out “obvious” cases, where a simple test can determine that an incoming string does not belong to the set. We employ the concept of Bloom Filter, which defines a set of independent hash values for each reference string. The set of reference strings is then approximated by a bit vector where the corresponding bits of all hash values of the reference strings are set to ‘1’s. If the hash values of the subject string produce a new ‘1’ in the bit vector, then we can conclude that the subject string does not belong to the set.

4.1.3 One-to-many String Search

This idiom finds a subject string among a set of reference strings. Note that while seemingly similar, the previous idiom only needs to return a binary answer, whereas this idiom effectively performs a lookup into an associative array (dictionary) of strings.

Example 3: *During schema validation, each element or attribute needs to search for its corresponding schema rule among a set of candidates.*

This idiom is commonly implemented as a hash table in software. We employ the BART scheme [8], originally proposed in the context of network routing table lookup. Unlike software hash table implementation where the lookup time can be undeterministic in the presence of hash value conflict, the BART scheme guarantees that the number of conflicts is bounded to a

predefined value. Therefore, a string search amounts only to an on-chip memory access and parallel comparisons of bounded size.

4.2 Key Architectural Decisions

Before describing our architectural decisions, it is instructive to describe a naïve baseline architecture shown in Figure 3. The architecture mimics a textbook decomposition of compiler frontend, which suffer from poor performance even when the individual blocks are pipelined. First, the number of pipeline stages is large, leading to long latency in processing. Second, blocks have diverse worst case, leading to poor overall throughput. In the sequel, we describe architectural techniques to improve the baseline architecture.

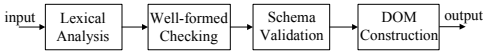


Figure 3. Conventional serial XML parsing.

4.2.1 Speculative Pipeline

While a compiler usually constructs a syntax tree only after it passes correctness check, we choose to construct DOM tree immediately after lexical analysis, as shown in Figure 4. This is speculative since we may construct a tree only later to find out invalid. Although in this case the tree has to be discarded, this mechanism allows the DOM tree construction stage to run independently of well-formed checking and schema validation stage, thereby significantly reducing the latency of the accelerator.

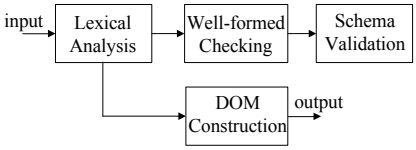


Figure 4. Hardware speculation XML parsing.

4.2.2 Multi-rate Pipeline

Well-formed checking and schema validation are different in processing rate and granularity. Well-formed checking performs the syntax checks on each single character, while schema validation validates the semantics of extracted token flow. Well-formed rules are simpler compared to schema rules. To achieve a balanced pipeline design, we device a 3-level multi-rate pipeline structure as shown in Figure 5.

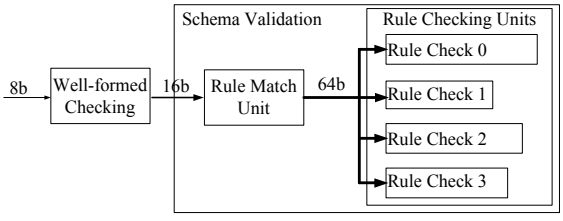


Figure 5. Multi-rate pipeline structure.

In the first level, the well-formed rules are checked against each character. In the second level, a rule math unit inside the schema validation stage search for the corresponding schema rule for each token, which is hashed into a 16 bit integer. In the third level, the rule checking units perform checking on multiple bytes of data simultaneously, such that they have multiple cycles of time budge to achieve the same throughput as the other stages.

4.2.3 Common Case Optimized Stallable Pipeline

A pipeline stage can have dynamically different latencies. If following the regular, static pipeline design, then we have to use the worst case latency as criterion to advance the pipeline. We employ a skewed, stallable pipeline structure where under the common cases, each pipeline stage is designed to have latency of one. They are stalled to carry out long latency computations only under uncommon cases. More specifically, we exploit the observation that in most cases, the XML document being parsed is a valid document. For example, in implementing the membership test idiom, we use the Bloom filter to detect the majority of common cases where the string uniqueness requirement is satisfied. Under these cases, no further test is needed and the pipeline can be advanced. Only in rare occasions where Bloom filtering fails, a long-latency string comparisons is invoked, in which case the pipeline is stalled.

4.2.4 High-bandwidth On-chip Data Structure

To perform schema validation, many rules have to be checked against an XML construct under parsing. Typically, the types of checks need to be encoded in memory. To reduce latency, it is desirable to parallelize the rule checking, which dictates that the encoded rule information needs to be accessible in parallel.

FPGAs offer very large bandwidth on-chip memories. We devised a custom schema rule representation. The schema rules are divided into three portions and distributed into three local memories. Each memory has a wide data bus, allowing a single-cycle access of all schema rules associated with the XML construct under validation.

4.2.5 Final Architecture of the XPA

The final architecture of the XPA as a result of above decisions is shown in Figure 6. The lexical analysis stage is merged into well-formed checking stage, since some well-formed rules are also checked during lexical analysis.

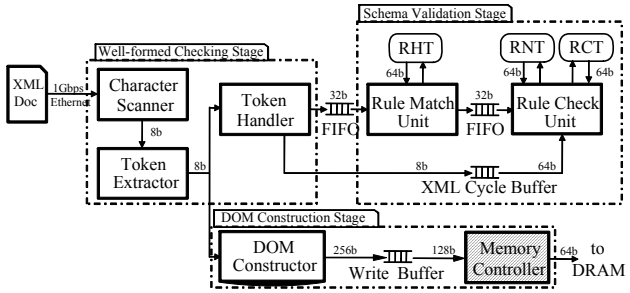


Figure 6. Top level diagram of the XPA.

The entire design has three stages: well-formed checking stage, schema validation stage and DOM construction stage. Functional units in well-formed checking stage scan XML character streams from the Ethernet, extract tokens from the streams and perform syntax rules checking on the tokens. In schema validation stage, the Rule Match Unit searches for corresponding schema rule for each valid token. The Rule Checking Unit is configured according to the schema rules and rules checking are performed on the content of each token. The schema rules are distributed into three local memories: Rule Header Table, Rule Name Table and Rule

Content Table. FIFOs are used between units to accommodate processing rate changes. The DOM construction stage runs in parallel with the schema validation stage and the well-formed checking stage. Tokens extracted from input streams are fed immediately into DOM construction stage to generate tree data, and written into DRAM memory through a memory controller.

5. Design

This session will present the detailed implementation of each functional unit of the XPA.

5.1 Well-formed Checking Stage

5.1.1 Character Scanner Unit

The Character Scanner Unit retrieves data from the Embedded Ethernet MAC (EMAC), and outputs data byte by byte to the next unit in the XPA. The block diagram of Character Scanner Unit is shown in Figure 7.

A 1Gbps PHY is connected to the Embedded MAC through a SGMII interface. We implemented a simple UDP receiving logic block to deliver the incoming packet payload sent from host PC into the parser. In addition, a 1KB asynchronous FIFO is used to bridge the different clock domains between the Character Scanner Unit and the next cores.

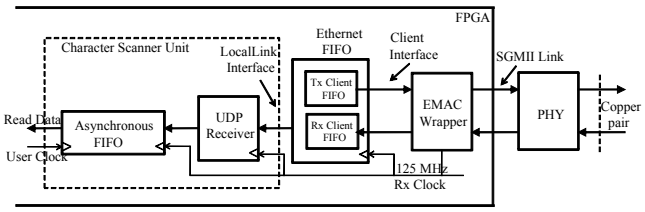
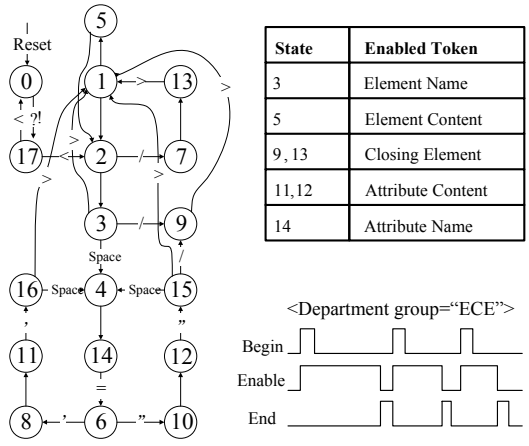


Figure 7. Block diagram of the Character Scanner Unit.

5.1.2 Token Extractor Unit

The Token Extractor Unit is responsible for recognizing all the tokens from the input stream. It is implemented as a finite state machine that makes state transitions on valid input characters.



State	Enabled Token
3	Element Name
5	Element Content
9, 13	Closing Element
11,12	Attribute Content
14	Attribute Name

Figure 8. The FSM and output waveform of the Token Extractor Unit.

In contrast to software parser states, the goal of our finite state machine is not to perform the entire well-formed checking but to extract the tokens and output their types as well as the position signals as "begin", "enable" and "end". The finite state machine and sample signal behavior are shown in Figure 8. The core well-formed checking functions are then executed in the Token Handler Unit.

5.1.3 Token Handler Unit

The Token Handler Unit performs a series of operations on each token extracted by the Token Extractor. Main operations include: A) Checking the correct nesting of each element, and the uniqueness of root element name. B) Checking the uniqueness of each attribute name within every element. C) Generating information of type, length and hash code for each token, passing them down to schema validation stage through FIFO. D) Storing useful characters into XML Cyclic Buffer for schema validation. The first two tasks are described in details.

5.1.3.1 Element Name Correct Nesting Checking

To check the correct nesting of each element, the closing tag of each element needs to be compared with the last opening tag. As described in section 4.1.1, the comparison is carried out on each input character. This task is done with the help of an Element Name Stack. Whenever an element opens, its name is pushed into the Element Name Stack character by character. When it is being closed, one character is popped from the Element Name Stack per cycle and compared with the incoming character. Because the element tags are required to nest properly, a mismatch in the input character of closing element with the output of Element Name Stack always means a violation. The usage example is shown in Figure 9.

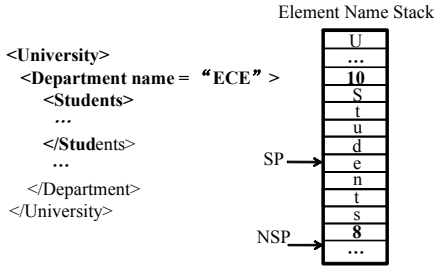


Figure 9. Example of Element Name Stack operation. *When the 'Students' element is being closed, SP starts at 'S' and moves cycle by cycle to '8'. At the end of the matching. The whole element is popped off the Element Name Stack by updating NSP=NSP - 8 -1 and SP=NSP-10 – 1.*

5.1.3.2 Attribute Name Uniqueness Checking

The uniqueness checking requires each attribute name to be compared against multiple preprocessed names. This problem is identified as membership test dwarf in section 4.1.2. We employed the concept of Bloom Filter [19][20] and implemented a 3-stage pipeline for this task as shown in Figure 10. In the first stage, a HashCode Generator generates *k* independent hash codes for each attribute name. In the second stage, the *k* hash codes are used to access *k* different bits in a bit array. In the third stage, the fetched *k* bits are examined whether any bit is '0' (initial value), which means the attribute name is guaranteed to be unique. Once uniqueness is confirmed, all corresponding *k* bit in the bit array are updated to '1' and the attribute name is stored into the Attribute Name Stack. In case, all *k* locations returned '1', it infers potential violation, hence the whole pipeline will be stalled to compare the

attribute name against each strings previously stored inside the Attribute Name Stack, character by character, to remove the false-positive case (Figure 11).

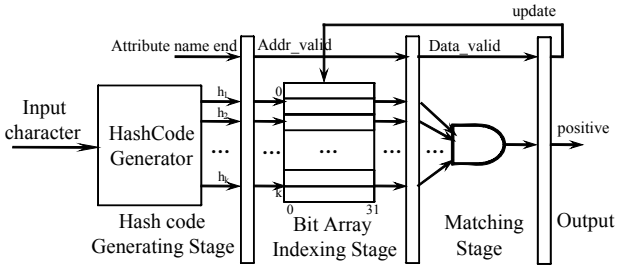


Figure 10. Bloom Filter pipeline for Attribute Name Uniqueness Checking.

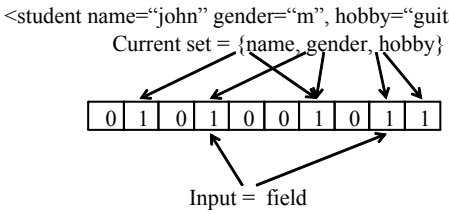


Figure 11. Example of false positive of the Bloom Filter. *There are 3 attributes existing in current test set when parsing token "field" which generates 2 hash codes colliding with "gender" and "hobby". This potential violation turns out to be a false positive in this case.*

5.2 Schema Validation Stage

A valid element/attribute token not only needs to be syntax correct, but also contain its conforming definition in its XSD file in the correct context. Due to the relatively small volatility of schema files, we first pre-compile the current schema file into a custom local memory format that is efficient for lookup. We use three tables to store the contents: Rule Header Table (RHT), Rule Name Table (RNT) and Rule Content Table (RCT), each maintaining the tree structure of every rule, the name of each rule and the rule contents respectively.

5.2.1 Rule Match Unit

The Rule match Unit is responsible for selecting the corresponding schema rule for each element name and attribute name among a set of candidate rules.

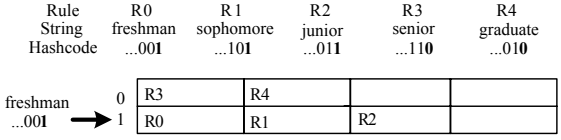


Figure 12. Example of the BART scheme. *From Figure 1 and 2, there are 5 candidate rules when parsing "freshman" element name. By setting P=4, bit 0 can be used as a bit mask to divide into 2 groups with less than 4 members. Each incoming hash code only accesses a row that its bit 0 indexes and guarantees be able to select one rule by doing at most 4 parallel comparisons. In this case, R0, R1 and R2 are compared against input in parallel.*

As discussed in Section 4.1.3, we employed the idea of the Balance Routing Table Search (BART) scheme [4][8]. BART is based on a novel hash function with the special property that the maximum number of collisions for any hash index can be limited by a configurable bound *P*. The hash index is extracted from bit positions within the input hash code, which are selected to realize the maximum collision bound *P*. The value of bound *P* is based on the memory access granularity to ensure that all collisions for a given hash index can be resolved by a single memory access and by at most *P* parallel comparisons. A simple illustration is shown in Figure 12.

The Rule Match Unit consists of a two-stage pipeline where the first stage selects at most *P* rules (We chose 4 for our design) using XORed value of input hash code and a bit mask as index into the Rule Header Table and passes them to the second stage for the parallel hash code comparison. Then in the second stage, one or no rule is selected and outputted to the next module with information on type and length of the token as well as pointers to the Rule Name Table and Rule Content Table (Figure 13). After every selection, the result is used to advance the document context by updating the table address register and bit mask register. One exception is that when the input is a closing element, the table address and bit mask register need to be restored to previous level. To handle the case, a Table Address Stack is maintained to store the history of the two values in a stack fashion.

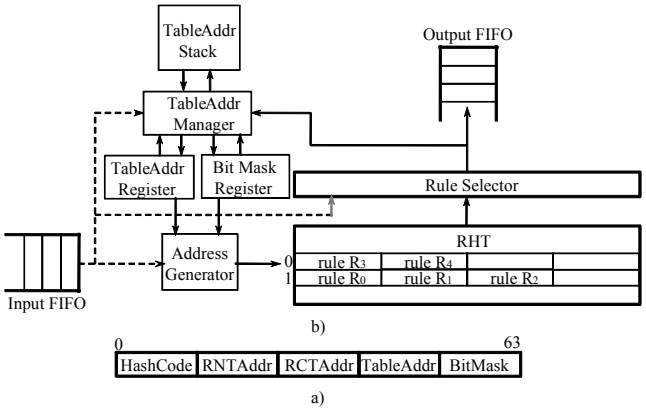


Figure 13. a) Data format of RHT entry b) Block diagram of the Rule Match Unit.

5.2.2 Rule Check Unit

The Rule Check Unit is responsible for the schema validation on the contents. It is further divided into 2 sub units: Rule Name Check Unit and Rule Content Check Unit. Rule Name Check Unit verifies the selected rule from the Rule Match Unit is hash-code-collision error free. The Rule Content Check Unit checks if the contents of elements and attributes conform to the selected rule.

5.2.2.1 Rule Name Check Unit

The logic of the Rule Name Check Unit is shown in Figure 14.

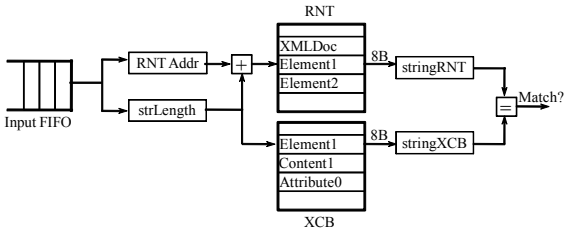


Figure 14. Block diagram of the Rule Name Check Unit.

When a rule arrives from the Rule Match Unit, it starts reading out characters from two different local memories: the XML Cyclic Buffer (XCB), which contains the actual string of the input token pushed in by the Token Handler Unit, and the Rule Name Table pointed by the RNTAddr in Figure 13. Both data are fetched out and compared, 8-byte by 8-byte, to verify the match.

5.2.2.2 Rule Content Check Unit

The Rule Content Check Unit (Figure 15) is responsible for performing schema validation on element and attribute contents as well as checking their arrival sequence. Once the Rule Match Unit finds a rule, this unit fetches the corresponding rule contents (Figure 15 a) and use them to configure four different checking units for next arriving content check. The Rule Content Stack is required to store the history of rule content for similar reason as the Table Address Stack in section 5.2.1. The actual check is again executed in 8-byte by 8-byte manner and is triggered as soon as a valid element content or attribute content appears in the output of the XML Cyclic Buffer. The four checking units implemented in our design are: sequence check, type check, range check and key check.

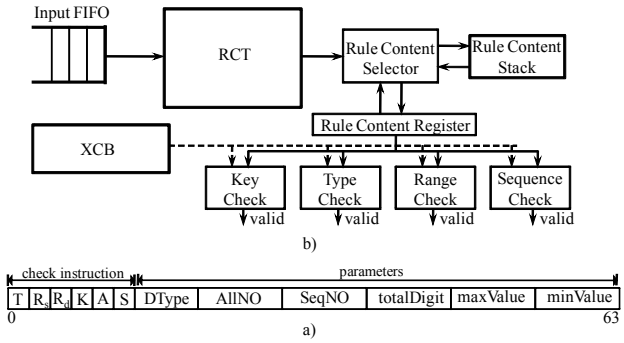


Figure 15. a) Data format of RCT entry. b) Block diagram of the Rule Content Check Unit. T=Type, R=Range, K=Key, A=All and S=Sequence.

The Sequence check block ensures if the sequence of incoming tokens follow the specified order in XSD. (e.g. <*Sequence*> in Figure 2) we use a stack to record the latest sequence number in each level of tree hierarchy and compare against the *SeqNO* field in Figure 15. The Type check block checks if the pattern of the content is correct. We currently support string, integer, decimal and date. The Range check block checks if a content value falls within the allowed range, such as "*minOccur*" in Figure 2. Lastly but not least, the Key check block checks if a token marked as "key" is unique throughout a document. This problem is categorized as the dwarf in section 4.1.2 because each 'key' type value needs to be compared against a previously parsed "key" to verify the uniqueness. The same Bloom Filter approach as described in section 5.1.3.2 is used except that actual strings of the key contents are stored in DRAM instead of local memory as the set could grow over thousands. (e.g. list of student numbers)

5.3 DOM Construction Stage

The DOM Constructor Unit is responsible of building a DOM tree of the input XML document in DRAM, which can then be used to develop a DOM Application Programming Interface. In order to support industry specified efficient tree operations, the base data structure should contain enough pointers in each node such that every part of XML data is tightly connected. In our current design, a simple and straight forward 32-byte aligned data structure is employed to implement the DOM Construction (Figure 16). With the data structure, each element name requires a) as its header and c) to contain its name strings. Each attribute name uses b) as header and c) for its name strings. Contents only use a c) with parent link linked back to their parents.

The DOM Constructor exercises three main tasks, new node allocation, update of parent and update of sibling. When a new token other than closing element is parsed, it allocates a new node in DRAM in an appropriate format. If the token has a previous sibling in the same hierarchy, the *NextSibling* pointer in previous sibling header is updated in the next cycle. When parsing a closing element, the *ChildList* pointer of the corresponding element header is updated if it appears to be a parent of already parsed nodes. In addition, we employed multiple techniques such as a stack to store DRAM addresses of active parent nodes and register last closed element to locally keep track of DRAM addresses for update of parent and sibling respectively. Because the DOM Constructor requires no DRAM read operation, the data structure is optimized for write only data access by reducing page crossing.

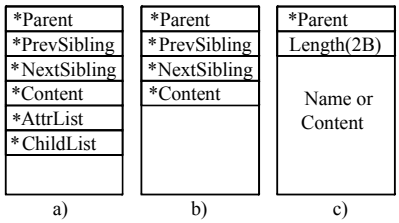


Figure 16. Data structure of nodes: a) Element name header b) Attribute name header c) String data.

6. Evaluation

In this section, we carry out comparative performance study of our design against 4 publicly accessible software XML processors. We use throughput in both CPB and Gbps as performance metrics. In addition, we examine the implementation cost and speed on FPGA, and scalability issues.

6.1 Hardware Experimental Setup

Our design is implemented and tested on Xilinx Virtex-5 XC5VSX50T FPGA on the ML506 evaluation board. To perform the test under a practical environment, we connect the input of the XPA to a Tri-mode Ethernet MAC, configured to work with a 1Gbps SGMII PHY device. A simple UDP receiving protocol is used to extract data and commands from incoming UDP packages. The test files are fed from a laptop to the Xilinx board through a 1 Gbps Ethernet link. The output data of the XPA is written to an on-board 256MB SODIMM DDR2-533 memory module through the Memory Controller (MC). A serial port is integrated to display experimental results. The structure of the XPA test bed is shown in Figure 17.

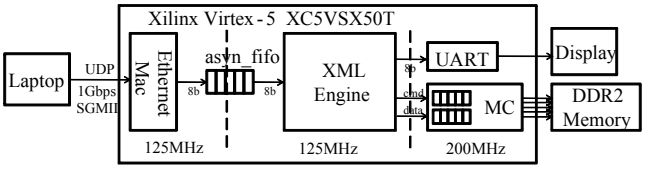


Figure 17. Block diagram of the XPA test bed.

The test bed is separated into 3 clock domains. The Ethernet MAC works with a 125MHz clock in order to cope with 1Gbps wire speed. The XPA communicates with the Ethernet MAC using a 1KB asynchronous FIFO and runs at a different 125MHz user clock. The serial port and the Memory Controller work with a 200MHz clock.

6.2 Software Experimental Setup

To compare the performance of the XPA against other XML processors, we test 4 software XML parsers with the same set of benchmarks. The parsers are chosen from well-known open source commercial tools that have the best reported performance according to [24]. The tests of software XML parsers are carried out under the configurations listed in Table 2.

Table 2. Software experimental setup.

Hardware and software platform	Tested XML parsing libraries
Intel Core 2 Quad Q9300 (2.5GHz, 6MB L2 Cache) 2GB DDR2-800 Memory Debian Linux 2.6.18-6 x86-64 GNU C 4.1.2	Xerces-c 2.8.0 x86-64 Libxml2 DOM4J-1.6 JAVA API for XML Processing (JAXP) 1.6.0

We used the XML Benchmark Tool [23] from Intel to gather performance results for the 4 software XML parsers. For each benchmark, the XML Benchmark Tool will perform multiple iterations of warm-up and test to get stable results such that the overhead of memory load and operation system management are minimized. All the benchmarks are read from local hard drive for software tests.

6.3 Benchmarks

The benchmarks are chosen from different XML projects. Each benchmark contains multiple test files from the same project. The file size varies from 3 KB to 116MB. The benchmarks are separated into 2 groups: DOM parsing benchmarks and schema validation benchmarks. Schema validation benchmarks contain one XSD file for each benchmark. Table 3 lists the names of these projects, the maximum sizes of test files as well as the source of the projects.

Table 3. Information of benchmarks.

Group	Benchmark	XML Size	XSD Size	Source
DOM Parsing	Security	3 KB	-	Intel Corporation
	Structure	12 KB	-	codesynthesis
	Tpox	15 KB	-	tpox
	Hl7	136 KB	-	hl7-testharness
	Qedeq	211 KB	-	qedeq.org
	Xmark	116 MB	-	xml-benchmark
Schema Validation	CustomInfo	1 KB	2 KB	Intel Corporation
	CDCatalog	105KB	2 KB	w3schools
	Workflow	13 KB	10 KB	qedeq.org

6.4 Measurement

6.4.1 Throughput

The detailed test results on performance of different XML processors are presented in Table 4 and Table 5 ('-' indicates that certain functionality is not implemented and thus result unavailable). Table 4 lists the throughput of different tests in Gbps. Table 5 presents the same results in CPB.

As illustrated by Table 4, XPA achieves the raw throughput it is designed for, 1Gbps. The throughput is in fact bounded by the Ethernet link speed. Sine the maximum frequency achieved is 130MHz, which we did not make an effort to further improve, the actual raw throughput can be slightly higher: 1.04Gps. Note that although the software parsers run on processors with 2.5GHz of frequency, XPA is still faster. For parsing benchmarks, which involves only well-formed checking and tree construction, XPA outperforms the best performing software parser (JAXP) by 2.8 times. For the much more difficult validation benchmarks, XPA outperforms the best performing software parser (libxml) by 3.7 times.

Table 4. Results of performance tests (throughput: Gbps).

Benchmark	JAXP	DOM4J	Libxml2	Xerces-c	XPA	XPA_{max}
Security	0.199	0.059	0.294	0.100	1.000	1.040
Structure	0.274	0.110	0.202	0.091	1.000	1.040
Tpox	0.292	0.099	0.264	0.124	1.000	1.040
Hl7	0.415	0.189	0.360	0.128	1.000	1.040
Qedeq	0.481	0.221	0.338	0.133	1.000	1.040
Xmark	0.550	0.256	0.416	0.187	1.000	1.040
Average_par	0.373	0.158	0.314	0.127	1.000	1.040
CustomInfo	0.062	-	0.107	0.054	1.000	1.040
CDCatalog	0.128	-	0.232	0.113	1.000	1.040
Workflow	0.227	-	0.396	0.185	1.000	1.040
Average_vld	0.161	-	0.283	0.134	1.000	1.040
Average_all	0.267	0.158	0.299	0.131	1.000	1.040

As illustrated by Table 5, XPA outperforms other tested software XML processors by more than 66 times in term of CPB, which illustrates the potential of XPA architecture when implemented in ASIC with more aggressive frequency optimizations.

Table 5. Results of performance tests (CPB).

Benchmark	JAXP	DOM4J	Libxml2	Xerces-c	XPA
Security	100.6	339.7	67.9	201.0	1.0
Structure	73.1	181.3	99.1	220.5	1.0
Tpox	68.5	201.3	75.9	161.0	1.0
Hl7	48.2	106.0	55.6	155.8	1.0
Qedeq	41.5	90.4	59.2	150.6	1.0
Xmark	36.4	78.0	48.0	106.7	1.0
Average_pars.	53.6	126.9	63.6	157.2	1.0
CustomInfo	321.8	-	186.2	373.7	1.0
CDCatalog	156.5	-	86.3	176.8	1.0
Workflow	88.3	-	50.4	108.3	1.0
Average_valid.	124.4		70.6	148.8	1.0
Average_all	75.0	126.9	66.9	152.9	1.0

6.4.2 Stall Rate

To further understand the performance of the XPA, Table 6 lists the statistics of pipeline stalls in each parsing stage and memory controller (only the maximum number of stalls for each benchmark is shown). In addition, the average memory bandwidth requirement for each benchmark is also shown in Table 6.

Table 6. Stall rate of the XPA.

Benchmark	Well-formed Checking	DOM Cons-tructor	Schema Valid-ation	Memory Con-troller	Memory Bandwidth Requirement
Security	0	0	-	0	719 MB/s
Structure	0	1	-	0	652 MB/s
Tpox	0	0	-	0	1250 MB/s
Hl7	0	2	-	0	994 MB/s
Qedeq	0	4	-	0	614 MB/s
Xmark	0	1263	-	0	782 MB/s
CustomInfo	0	0	0	0	1030 MB/s
CDCatalog	0	0	0	0	1460 MB/s
Workflow	0	0	0	0	798 MB/s
Average Memory Bandwidth Requirement:					908MB/s

No stall on the well-formed checking stage and schema validation stage is observed. This indicates that common XML documents are not likely to cause a false positive on our Bloom filters.

All observed stalls occur in the DOM construction stage. This is because the DOM constructor often needs to generate multiple

write requests at the same clock cycle on cases when multiple pointers in a DOM tree need to be updated. Normally the extra requests are buffered. When these cases happen too close to each other, the buffer might become full. However, these types of stalls do not happen frequently as illustrated by Table 6: 1263 stalls occur for a 116 MB input file, which contributes to tiny portion of the whole processing time.

No stall is observed on the Memory Controller either, thanks to the large command FIFO deployed in the Memory Controller and the high performance of DDR2 memory. For each benchmark, the memory bandwidth requirement is calculated by counting the number of memory accesses. As shown in Table 6, the average memory bandwidth requirement of all benchmarks is 908 MB/s. Because DDR2-533 memory has a maximum available bandwidth of 4.2 GB/s [25], it is sufficient to consume the memory requests generated by DOM constructor. Therefore, Memory Controller is not likely to generate a stall.

6.4.3 Area and Clock Frequency

The device utilization of our design is shown in Table 7. The XPA accounts for about 2/3 of the total area cost, almost 3 times larger than the Memory Controller. The Memory Controller employs a simple request scheduling algorithm, thus most of its resources are spent on implementing the physical interface to DDR2 memory. The EMAC requires only small amount of logic to implement the packet FIFOs thanks to Xilinx's Embedded MAC. Besides, the XPA uses 13 Block RAM for FIFOs and local memories. However, most of the Block RAMs are configured to be less than 1KB. Thus the actual memory usage is much less than reported. The reported maximum frequency achieved by the design is 130 MHz.

Table 7. Details of device utilization.

Logic Utilization	Slice Register	Slice LUT	Block RAM
XPA	4455 (13%)	6594 (20%)	13 (11%)
MC	1960 (6%)	1683 (5%)	5 (3%))
EMAC	927 (2%)	712 (2%)	3 (2%)
UART	151 (1%)	187 (1%)	2 (1%)
TOTAL	7493 (22%)	9176 (28%)	23 (17%)

6.5 Scalability Study

In this section, we study the sensibility of various design parameters against XML file sizes and characteristics, to ensure the robustness of our design.

6.5.1 Bloom Filter Requirement

The Bloom Filter is one of the key enabling techniques of our design. However, its false positive rate also has great impact on the scalability. Thus it is important to examine the requirements of achieving a low false positive rate.

The false positive rate of the Bloom Filter depends on the size of the tested set n, the size of the bit array m and the number of independent hash functions k. A false positive can be described as the probability of k hashed locations all equal to 1. It can be calculated using following equation as presented in [20]:

$$Rate_{fp} = (1-(1-1/m)^{kn})^{k}$$

To verify the space efficiency of the Bloom Filter, a separate experiment is performed on the Attribute Name Uniqueness test. A set of test files containing a large number of elements, each having a certain number of attributes, are generated. The names of attributes are randomly chosen from popular Google keywords and Wikipedia articles on different subjects. The files are tested on Bloom Filters with different configurations. And results are shown in Table 8.

For every false positive, assume an overhead of 100 clock cycles is needed for doing real string comparison. We hope there are less than 10 false positives, so that the extra cycles can be tolerated by the 1 KB buffer in the Character Scanner Unit. A reasonable test case is when the attribute tokens consist of 25% percent of all the tokens, and each token of any type has an average size of 4 characters. Then a 100 KB file would require a practical false positive rate of:

(number of false positives / number of attribute name tokens) = (10/((100 K/4)*25%))=0.01%

This means that for every 10,000 attribute name tokens there should be less than 1 false positive. Therefore, the test results illustrated by Table 7 show that a configuration of 1-kb bit array with 3 hash functions or 2-kb bit array with 2 hash functions should be practical enough for the Attribute Name Uniqueness test task.

Table 8. False positive rate test results.

m_k	G4x1k	G8x1k	G16x1k	W4x1k	W8x1k	W16x1k
64b_2h	1	66	509	6	129	502
256b_2h	0	5	60	1	8	56
256b_3h	0	0	14	1	3	9
1kb_2h	0	1	6	1	2	2
1kb_3h	0	0	1	0	0	0
2kb_2h	0	0	1	0	0	0
2kb_3h	0	0	0	0	0	0

Table 8 also shows that, increasing the bit array size from 64 bits to 256 bits, the false positive rate of all tests is reduced by 10 times. Besides, by increasing the number of independent hash functions from 2 to 3, the false positive rate is reduced by more than 5 times in most test cases.

6.5.2 On-chip Storage Requirement

In this section, we analyze the scalability of the XPA in term of on-chip memory requirement for processing different sizes of XML files.

The required size of the Element Name Stack is determined by the *depth* of the XML document tree. And the size of the Attribute Name Stack is determined by the largest number of attributes one element has in a document. Both are not likely to scale with XML file size.

The Schema Rule Memory used in schema validation stage consists of the Rule Header Table, Rule Name Table and Rule Content Table. To support variable types of XML documents, the schema Rule memory needs to be large enough to store their XSD files. A typical schema file like XHTML schema requires less than 70 KB. Thus, the storage requirement of the Schema Rule Memory is not likely to become a limit.

7. Conclusion

In the paper, we present an innovative XML processing architecture and design that achieves 1 CPB performance on both tree construction and schema validation with very good scalability. The architecture is implemented on a Virter-5 FPGA board and successfully saturates a 1 Gbps Ethernet Link when running at

125MHz clock frequency. With our demonstration, we believe FPGAs can become a valid contender in winning the enterprise XML processing sockets.

8. Limitations

We acknowledge the following omissions of our design in the interest of time. First, our token extractor does not handle the full UTF-8 character set, and settles only with the ASCII character set. Second, our schema validation does not yet handle full regular expression check. It can be argued that both features are unlikely to be performance bottlenecks, as efficient implementations have been demonstrated elsewhere [7][12][22].

9. ACKNOWNLEDGEMENTS

The authors like to thank the support of National Sciences and Engineering Research Council of Canada, as well as China Scholarship Council for the first author.

10. REFERENCES

[1] Extensible Markup Language, http://www.w3.org/XML.

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

[3] Document Object Model, http://www.w3.org/DOM.

[4] J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, C. Larsson, XML Accelerator Engine, In Proceedings of the First International Workshop on High Performance XML Processing, New York, USA, May 2004.

[5] M. Nicola, J. John, XML Parsing: A Threat to Database Performance, In Proceedings of the 12th International Conference on Information and Knowledge Management, Louisiana, USA, Nov 2003.

[6] R. Krishnamoorthy, Hardware Implementation of an XML Parser, MSc Thesis, Computer Engineering, North Carolina State University, North Carolina, USA, 2008.

[7] J. Moscola, J. W. Lockwood, Reconfigurable Content-based Router using Hardware-Accelerated Language Parser, In the ACM Transactions on Design Automation of Electronic Systems, Vol. 13, No. 2, Apr 2008.

[8] J. v. Lunteren, Searching Very Large Routing Tables in Wide Embedded Memory, In Proceedings of the Global Telecommunications Conference, Texas, USA, Nov 2001.

[9] W. Lu, K. Chiu, Y. Pan, A Parallel Approach to XML Parsing, In Proceedings of The 7th IEEE/ACM International Conference on Grid Computing, Barcelona, Spain, Sept 2006.

[10] J. Zhang et al., VTD-XML: The Future of XML Processing, http://vtd-xml.sourceforge.net.

[11] L. Zhao, L. Bhuyan, Performance Evaluation and Acceleration for XML Data Parsing, In Proceedings of the 9th Workshop on Computer Architecture Evaluation using Commercial Workloads, Texas, USA, Feb 2006.

[12] A. V. Aho, M. J. Corasick, Efficient String Matching: an Aid to Bibliographic Search. In the Communication of the ACM, Vol. 18, No. 6, Jun 1975.

[13] R. D. Cameron, K. S. Herdy, D. Lin, High Performance XML Parsing Using Parallel Bit Stream Technology, In Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research, Ontario, Canada, Oct 2008.

[14] M. G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization, In Proceedings of the 15th International World Wide Web Conference, Edinburgh, UK, May 2006.

[15] M. R. Head, M. Govindaraju, R. v. Engelen, W. Zhang, Benchmarking XML Processors for Applications in Grid Web Services, In Proceedings of the ACM/IEEE Conference on Supercomputing, Florida, USA, Nov 2006.

[16] Y. Pan, Y. Zhang, K. Chiu, Parsing XML Using Parallel Traversal of Streaming Trees, In Proceedings of the 15th International Conference on High Performance Computing, Bangalore, India, Dec 2008.

[17] P. Apparao, R. Iyer, R. Morin, N. Nayak, M. Bhat, D. Halliwell, W. Steinberg, Architectural Characterization of an XML-centric Commercial Server Workload, In Proceedings of the International Conference on Parallel Processing, Quebec, Canada, Aug 2004.

[18] M. Leventhal, E. Lemoine, The XML Chip at 6 Years, In Proceedings of the International Symposium on Processing XML Efficiently. Quebec, Canada, Aug 2009.

[19] A. Kirsch, M. Mitzenmacher, Less Hashing, Same Performance: Building a Better Bloom Filter, In Proceedings of the 14th Annual European Symposium on Algorithms, Zurich, Switzerland, Sept 2006.

[20] B. H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors, In the Communications of the ACM, Vol. 13, No. 7, Jul 1970.

[21] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams and K. A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, Dec 2006.

[22] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, J. Turner, Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection, In Proceedings of the Conference on Applications, Technologies, Architectures and Protocols for Computer Communications, Pisa, Italy, Sept 2006.

[23] Intel XML Benchmark Tools, http://software.intel.com/en-us/articles/xml-benchmark-tool-10-accept-end-user-license-agreement-and-download.

[24] XML Benchmarks, http://xmlbench.sourceforge.net/results.

[25] DDR2 performance. http://ixbtlabs.com/articles2/ddr2-rmma/ddr2-rmma-fsb266.html.

[26] IBM Websphere Datapower XML Accelerator XA35, http://www-1.ibm.com/software/integration/datapower/xa35.