RJ 10311 (A0311-057) November 13, 2003
Computer Science

IBM Research Report

An Evaluation of Binary XML Encoding Optimizations for Fast Stream-Based XML Processing

Roberto J. Bayardo, Daniel F. Gruhl, Vanja Josifovski, Jussi Myllymaki
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing

Roberto J. Bayardo
IBM Almaden Research Center
bayardo@alum.mit.edu

Daniel Gruhl
IBM Almaden Research Center
dgruhl@us.ibm.com

Vanja Josifovski
IBM Almaden Research Center
vanja@us.ibm.com

Jussi Myllymaki
IBM Almaden Research Center
jussi@us.ibm.com

ABSTRACT

This paper provides an objective evaluation of the performance impacts of binary XML encodings, using a fast stream-based XQuery processor as our representative application. Instead of proposing one binary format and comparing it against standard XML parsers, we investigate the individual effects of several binary encoding techniques that are shared by many proposals. Our goal is to provide a deeper understanding of the performance impacts of binary XML encodings in order to clarify the ongoing and often contentious debate over their merits, particularly in the domain of high performance XML stream processing.

1. INTRODUCTION

While XML has arguably overcome questions whether it will succeed as a lingua-franca of data interchange, debate continues as to whether XML has a role to play in performance-critical applications such as database systems and high-performance network-available services. Concerns about performance of XML-based applications have steadily increased with XML's ubiquity. While these concerns are being addressed in various ways, one recurring proposal is to exploit binary serialization formats of the XML document model that are more efficient than the standard textual representation. The topic was in fact the subject of a recent W3C workshop [1] which itself contained half a dozen such proposals.

The goals of these proposals typically fall along three dimensions: Some of them attempt to reduce the size overhead of XML data processing by applying XML-specific compression techniques [25, 15, 17] . Others aim to (also) improve parsing performance, or more simply reduce the complexity of parser implementation [2]. In response to such proposals, some (e.g. [22]) have argued that the rush to discard the benefits of the firmly standardized and convenient textual XML representation should not be made without hard, compelling evidence as to necessity of such optimizations. Most of the previously cited proposals provide only limited performance studies that fail to precisely quantify the performance gains one might expect in a variety of applications.

In this paper, we aim to provide an objective evaluation of the effect of "binarization" of XML data with respect to high performance XML stream processing. Rather than propose one single format for evaluation, we quantify the individual effects of several typical binary encoding optimizations that can be exploited during XML streaming. XML stream processors, such as SAX-based parsers [18], avoid memory management overhead of DOM-based approaches, and are thus the method of choice when striving for high performance. By processing the XML stream as events, applications can more efficiently capture only the relevant portions of any incoming XML for immediate conversion into an optimized application specific format.

This paper specifically examines the effects of optimized binary XML representations on high performance XML-processing applications, such as XML database systems [26] and web services residing on a fast network [2]. We assume the infrastructure has suitable network bandwidth and storage such that the bottleneck is the XML processing itself, not the delivery of XML over the network or disk subsystem (which have motivated proposals for XML-specific compression schemes). We thus focus on optimizations that have the potential of providing benefits in this scenario. In situations where the bottleneck may also be network overhead, XML-specific or generic data compression schemes could be applied orthogonally, provided the compression scheme is of low overhead.

The paper proceeds as follows. We first outline various stream-based binary encoding options, starting from a "minimal binary" encoding, then extending it with other optimizations such as string tokenization and embedded offsets to support rapid identification of document elements that are of interest through random access. The next section discusses a stream-based XQuery processor which we use as the basis of our evaluation. Applications that use stream-based XML processing are likely to use XPath or XQuery-like methods (if not an XPath or XQuery processor itself) for extracting the information of interest from the incoming XML. This method thus provides an evaluation that should be indicative of application performance in general. Our evaluation is a carefully crafted "apples to apples" comparison of the fastest XML parser we are aware of (expat [5]), the most popular high performance parser (Xerces in C [27]), and simple parsers for processing the various binary encodings. The evaluation uses large XML datasets that are representative of both relatively flat and nested XML. To quickly summarize the results, a minimal binary encoding supports typical performance improvements of a factor of 2 over expat. Further optimizations of the binary encoding can boost performance improvements up to a factor of 6, depending on the structure of the incoming dataset and the particular information required by the application from that document. The following section discusses the impact of these results. While the minimal XML encoding has few implications on the parsing interface or the XML generation strate-

WWW2004, May 17–22, 2004, New York, NY USA.
ACM xxx.xxx.

```
<PROPERTY>
 <URL>http://...</URL>
 <ADDRESS>
   <STREET>1157 SUMNER AV</STREET>
   <CITY>APTOS</CITY>
   <STATE>CA</STATE>
   ...
 <TYPE>Detached Single Family</TYPE>
...
</PROPERTY>
```

Figure 1: An XML Fragment from the Real Estate dataset.

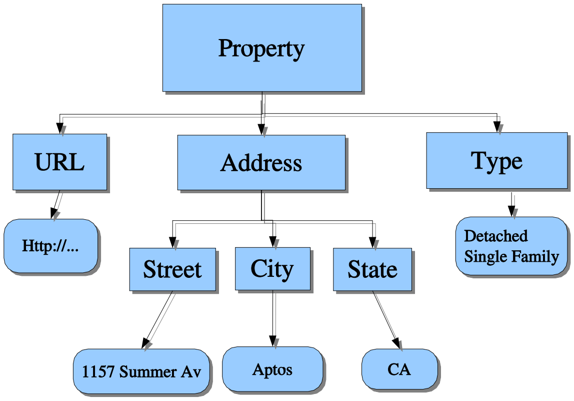

Figure 2: The XML document model depicted as a tree.

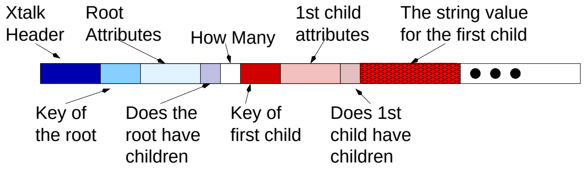
Xtalk Header
Root Attributes
How Many
1st child attributes
The string value for the first child
Key of the root
Does the root have children
Key of first child
Does 1st child have children

Figure 3: The XTalk Serialization Format

gies, the other optimizations introduce various complexities along these dimensions. We discuss these complications in order to put the performance gains into perspective. We finally conclude the paper with a discussion of where we see the binary encoding debate to be heading.

2. BINARY ENCODINGS

An XML document can be represented as a tree, with data elements appearing at the leaves and nodes corresponding to the document's tag nesting (see figure 2). We sometimes refer to this tree-structured representation of an XML document as the XML *document model*. The tree representation can be *serialized* by performing a depth-first traversal of the document tree and outputting data to a stream at each node. The standard serialization of the XML document model is the familiar textual XML document format [29].

Of the various alternate XML serialization proposals, some involve serializing the document to *multiple* streams and some a single stream. Multiple-stream serializations are typically used to improve compression by individually grouping and compressing related document components. For example, a common theme in these proposals is to serialize data and structure components to independent streams, allowing the more redundant (and hence compressible) structure components to be compressed independently. Multi-stream serializations prohibit simple pipelined streaming and processing of XML, since they typically require at least one of the streams be retrieved in its entirety before any processing takes place (stalling the pipeline for that duration). Because of such complications multiple-stream based approaches impose on pipelining, in this paper we focus on single-stream serializations. Note however that even single-stream serializations can be burdensome if the data that must be generated early on in the stream requires knowledge of data that will be serialized much later. We will be careful to point out impacts of serialization optimizations on pipelining when appropriate.

2.1 Minimal Binary Encoding

We define a *minimal binary encoding* of the XML document model as a single-stream serialization which represents only the document's structure using binary information. Thus, in a minimal binary encoding, all attribute names, values, element names, and PCDATA text appears as text in the serialization. Only delimiting of this data is achieved through binary encoding, for example through integer lengths or binary end-of-string indicators. An example of a minimal binary encoding is XTalk [2] (see figure 3), though other minimal binary encodings are possible.

Minimal binary encodings are desirable for various reasons. First, they are typically very easy to parse, and unlike more complicated binary encoding strategies, they do not adversely impact performance of generating the serialization when compared to textual XML. Though a minimal binary representation no longer satisfies the "view source" principle which has fostered growth of the world wide web, it violates this principle only slightly. Due to its simplicity, a widely adopted minimal binary encoding could quickly become viewable and editable with standard editors such as emacs via (built-in) extensions and plugins. Another positive aspect of a minimal binary encoding is that it can easily support streaming, in both inbound (parsing) and outbound (generation) directions. Streaming of both inbound and outbound XML supports *pipelined* XML processing, whereby only a small part of the stream must remain in memory to perform the desired computation. A final positive aspect of such an encoding is that no modifications of existing XML processing APIs are required to exploit them.

Though minimal binary encodings can support pipelined XML processing, not all do. For example, a problematic aspect of the XTalk serialization in this context is that the number of children of a node must be known before the node and its descendants are serialized. While the DOM API and most other memory-resident document model APIs allow applications to quickly query the number of children of any node, the stream-oriented SAX API does not. SAX supports the startElement() application callback that is invoked by the parser whenever encountering a new element in the incoming XML stream. However, this method does not provide the number of children of the element, and indeed, in many applications it would be unreasonable to assume that an XML generator would know the number of children of an element apriori. Generating XTalk output from a sequence of SAX events cannot be accomplished using bounded main memory as a result. We therefore suggest that streaming applications instead use a serialization that generates an "end of children" indicator after the node's children have been fully serialized. Our experiments have shown that the effect on parsing performance of this variation is negligible.

2.2 Tokenization

In addition to encoding delimiters, most binary encoding proposals also suggest *tokenization* of the textual strings within a document, such as those used for element and attribute names. Tokenization is useful for compression since frequently recurring strings need not be repeated within the encoding. Instead, a much smaller token identifier can be output with each occurrence. Tokenization can also improve application performance since instead of using expensive string comparisons to identify portions of the document of interest, applications can simply compare the token value – typically a single byte or integer.

The schemes for mapping token identifiers to actual string values are varied. The simplest involve producing a string table before the document itself, allowing strings in the string table to be identified by the string's offset in the table. Others involve complex code page assignments [25]. The end result however is that common strings can be more compactly and efficiently represented using much less data, often only a single byte.

Tokenization, if it is implemented through prepended token tables or through code pages, prohibits pipelined generation and consumption of the document stream. However, it is possible to instead embed token definitions into the data stream as new strings are encountered by providing a special token definition directive in the binary representation [25]. In our experiments, we evaluate the effects of an extended XTalk encoding that uses such *demand-driven* tokenization of element and attribute names. To keep the implementation simple and fast, tokens in our implementation are always 4-byte network unsigned integer values. We did not consider tokenization of element and attribute values, since they recur much less frequently.

Since a token ID can be trivially mapped to a string reference during parsing, existing APIs such as SAX are compatible with the optimization. However, to better exploit the processing advantages of tokenization, the API might be extended to allow the application to obtain the canonical string references, for example through a defineToken() callback. Such an extension would allow the application to test for element or attribute name equality via simple pointer/reference comparison instead of a full string compare. The tokenization implementation we evaluate in the experimental section exploits this strategy.

2.3 Embedded Indexes and Skip-To Pointers

Another enhancement of the binary stream representation is to embed pointers that support "random access" navigation without requiring a full depth-first traversal of the entire tree structure. Random access is valuable when only a portion of the document is needed by the application. In the context of streaming, pointers can be represented by embedding a value representing the *distance* in bytes from the value's location to a particular portion of the document that may be of interest. Such pointers allow a parser to quickly advance to relevant portions of the document, where relevance is determined by the application.

Many random-access optimizations require significant extensions to existing stream-based XML processing APIs in order to exploit them. We therefore chose to evaluate a simple but potentially powerful method of embedding pointers that can be exploited with only one small SAX API modification. This method involves embedding into the binary encoding stream the distance to the subsequent *sibling* of a given element. Conceptually, this adds navigation pointers to the XML document model as depicted in figure 4. Sibling pointers allow the application to quickly advance over the descendants of a given node when it can be determined that none of the descendants are relevant. These decisions can be made in various ways depending on the application. We will describe how such relevance decisions can be made by XPath or XQuery processors in the subsequent section.

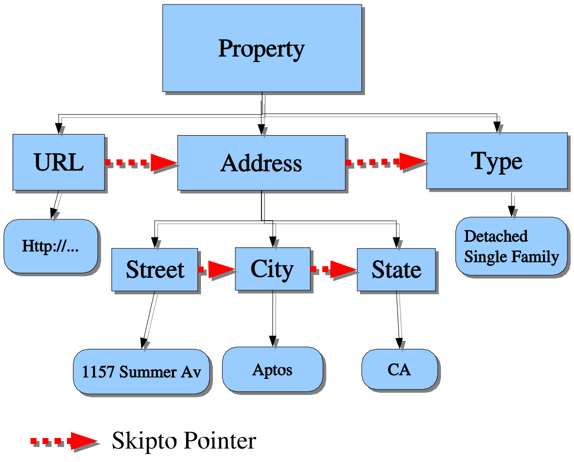

Figure 4: Sibling pointers allow subtrees to be skipped.

The impact on stream-oriented XML processing APIs required to exploit sibling pointers is small. In our implementation, only the standard startElement() callback of the SAX API is modified to return a special flag indicating the parser should advance, as quickly as possible, to the element's nearest subsequent sibling. Recall that the startElement() method is an application-provided handler invoked by the SAX parser whenever it encounters a new node in the incoming stream. The function provides the name of the element, and any attributes and attribute values embedded within that element, all of which can be used by the application in determining whether a skip to the next sibling is warranted.

While the impact of sibling pointers on stream consumption is minimal, it is not typically possible to generate such encodings using bounded memory in order to support pipelining. This is because the XML generator may not know apriori precisely what information will be serialized by an element's descendants, making it impossible to compute the sibling's distance without significant computation or delaying the stream output until the entire document has been serialized in memory. Stream-oriented XML generation is a problem in general when long-distance pointers are required. Nevertheless, we evaluate its impact since there are situations where the XML may be generated "offline", and stream-oriented processing is used only for XML consumption. For example, consider the case where information may be shipped in XML format on CD-ROM.

Random access optimizations may place restrictions on other optimizations such as on-demand tokenization. If tokens are defined as new strings are encountered in the document, then any advance to a new part of the document will likely lead to skipping over required token definitions. Formats supporting index-like optimizations such as sibling pointers should therefore provide all token definitions priori to any index structures or sibling pointers.

2.4 Schema based Optimizations

Some proposals [13, 25] suggest exploiting schema information, when available, for further optimizing the encoding. Such optimizations show promise for improving run time of applications such as XQuery processing since type information may be inferred from the schema instead of determined at runtime from the information provided by the incoming stream. A problem with relying on schema information (other than it not always being available) is that it causes applications to become tightly coupled by creating strong dependencies between parsing/generation and the schema. While these limitations may be acceptable in some settings, some would credit the success of web services and XML-based data interchange in general to its support of of loose coupling. This paper focuses on evaluating impacts of binary encoding optimizations which have only minimal impacts on loose coupling and existing programming interfaces such as SAX, since we believe such optimizations are the most pertinent to the binary encoding standard debate.

3. XQUERY OVER SAX EVENTS

To evaluate the usability of the different binary encodings in high-performance applications we used the TurboXPath [12] streaming XQuery processor as our representative application. As various systems are adding support for XML, XQuery is emerging as the main tool for optimized, high-performance querying and transforming of XML documents. XQuery, along with XSLT, is based on XPath - a path language allowing navigational access to parts of XML documents. Efficient path processing will be crucial for achieving scalability in any XQuery implementation. We describe here how TurboXPath works, and how we have modified TurboXPath to exploit the binary encoding optimizations such as tokenization and skip pointers for improved performance.

TurboXPath accepts a simple form of the XQuery **for-let-where-return** (FLWR) construct. It operates in two phases. The *retrieval* phase extracts tuples of variable bindings representing document fragments or atomic values. Each binding can represent a single item, or a sequence of items extracted from the document. During the second, *result construction* phase, the bindings are processed by a return expression, to produce the result of the query. XML constructors are common return expressions. A constructor produces a new XML data model instances based on the input expressions. For example, the query

```
for $c in
 document("c.xml")/customer[order/@date="12/12/01"]
 let $cid := $c/cid/text()
 let $name := $c/name
 for $o in $c/order
  let $a in $o/amount
    return
       <customerSummary>
         <customerID> {$cid} </customerID>
         {$name}
         {$a}
       <customerSummary>
```

extracts the names, cids and the order amounts of the customers that have placed an order on the given date. The return clause has an expression that for each triplet extracted from the document, creates a new XML element named 'customerSummary' containing the required information. The $cid and $name variables are defined using a **let** clause and can be bound to sequences (e.g. if a customer is registered using two names)

TurboXPath supports a complex set of XQuery features including **for**, **let where** and **return** with XPath paths composed of steps using **child**, **descendant**, **self**, **parent** and **ancestor** axes; any node test ('*'); functions, arithmetic and structural predicates. TurboXPath operates over any XML document, including cases when the document is recursive (e.g. nested part elements). Recursive documents impose harder processing requirements on streamed XQuery processors. In conjunction with a new XML data type, XML indexing, and a library of XQuery operators, TurboXPath can be used to extend a database engine or an application sever to full XQuery compliance.

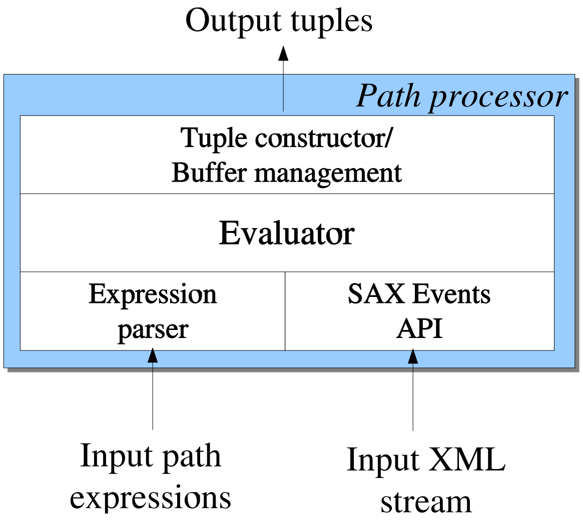

Figure 5: System architecture of the retrieval phase

Figure 5 show the internals of TurboXPath. The expression parser is responsible for parsing the input query expressions and producing a single parse tree (PT) representing all paths in the query. Nodes in the PT correspond to node tests of the query XPath expressions. Each node is annotated with name, namespace URI, node test type (element, attribute, etc.) and axis. The trees of the different XPath expressions are attached based on the variable correlation.

Figure 6 illustrates the PT generated for the example query above. Each PT has a special root node at the top, represented by 'r' in Figure 6. Nodes corresponding to the variables used in the return clause expression are called *output* nodes. In Figure 6 there are 3 output nodes: 'name', 'amount' and the text node.

A PT node may also have a set of associated predicate trees. Each predicate tree is *anchored* at a PT node, called the *context node* of that predicate. In the example, 'customer' is the context node for the predicate on the order date. Predicate tree nodes are shown in gray in the figure. Predicate trees are composed of leaves that are either constants or pointers to nodes in the PT subtree rooted at the context node. Internal predicate nodes are operators as defined in the XQuery/XPath standard specifications.

The evaluator uses the parse tree to process the stream of SAX events generated from the input document to identify the fragments that are to be extracted and returned to the consumer. During document processing, a SAX parser or a binary encoding SAX-compliant wrapper generates events from the input XML document. The evaluator uses these events to perform the state transitions and populate the buffers. While the query parse tree is *static* and does not change during processing, a stack-based data structures is used to keep track of the mapping state. Document fragments needed for predicate evaluation or matching an output node are stored as intermediate results in buffers queues associated with the output and predicate nodes. After bindings for all the output variables are generated the tuple construction module constructs tuples from buffers in the queues. The return clause expression is evaluated over the tuples bindings to produce the query result.

For this paper we have adapted TurboXPath to exploit tokenization and skip pointer optimizations when available. The most com-

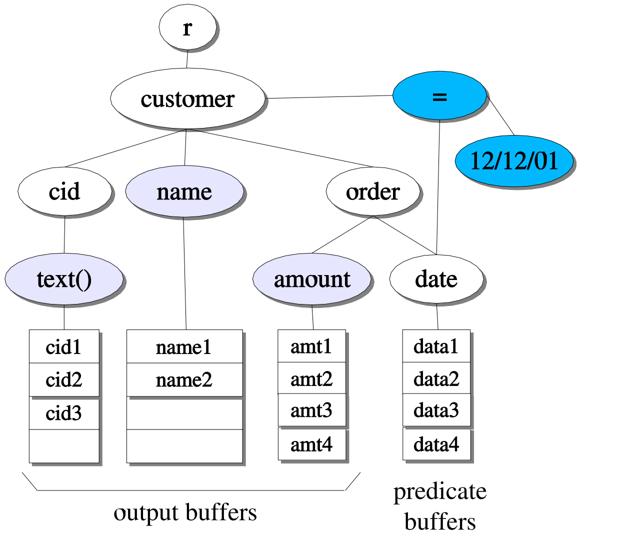

Figure 6: Parse tree example

Label	Description
Xalan-J	Xalan-J V2.5.2 with Xerces-J V2.4.0 parser
Xalan-C	Xalan-C V1.6 with Xerces-C V2.3.0 parser
Xerces	TurboXPath with Xerces V2.1.0 parser
Expat	TurboXPath with Expat V1.95.7 parser
XTalk	TurboXPath with XTalk V1.0 parser
Skip	TurboXPath with XTalk + Skip option
Token	TurboXPath with XTalk + Token option
Token+Skip	TurboXPath with XTalk + Token and Skip

Table 1: Methods compared and their labels.

mon test performed by a TurboXPath predicate node is to match an element or attribute name against one that appears in the query. When the encoding supports tokenization, our implementation uses appropriate pointer equality instead of string equality to test if the element or attribute names match. Our implementation also augments the query evaluator to return the "skip" flag from startElement() whenever it reaches a state in the predicate tree where the processor cannot generate any results until the given element is terminated. For example, if the customer elements of the document in the example above contained multiple 'payment' children, then these could be skipped since there are no conditions or extractions that depend on them.

4. EXPERIMENTAL SETUP

We conducted a series of experiments to measure and compare the performance of the parser and query methods discussed earlier. The methods were labeled as shown in Table 1. In an experiment, a chosen method is invoked to answer a query on a particular dataset. The datasets used and queries answered are described below. Each query was presented to TurboXPath-based methods using the XQuery syntax and to Xalan-based methods using an equivalent XSL syntax. Each experiment was executed three times and the lowest observed run time was used in the comparison.

All parsers were run in non-validating mode and all query methods based on TurboXPath used the SAX callback API. Datasets were normalized to use UTF-8 encoding, and entity references in the DBLP data were pre-expanded to support non-validating parsing. Our decision to focus on non-validating parsing was based on the fact that validation adds another layer of overhead, and is likely to be disabled in high performance applications. Since expat does not provide a SAX callback API, we implemented a SAX callback layer on top of expat's lower-level callback interface. The implementation closely mimicked the implementation of our binary parser implementations with respect to buffer and memory management to ensure a fair comparison.

The experiments were performed on an IBM Thinkpad T23 with a 1.133 GHz Pentium III CPU and 512 MB of memory running Windows 2000. Each program was compiled with the Miscrosoft Visual C++ 6.0 compiler using identical compiler settings. Xalan-J was executed in the Sun Java Runtime Environment V1.41_04. We also ran the same experiments on a Red Hat Linux 8.0 and gcc-3.2 based machine. Since results were qualitatively similar, we present only the Windows run times here.

All datasets used in the experiments were buffered in main memory during runs. This eliminated the effect of disk I/O and focused the experiments on pure CPU cost.

4.1 Datasets

Two datasets were used in the experiments, labeled DBLP and RE. DBLP is an XML version of the DBLP Computer Science Bibliography database available at http://dblp.uni-trier.de/xml/, and is roughly 180 MB in size. The structure of DBLP is fairly flat, consisting of a listing of scientific publications along with bibliographic information such as title, name of authors, and name of conference or journal.

The RE dataset is a listing of 60,000 real estate properties (homes) that were offered for sale in the San Francisco Bay Area in 2002-2003. The dataset is roughly 50 MB in size and its structure is 6 levels deep (see DTD in Appendix B). In addition to basic descriptive information of each property such as address, age, and size, the listing includes date and price information. The dataset was crawled and extracted from mlslistings.com, a public web site providing real estate data. Data extraction was performed using XSL and the ANDES Web Data Extraction system [19].

In order to study the scalability of the methods, we created additional RE datasets containing a different number of properties. Datasets were labeled RE-n where n was set to 2.5, 5, 10, 20, 40, and 60 and indicates the number of properties (in thousands) contained in the dataset. For example, RE-2.5 contains the most recent (newest) 2,500 properties. It is assumed that data distribution is uniform along the time dimension, meaning, for example, that RE-10 contains twice as many properties located in a given city or having a certain size as RE-5. In other words, the selectivity of the queries used in the experiments was independent of the dataset size.

4.2 Queries

For the DBLP dataset, we used the following query (DBLP-QUERY): *Return the year and title of all conference publications authored by "Peter P. Chen"*. The XQuery syntax is shown below and the corresponding XSL syntax is shown in Appendix B. We also ran a test on the DBLP dataset to measure to cost of parsing alone.

```
for $I in /dblp/inproceedings
            [author[text()="Peter P. Chen"]];
  let $Y := $I/year;
  let $T := $I/title;
  return $Y, $T
```

Three queries were run on the RE dataset. RE-QUERY1 is *"Return all properties listed in the city of Aptos"*. It returns one particular section of the dataset and does not need to search deep into the tree. RE-QUERY2 is *"Return the age and price of all properties in ZIP code 95032 with size between 1875 and 2000 square feet"*. It is highly selective but requires that each `PROPERTY` element be visited and evaluated. RE-QUERY3 is *"Return all properties in ZIP code 95037"*. It performs a recursive search on all `ZIPCODE` elements in the dataset and returns the few that match. The XQuery syntax for the three queries is shown below and the corresponding XSL syntax is shown in Appendix B.

```
RE-QUERY1:
  for $I in /REALESTATE/CITY[@NAME="APTOS"]/PROPERTY;
    return $I

RE-QUERY2:
  for $l in /REALESTATE/CITY/PROPERTY
              [SQFT > 1875 and SQFT < 2000
               and ADDRESS/ZIPCODE = 95032];
    let $a := $l/AGE;
    let $p := $l/LISTINGS/LISTING/PRICE;
    return $a, $p;

RE-QUERY3:
  for $I in //ZIPCODE[text() = 95037];
    return $I
```

5. RESULTS

Charts depicting the experimental results are shown in Figures 7 through 10. Because some of the figures are small and may be difficult to interpret, we provide raw numerical values in Appendix B. In Figure 7, we show the run time of the different query methods for DBLP-QUERY as well as a "parse-only" run time. Results for Xalan-J and Xalan-C are not shown in the graph because Xalan-J was unable to execute the query due to insufficient memory, and the run time of Xalan-C was off the chart (1,800 seconds) due to excessive memory consumption and paging. Run time for TurboXPath with the Xerces parser is labeled XMLNav.

Parse-only run times were calculated by running each parser with a "no-op" application. That is, the SAX callback functions were invoked by the parsers, but the bodies of these functions were empty. Looking at these run times, we observe that the expat parser performs roughly 3 times faster than Xerces. XTalk and its different variations ran a bit more than twice as fast as expat. A 5% improvement in run time is obtained by tokenization, while the skip option slowed parsing times by about 10%. This slowdown results from the combination of extracting the skip pointers, and an increase in dataset size of 10%.

When the query evaluation cost is included in the run times, we notice that the performance ratio between expat and Xerces remains constant at 3, but the performance advantage of XTalk relative to expat is reduced somewhat. Switching from expat to XTalk improved the run time by 35%, and a further improvement of about 10% resulted from exploiting the skip option. Tokenization helped yet another 5%.

We note that the benefit of skipping in DBLP-QUERY was limited because the skip distance was rather small; child elements of `/dblp` which did not match `inproceedings` were fairly small, about 200 bytes each. Small skips such as these were performed by using `fread()` instead of `fseek()` because the latter causes I/O buffers to be flushed.

A larger benefit of skipping is seen in our experiment with RE-QUERY1 (Figure 8). The query searches for properties in the city of Aptos, which means that the entire XML subtree of all cities whose name was not Aptos could be skipped. The RE dataset contains 102 cities with an average XML content size of about 0.5 MB per city, so the skip distance was significant. TurboXPath using the Xerces parser executed the query about 5 times faster than Xalan-J and twice as fast as Xalan-C. Switching from Xerces to the expat parser reduced the run time by another factor of 2. The run time of XTalk was about half of that of expat, and the skip option reduced the run time by another factor of 3. Tokenization seemed to have very little effect in this query.

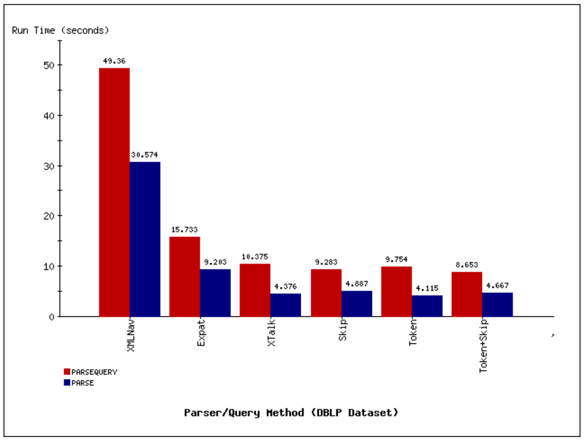

Figure 7: Results of DBLP-QUERY experiment.

The results of RE-QUERY2 confirmed the performance advantage provided by XTalk over Xerces and expat, but also highlighted the significance of complex query evaluation with respect to parsing time. The query reaches deeper into the RE data structure, requiring more XPath navigation than RE-QUERY1. This reduces the performance difference between the different parser options: TurboXPath with Xerces is only 40% faster than Xalan-C, and TurboXPath with XTalk is only 35% faster than TurboXPath with expat. The relative difference between Xerces and expat remained at about a factor of 3.

RE-QUERY2 does not allow for effective skipping because every `PROPERTY` element must be inspected for a potential match. The run time of Xalan-J and Xalan-C is virtually unchanged from RE-QUERY1. Tokenization does seem to help more than it did in RE-QUERY1, this time providing a 5% run time improvement. This is because the query reaches deeper into the document tree, requiring more element and attribute-name equality comparisons which can be performed by simple pointer comparisons when tokenization is used.

All methods except Xalan-J executed RE-QUERY3 faster than RE-QUERY2. Query evaluation was significantly simpler in the former, resulting in a vast reduction in overall run time. The relative performance of the methods was similar to RE-QUERY1, except for methods using the skip option. Those methods were not able to exploit skipping (as every `ZIPCODE` had to be examined by the query), and in fact suffered from the increased overhead required to extract and advance over the skip pointers.

The final experiment measured the scalability of the different methods as a function of dataset size (Figures 9 and 10). RE-QUERY1 was executed against different RE dataset sizes, ranging from 2,500 property listings (3 MB) to 60,000 (54 MB). We observe that all methods except Xalan-J scale up linearly with the size of the dataset, which is expected since stream-oriented XML (query) processing is a linear time procedure. Though we did not confirm it, we suspect the non-linear scale-up of Xalan-J is likely due to a non-linear increase in garbage collection overhead.

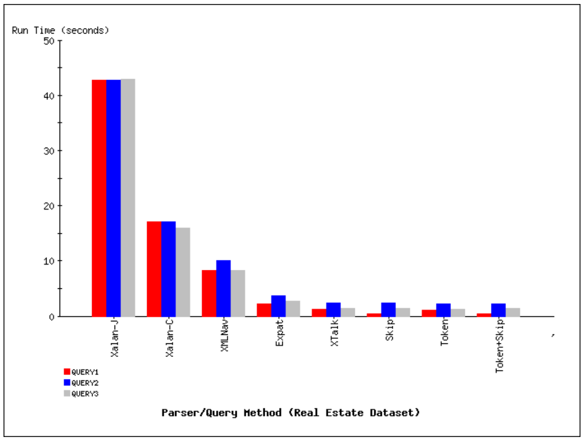

Figure 8: Results of RE-QUERY experiments.

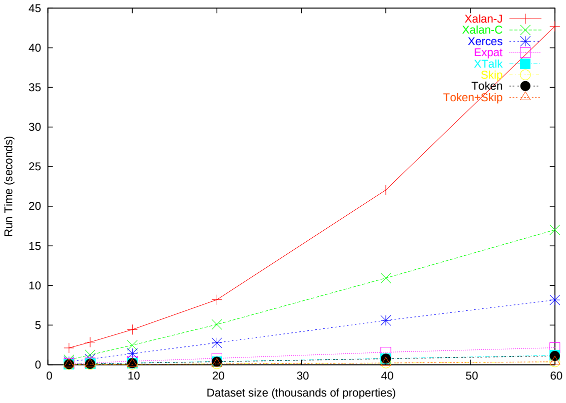

Figure 9: Scalability of methods (RE-QUERY1).

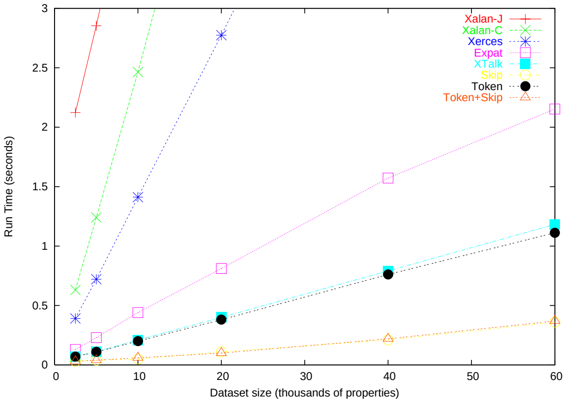

Figure 10: Expanded range of scalability experiment.

6. DISCUSSION

The parse-time only experiments clearly show that the binary XML encodings can be parsed over 2 times faster than XML. It's possible that further optimization of the binary parsers can increase the performance margin, but we feel it would be unlikely for a binary parser to outperform an optimized XML parser by much more than a factor of three in general without exploiting random access or schema-specific optimizations.

While the performance gains of binary encodings over expat are significant, the performance gains of expat over Xerces are even more so. This suggests that those who encounter performance problems in their XML applications should first consider converting DOM-based implementations to stream-oriented ones that exploit an optimized parser such as expat.

The subsequent experiments reveal that the application overhead of extracting relevant data from the XML document can be kept rather small. For the simplest queries, the XQuery overhead was less than the parser overhead. For more complex queries, the experiments revealed a clear shift in overhead from parsing to the XQuery application, as indicated by the decrease in speedups between expat and the binary encoding parsers. One anomoly we identified with respect to application overhead was that TurboXPath with Xerces spent considerably more time in the application than did TurboXPath with the lighter-weight parsers, even though the application code performed the same operations. For example, figure 7 shows that Xerces-based processing spent almost 15 seconds in application code compared to 6.5 seconds for expat and XTalk. (Tokenization reduced application overhead slightly to 5.6 seconds due to its enabling of pointer-based equality tests instead of full string comparisons.) Profiling did not reveal any clear cause of this anomoly – TurboXPath with Xerces simply spends more time in the same looping structures compared to when using the other parsers. We conjecture that the increased memory footprint of the Xerces parser could be resulting in poor CPU cache utilization.

A surprising finding is that tokenization only has a small effect on overall parsing as well as application performance. In situations where network or disk throughput is the bottleneck, tokenization could prove much more beneficial. However in situations where CPU is the bottleneck, the performance gains are typically under 10%. While tokenization might be an effective means at document compression, it does not necessarily significantly reduce compute overhead.

The results for the skip-pointer enhanced encoding show that random access can be very beneficial for the appropriate queries on deeply nested data. Unfortunately this comes at the expense of pipelining since skip-pointers cannot be produced while generating XML. It also slows the performance of queries that cannot exploit the skip-pointers, though only by a small (10%) amount.

In summary, from the perspective of CPU impacts on stream-based XML processing, an optimized XML parser implementation such as expat yields significant and consistent performance improvements over Xerces, and minimal binary encodings provide significant and consistent reductions in parsing overhead over XML. Tokenization helps improve performance only slightly even though it has the potential to reduce cost of both parsing and application-layer operations. Skip-pointers provide the most substantial performance improvements, though only in limited circumstances, and at the expense of pipelining.

7. RELATED WORK

The Apache Xerces project [27] has set the standard in XML parser implementation, providing both Java and C++ versions that implement both the DOM [28] and SAX [18] standard parsing APIs. There are various simplified parser implementations that trade generality for improved performance, including Sparta [10] which claims a factor of 5 improvement over Xerces Java. On the C side, the expat [5] parser is the fastest publicly available XML parser of which we are aware. We found that in general the performance of the C and C++ parsers significantly outperform their Java counterparts.

There are several proposals for binary XML serialization and compression formats, including Millau [25], XCQ [13], XTalk [2], XMill [15], WBXML [17], and half a dozen proposals from the Workshop on Binary Interchange of XML Information Item Sets [1]. These proposals all perform only limited comparisons of their encodings to standard XML. This paper instead individually evaluates optimizations that are common across many of these proposals in order to understand their effects in the context of high performance XML stream-based applications.

Several projects including Lore [8] and Apache Xindice [26] provide XML data-storage and data-management tools. Commercially there are several XML data management packages available including the DB2 XML Extender [11] and Oracle XML DB [21], both of which solve the more general XML selection problem (as well as providing the expected ACID guarantees) and thus have performance challenges compared to a simple approach. Product Attribute Table approaches, which are suitable for representing market-basket like data, are covered in [3].

There are also several schemes in the literature for indexing XML and semi-structured data for both simple [4, 14] and branching [24, 6] path expressions. Unfortunately indices for XML data that are general enough to be used in answering a wide range of queries tend to be so large in practice that they offer little improvement over evaluating queries directly on the data [24]. They are also not compatible with stream-based processing – the focus of this paper.

Recently an XQuery implementation [7] and some XPath implementations [20, 9, 16, 23] that operate over XML streams have been reported. In [7], Florescu et. al. describe a XQuery engine designed for data transformations on XML streams based on the iterator model. While the focus of this engine is on completenes, TurboXPath, on the other hand, has been designed as high-performance iterator that can be used by an iterator-based query processor. The XPath processors proposed in [16, 20, 23] are based on connecting a set of FA in a network that represents the query. These approaches support a much smaller set of features compared to TurboXPath and may require memory that grows exponentially with the size of the query.

8. CONCLUSIONS

We have evaluated the impact of several common binary XML encoding optimizations on XML parsing and application performance. Our representative application was a stream-based XQuery processor enhanced to exploit encoding-specific optimizations such as tokenization or skip-to pointers, when available. We chose these optimizations for our evaluation because they can be exploited by applications through only minimal modification of stream-based APIs such as SAX.

Our experiments have shown that there is some merit to suggestions that XML parsing and application performance can be improved upon through binary encodings. However, more significant improvements can be obtained by adopting optimized stream-based XML parsers if they are not already being used. Also, because C and C++ parsers outperform their Java counterparts, Java applications could make substantial performance gains through native libraries for handling the XML streams.

We have also found that some optimizations such as tokenization do not lead to substantial performance improvements, even if the application can exploit tokens for reduced string comparison cost. Tokenization may be more useful in applications requiring XML compression, though further experiments are required to justify it even in this context since generic compression schemes may be just as effective.

We have attempted to remain impartial regarding the debate over the need for a standard binary encoding format for XML. Our results have shown that with the exception of minimal binary encoding strategies, most binary encoding optimizations yield performance improvements in only limited applications or situations, and/or restrict the ability for pipelined XML processing. This supports the contention in [22] that there is not one binary encoding standard that could satisfy the needs of all applications. On the contrary, however, a minimal binary encoding standard would appear to at least provide performance benefits to most applications, without any significant drawbacks other than compromising the view-source principle. We therefore suggest any standards work in the binary encoding area consider the possibility of a minimal binary encoding standard as opposed to attempting to maximize performance gains at the expense of encoding generality. Optimizations that can offer substantial performance improvements in more limited circumstances, such as random-access or schema-specific optimizations could still be exploited by internal representations when appropriate, but may not be well-suited for standardization efforts.

9. REFERENCES

[1] Report from the w3c workshop on binary interchange of xml information item sets. `http://fr.w3.org/2003/08/binary-interchange-workshop/Report.html`, 2003.

[2] R. Agrawal, R. Bayardo, D. Gruhl, and S. Papdimitriou. Vinci: A service-oriented architecture for rapid development of web applications. In *WWW10*, Hongkong, May 2001.

[3] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB 2001)*, Roma, Italy, September 2001.

[4] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *The VLDB Conference*, pages 341–350, 2001.

[5] C. Cooper. Using expat. In *xml.com (http://www.xml.com/pub/a/1999/09/expat/index.html)*, 1999.

[6] A. M. Flavio Rizzolo. Indexing xml data with toxin. In *WebDB-2001*, 2001.

[7] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, A. Sundararajan, and G. Agrawal. The bea/xqrl streaming xquery processor. In *Proc. of the 29th VLDB Conference*, 2003.

[8] R. Goldman, J. McHugh, and J. Widom. From semistructured data to xml: Migrating the lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, Philadelphia, Pennsylvania, June 1999.

[9] T. Green, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. Technical report, University of Washington, 2001.

[10] HP Labs. The sparta project. `http://sparta-xml.sourceforge.net/`.

[11] IBM. Db2 xml extender. `http://www-3.ibm.com/software/data/db2/extenders/xmlext/`.

[12] V. Josifovski, F. Fontoura, and A. Barta. Querying xml streams.

[13] W. Y. Lam, W. Ng, P. Wood, and M. Levene. Xcq: Xml compression and querying system. In *Proc. of the Twelfth Int'l World Wide Web Conf.*, 2003.

[14] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB 2001*, 2001.

[15] H. Liefke and D. Suciu. Xmill: an efficient compressor for xml data. In *Proc. of the ACM SIGMOD conf. on Management of Data*, 2000.

[16] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based xml query processor. In *Proc. of the 28th VLDB Conference*, 2002.

[17] B. Martin and B. Jano. Wap binary xml content format, w3c recommendation. `http://www.w3.org/TR/wbxml/`, 24 June 1999.

[18] D. Megginson and D. Brownell. Sax. `http://www.saxproject.org/`.

[19] J. Myllymaki. Effective Web data extraction with standard XML technologies. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001.

[20] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[21] Oracle. Oracle xml db. `http://www.oracle.com/xml`.

[22] S. Pal, J. Marsh, and A. Layman. A case against standardizing binary representation of xml. In *Workshop on Binary Interchange of XML Information Item Sets*, 2003.

[23] F. Peng and S. S. Chawathe. XSQ: Streaming XPath Queries.

[24] J. F. N. Raghav Kaushik, Philip Bohannon and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD 2002*, 2002.

[25] N. Sundaresan and R. Moussa. Algorithms and programming models for efficient representations of xml for internet applications. In *Proc. of the Tenth Int'l World Wide Web Conf.*, 2001.

[26] The Apache Project. Apache xindice. `http://xml.apache.org/xindice/`.

[27] The Apache Project. Xerces-c++ version 2.3.0. `http://xml.apache.org/xerces-c/`.

[28] The World Wide Web Consortium. *Document Object Model (DOM)*. http://www.w3.org/DOM.

[29] The World Wide Web Consortium. *Extensible Markup Language (XML)*. http://www.w3.org/XML.

APPENDIX

A. XTALK SPECIFICATION

```
doc ::= 'X' versionid int ('p' pi)*
          'E' element ('p' pi)*

versionid ::= byte

element ::= string int attr* int child*

child ::= ('s' string)
        | ('E' element)
        | ('p' pi)

attr ::= string string

pi ::= string string

string ::= int utf8

utf8 ::= (byte array of valid utf8 character data)
int ::= (4 byte big-endian unsigned integer)
```

Constraints:
doc:
Total number of pi occurrences must equal the value of the int + 1.
element:
String must abide by the XML 1.0 restrictions on tag names. Total number of attr occurrences must equal the value of the preceding int. Total number of child occurrences must equal the value of the preceding int.
attr:
First string must abide by XML1.0 restrictions on attribute names. Second string must abide by XML1.0 restrictions on normalized attri bute values.

B. EXPERIMENT DETAILS

B.1 DTD for Real Estate Dataset

```
<!ELEMENT REALESTATE (CITY+)>
<!ELEMENT CITY (#PCDATA|PROPERTY)*>
<!ATTLIST CITY NAME CDATA #IMPLIED>
<!ELEMENT PROPERTY (URL,IMAGEURL,ADDRESS,TYPE,AGE,SQFT,BEDROOMS,
                    BATHROOMS,GARAGE,LOTSIZE,DESCRIPTION,LISTINGS)>
<!ELEMENT URL (#PCDATA)>
<!ELEMENT IMAGEURL (#PCDATA)>
<!ELEMENT ADDRESS (STREET,CITY,STATE,ZIPCODE)>
<!ELEMENT STREET (#PCDATA)>
<!ELEMENT STATE (#PCDATA)>
<!ELEMENT ZIPCODE (#PCDATA)>
<!ELEMENT TYPE (#PCDATA)>
<!ELEMENT AGE (#PCDATA)>
<!ELEMENT SQFT (#PCDATA)>
<!ELEMENT BEDROOMS (#PCDATA)>
<!ELEMENT BATHROOMS (#PCDATA)>
<!ELEMENT GARAGE (#PCDATA)>
<!ELEMENT LOTSIZE (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ELEMENT LISTINGS (LISTING+)>
<!ELEMENT LISTING (PRICE)>
<!ATTLIST LISTING DATE CDATA #IMPLIED>
<!ATTLIST LISTING ID CDATA #IMPLIED>
<!ELEMENT PRICE (#PCDATA)>
<!ATTLIST PRICE CURRENCY CDATA #IMPLIED>
```

B.2 XSL Query for DBLP Dataset

```
<xsl:template match="/dblp">
  <RESULT>
    <xsl:for-each select="inproceedings[author[text() = 'Peter P. Chen']]">
      <YEAR><xsl:value-of select="year"/></YEAR>
      <TITLE><xsl:value-of select="title"/></TITLE>
    </xsl:for-each>
  </RESULT>
</xsl:template>
```

B.3 XSL Query for Real Estate Dataset

```
<xsl:template match="*|@*">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates select="*"/>
    <xsl:apply-templates select="text()"/>
  </xsl:copy>
</xsl:template>

<!-- RE-QUERY1 -->
<xsl:template match="/REALESTATE">
  <RESULT>
    <xsl:apply-templates select="CITY[@NAME = 'APTOS']/PROPERTY"/>
  </RESULT>
</xsl:template>

<!-- RE-QUERY2 -->
<xsl:template match="/REALESTATE">
  <RESULT>
    <xsl:for-each select="CITY/PROPERTY[SQFT > 1875 and SQFT &lt; 2000
                                        and ADDRESS/ZIPCODE = 95032]">
      <AGE><xsl:value-of select="AGE"/></AGE>
      <PRICE><xsl:value-of select="LISTINGS/LISTING/PRICE"/></PRICE>
    </xsl:for-each>
  </RESULT>
</xsl:template>

<!-- RE-QUERY3 -->
<xsl:template match="/">
  <RESULT>
    <xsl:apply-templates select="//ZIPCODE[text() = 95037]"/>
  </RESULT>
</xsl:template>
```

B.4 Numerical Results

	DBLP	DBLP	RE	RE	RE	RE	RE	RE	RE	RE	RE
Method	PARSE	QUERY	QUERY1	QUERY2	QUERY3	2.5	5	10	20	40	60
Xalan-J	N/A	N/A	42.701	42.731	42.972	2.123	2.854	4.437	8.212	22.052	42.701
Xalan-C	N/A	1804.765	17.024	16.994	15.993	0.631	1.241	2.464	5.088	10.935	17.024
Xerces-C	30.574	49.360	8.182	10.114	8.272	0.391	0.721	1.412	2.774	5.588	8.182
Expat	9.203	15.733	2.153	3.695	2.714	0.130	0.230	0.441	0.811	1.572	2.153
XTalk	4.376	10.375	1.182	2.353	1.372	0.070	0.111	0.210	0.400	0.791	1.182
Skip	4.887	9.283	0.360	2.363	1.442	0.030	0.040	0.050	0.110	0.210	0.360
Token	4.115	9.754	1.112	2.263	1.302	0.070	0.110	0.200	0.381	0.761	1.112
Token+Skip	4.667	8.653	0.370	2.263	1.402	0.030	0.040	0.060	0.100	0.220	0.370

Table 2: Numerical results of experiments.