Lazy XML Processing

Markus L. Noga
Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany
markus@noga.de

Steffen Schott
Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany
info@steffen-schott.de

Welf Löwe
Växjö universitet
MSI
Software Technology Group
351-95 Växjö, Sweden
welf.lowe@msi.vxu.se

ABSTRACT

This paper formalizes the domain of tree-based XML processing and classifies several implementation approaches. The lazy approach, an original contribution, is presented in depth. Proceeding from experimental measurements, we derive a selection strategy for implementation approaches to maximize performance.

Categories and Subject Descriptors

E.2 [**Data**]: Data Storage Representations; I.7.2 [**Document and Text Processing**]: Document Preparation; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Measurement, Performance, Design

Keywords

Parsing, Lazy evaluation, XML, Document object model

1. INTRODUCTION

XML data processing is a cornerstone of many contemporary applications. Examples of this include content management systems, office packages, web development and electronic business suites, all of which ultimately manipulate XML documents.

Certain simple operations on XML documents can be defined on a textual document representation [24]. Text substitution is such a case. However, most operations must be aware of nested structures, e.g., visualizing route planning results as vector graphics. Formulating these operations on a textual representation is hard. Trees [21] are a more convenient representation for these tasks.

Computations on XML trees are performed by *XML processors*. In general, a processor has m input and n output ports, which may be typed with DTDs or XML Schemas [22, 23]. We call $0:1$ processors *sources*, $1:0$ processors *sinks* and $1:1$ processors *transformations*. One example of a source is an XML parser which *deserializes* textual XML representations to trees. Correspondingly, *serialization* as performed by XML writers is a simple sink. When restricted to a single input, XSLT scripts [25] are transformations in our sense.

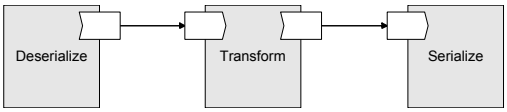

Figure 1: A simple XML processing network.

Many processors do not require full access to their inputs. E.g., when generating outlines, only tree nodes corresponding to chapters and sections are visited, but the bulk of paragraph nodes are not. Given a specific input tree, we define the *coverage* of a processor's input port as the fraction of nodes visited. For $1:n$ processors, we use the term processor coverage.

Individual processors can be connected to form an *XML processing network*. As in architectural systems [19, 4], every input port must be connected to a single output port. Data flows from output to input. Fig. 1 shows a pipeline, a simple case of processing network.

There are several approaches to implement XML processors within a network. In this paper, we ignore the issue of distribution and classify approaches according to control flow and type information (Fig. 2).

Control flow can be either push- or pull-based. In the *push* model, activity resides with the sources. They push data along the edges of the network irrespective of the attached processors' real needs. In the *pull* model, the sinks are active. They pull exactly the required partial documents from the processors attached to their inputs.

Type information can be interpreted or compiled. In *interpreted* typing, types are available at run time only. E.g., deserializers interpret and pass document types, and transformations interpret XSL transformations and pass output types. In *compiled* typing, type information is available during network composition. This admits static preprocessing.

The two dimensions of control flow and type information yield four different approaches. Traditional *eager* processing is the simplest of them, as it employs neither partial processing nor preprocessing. *Lazy* processing corresponds to lazy evaluation as in functional programming languages, which we address in section 2.1. *Data-driven* processing only implements preprocessing. Finally, the *demand-driven* ap-

DocEng'02, November 8–9, 2002, McLean, Virginia, USA.

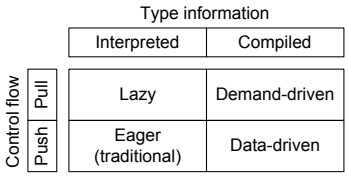

Figure 2: Approaches to XML processing

proach combines both lazy processing and preprocessing.

Given this classification of implementation approaches, we can pose the central question of this paper: "What is a selection strategy to maximize performance?"

When the consumer actually uses the entire document, pull-based processing should carry a performance penalty over push approaches due to multiple invocations over the system boundary. As coverage decreases, pull-based performance should improve dramatically. Also, we would expect compiled approaches to offer superior performance compared to interpreted ones.

This paper focuses on the first claim and analyzes it quantitatively. We choose the well-known domain of deserializers to measure the tradeoffs between push- and pull-based processing. As our prior work validates the second claim [13], we restrict ourselves to the simpler dynamically typed case.

As a welcome side effect, lazy deserialization produces information about the access profile of the consumer on the fly. By evaluating the progress of parsing and the fractions of the tree already created, an accurate picture can be obtained. This access profile may prove useful in tailoring future static optimizations. Thus, lazy deserialization is an interesting platform for future work.

The next subsection, 2 examines the state of the art. We describe our lazy deserialization architecture in section 3. Comparative measurements are taken in section 5. Section 6 summarizes the results and outlines future work.

2. STATE OF THE ART

We discuss lazy evaluation in the context of functional programming languages. Then, we focus on XML and examine its textual representation, the standard DOM interface to XML trees and the XSL transformation language from the viewpoint of pull-based processing. We examine available available parsers and transformers and classify them according to fig. 2. We check if available XSL test suites can be used to measure the tradeoff in question. Finally, we summarize our findings.

2.1 Lazy Evaluation of Functional Languages

Functional programming originated with Kleene's model of algorithms as μ-recursive functions [11]. In that model, functions are taken from a predefined set or composed with substitution, primitive recursion or μ-recursion.

The lambda-calculus [5] is a still simpler model. It consists only of function definitions and function applications. Most functional languages can be expressed completely in terms of the lambda calculus [15]. Lambda is also a convenient model for large subsets of imperative languages [12].

We illustrate the lambda calculus with a short example. `first:=`λ`x.`λ`y.x` simply returns its first argument. Let `succ` be the successor function on a natural number representation, then `firstsucc:=`λ`x.`λ`y.first(succ(x),succ(y))` will return the successor of its first argument.

Terms like `firstsucc 0 1` can be evaluated using different strategies. While all strategies evaluate that term to `1`, they may differ in performance. The direction of evaluation within a given function (left-to-right or right-to-left) usually has little impact. However, the treatment of nested function applications is the most important performance characteristic of an evaluation strategy.

Consider strategies that compute nested function applications in a bottom-up manner. They first evaluate all inner subterms, then they combine the results using the outer function. In our example, they compute the successors of both 0 and 1, then discard the second result. This approach is called *eager* evaluation.

The inverse approach first examines which results are required by a given outer term, then computes and combines them. In our example, such strategies determine that `first` only uses its first argument, then compute only the needed successor of 0. This approach is called *lazy* evaluation.

Lazy evaluation is generally faster than eager evaluation because it avoids unnecessary function applications. In our example, lazy evaluation requires one successor evaluation, whereas eager evaluation requires two. While successors are computationally cheap, this difference becomes even more pronounced for expensive functions like factorials.

Lazy evaluation has another important advantage. As only necessary subterms are ever evaluated, the total number of subterms may well be infinite. Many tough problems are easily expressed using such infinite intermediate representations [9]. Thus, most functional languages use lazy evaluation. Our lazy approach is an instance of this approach as well. Unfortunately, due to their preprocessing phase, lazy parsers cannot operate on infinite intermediate structures. However, these structures may be an important concept for future lazy XSL transformation.

2.2 XML Standards

The basic properties of XML [24] are well known. XML files store pre-order depth-first traversals of XML trees [21]. For pull-based processing, we are interested in parsing XML files partially. Unfortunately, their nesting structure encloses element content with pairs of corresponding opening and closing tags. The linkage between adjacent nodes is not stored explicitly, so subtrees cannot be skipped without being scanned. In other words, simple navigation operations on the tree translate to expensive operations on the serialization format. Furthermore, the entire document is contained within a single top-level element. Thus, a document has to be scanned in its entirety before any statement about its top-level structure can be made. As a result, partial parsing generally requires some form of preprocessing.

DOM defines an interface to tree representations of XML data. It is also a well-established standard [20]. For efficient pull-based processing, a tree interface must offer highly selective access operations. Informally, *selectivity* is the fraction of nodes returned by an operation that we are actually interested in. For low selectivity, the burden of eliminating undesired results resides with the caller. The more selective an interface is, the tighter the limits to coverage that can be preserved throughout the processing chain. DOM offers access operations based on position and name. Whether this is sufficient cannot be determined *a priori*.

XSLT scripts specify transformations over the set of XML documents [25]. Inspired by functional programming, XSLT operations are free of side effects other than tree output. Thus, bound functionals may be retained and evaluated to tree fragments at a later time [14]. XSLT is therefore well-suited to partial transformations of an input document and fits well into the concept of pull-based processing.

2.3 XML Parsers

Most widely available XML parsers follow the traditional approach of interpreting DTDs or XML Schemas. Examples of this genre are Apache Xerces [1], James Clark's xp [6] and the Sun parsers [27]. Xerces can produce DOM output, the others implement SAX-based messaging [17].

Our prior work includes compiler approaches to XML parsing, most notably xsdc [16] and aXMLerate [3]. Both provide proprietary messaging and DOM interfaces.

The Xerces parser also supports lazy processing via the `defer-node-expansion` feature. The documentation states: *If this feature is set to true, the DOM nodes in the returned document are expanded as the tree is traversed.* The FAQ document states more precisely that *All of the immediate children of a Node are created when any of that Node's children are accessed.*

Unfortunately, there is a dearth of specifics. An inspection of the source reveals that deferred node expansion does exactly what it says — it defers the object encapsulation of node data stored in tables. As these tables already contain fully parsed document tree data, performance gains are bound to be small.

There are no demand-driven parsers yet.

2.4 XML Transformers

Again, most widely available transformers follow the traditional approach of interpreting an XSLT script for immediate processing. E.g., Saxon [10], xt [7], and Xalan [2] fall into this category. Only Xalan can operate on DOM input, the other contenders accept SAX messages only.

Compiler approaches include [8] and [18]. The former uses proprietary specifications, whereas the latter accepts standard XSLT scripts. There are no lazy or demand-driven approaches yet.

2.5 XML Test Suites

Two test suites for XSLT processors are publicly available. The older XSLBench suite [28] is of historical interest only, as the newer XSLTMark [29] has incorporated XSLBench almost completely.

XSLTMark contains a variety of test cases, which simultaneously serve as regression and performance tests. Unfortunately, the regression aspect is highly dominant. All but one test case actually traverse the entire document. Although they form valuable execution frameworks, publicly available test suites are thus ill-suited to evaluate partial processing.

2.6 Summary

Lazy evaluation of a functional program is an important performance optimization that additionally allows programmers to use infinite intermediate structures. This makes lazy evaluation interesting for XML processing.

A lazier approach than Xerces' is called for. All such approaches require preprocessing. As DOM is the canonical interface to XML trees, it should be implemented by the lazy parser to ensure comparability. Experiments must tell if DOM is selective enough to retain coverage bounds. E.g., transformation processors map XPath selection operations on DOM in an implementation-dependent way, thus sparse XPath selection need not result in sparse DOM queries.

Comparing lazy and traditional parsing performance requires a transformation context. Apache Xalan can provide that context, and Apache Xerces in DOM mode is a suitable comparison candidate, especially as it offers a somewhat lazy mode of operation. As existing test suites do not span a wide coverage range, a custom test generator is required. The next section covers the generic architecture of a lazy XML parser.

3. ARCHITECTURE

We first discuss a generic architecture for lazy XML parsing. Subsequently, we refine the architecture to prepare the ground for a DOM implementation.

3.1 Generic Architecture

As discussed in the previous section, XML is not immediately suitable for partial parsing. Some form of preprocessing is required to locate closing tags and determine the nesting structure. Consequently, the process of lazy parsing can be decomposed into two phases (see fig. 3).

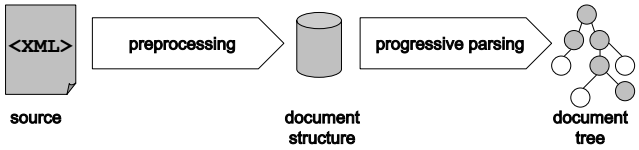

Figure 3: Lazy XML parser architecture

We illustrate the operation of each phase using the following XML document:

```
01 <article title="Of mice and men">
02   <section title="Unrelated work">
03     Grouped by subject.
04     <section title="Of mice">
05       Mice are rather small and felty.
06     </section>
07     <section title="Of men">
08       Men and women are rather tall.
09     </section>
10   </section>
11   <section title="Our work">
12     The tribulations of buckers, hands, tramps,
13     and swampers in the Great Depression.
14   </section>
15 </article>
```

The *preprocessing* phase skims the source to extract the document structure tree. This representation stores *node types* and *hierarchical relationships* between nodes as well as references to a *textual representation* for every node. The possible types are element, text, commment, processing instruction, document type or entity reference. Hierarchy amounts to storing references to all children of an element. For non-elements, hierarchy is trivial. The textual representation allows future progressive parsing. Should consumers actually access a node later on, detailed information can

then be parsed irrespective of the node environment. Conceptually, the resulting structure is a tree attributed with types and source references (see fig. 4).

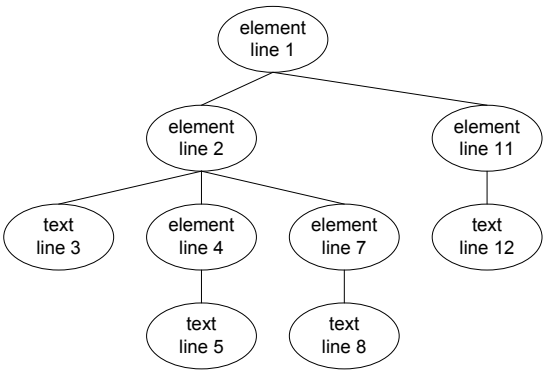

Figure 4: Sample structure tree

The *progressive parsing* phase constructs the highly more detailed *virtual* document tree on demand. Initially, only the root node of the virtual tree exists. All remaining nodes are virtual. Consumers can invoke methods on existing nodes to retrieve attributes and children. As parts of the document are requested by the consumer, the respective parts of the source document are parsed and the corresponding nodes in the virtual document tree are created.

Consider a consumer that creates high-level outlines which list all toplevel sections in an article. Fig. 5 shows the fragment of the virtual document tree created during the execution of this consumer.

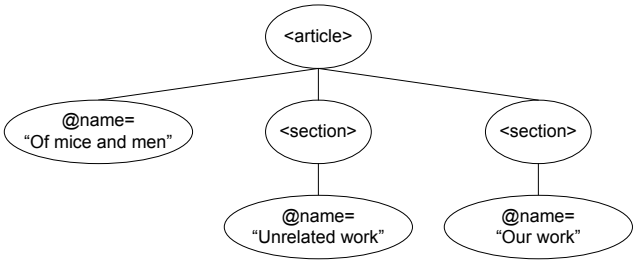

Figure 5: Sample virtual document tree

In this design, the concerns of preprocessing and progressive parsing are encapsulated separately. An implementation of preprocessing may thus be reused for various progressive parsers, independent of the interfaces they provide for tree access. We can build a virtual document tree with a DOM interface as well as arbitrary proprietary ones.

3.2 Refining the Architecture

Now, we are ready to discuss a concrete architecture for low-level access to trees. In this case, the document tree interface offers three categories of basic operations on node objects: hierarchical navigation, namespace prefix resolution and data retrieval. We examine their characteristics and requirements before choosing a design.

Hierarchical navigation retrieves parent, sibling or descendant nodes of a given node. Information about the document structure is the only prerequisite to this task, i.e., it can be accomplished without further parsing by solely relying upon the information delivered by the preprocessing phase.

Namespace prefix resolution maps a prefix to a namespace declared in the same element or an ancestor. As far as the enclosing elements are still unparsed, resolution triggers parsing to retrieve the necessary information.

Data retrieval operations obtain information about the node itself, i.e., tag names, attribute names and values etc. The required information is local to any given node.

Hierarchical navigation and namespace prefix resolution require information not directly available to individual node objects. Consequently, these requests cannot be properly handled by the node objects themselves. They have to be processed by *delegation*. Because the document tree root is the only node initially available, we choose to make all preprocessing data available through it. Non-local requests to other nodes are delegated to the document root.

This does not necessarily apply to data retrieval. In most cases, a node needs to be parsed before object creation. The information to process data retrieval requests is thus available when creating a node object. One can freely choose to either augment node objects with this information or to realize data retrieval operations by delegation consistent with the above approach. We choose the second alternative for reasons of consistency and flexibility. E.g., tag and attribute names are stored in the root.

Consistent use of delegation keeps the node objects lightweight. Most data structures reside centrally with the root node object, including bookkeeping data to track the progress of parsing and references to node objects already created. Lazy parsing is a centralized affair.

4. IMPLEMENTATION

In this section, we discuss a Java implementation of lazy parsing that conforms to the refined architecture from the previous section. Fig. 6 further refines the architecture.

As mentioned above, the choice of virtual tree interface is an arbitrary one. Therefore, we will give details of the most salient data structures in preprocessing, but we do not discuss progressive parsing in depth.

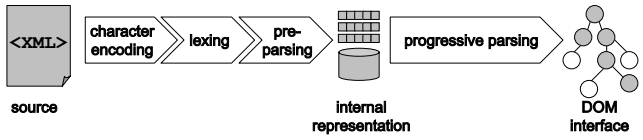

Figure 6: Lazy XML parser implementation

4.1 Preprocessing

In the preprocessing phase, our implementation analyzes the document structure and builds an internal representation suitable for partial parsing. As depicted in the left half of fig. 6, the phase consists of three steps. A pluggable *reader* converts the various character encodings permitted in XML sources to a stream of Unicode characters. A simple and fast *lexer* assembles a token sequence from this character stream. This token sequence controls the recursive descent of a *pre-parser*, which is responsible for building the internal representation.

In the course of recursive descent, nodes are indexed sequentially in depth-first pre-order, or document order. The

node index assigned in this way serves to uniquely identify individual nodes. For each node, recursive descent retains a reference to either the next sibling, or the parent if there are no following siblings. Furthermore, we retain node types and textual representations.

Because object creation is the single most expensive operation in Java, our implementation departs from the architectural view of structure tree nodes as individual objects. Instead, we chose a non-interleaved memory representation that employs three contiguous arrays and a string buffer. All arrays are accessed via the node index and grown dynamically during construction. Fig. 7 shows the internal representation of the sample document from section 3.

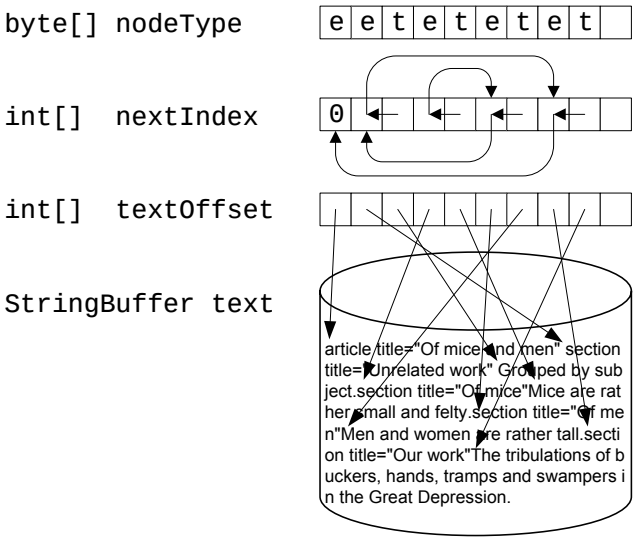

Figure 7: Internal structure representation

The `nodeType` type array stores node types. In this implementation, we choose to ignore document type nodes. Therefore, there are only six different node types: elements with and without children, text nodes, comments, processing instructions and entity references. For clarity, we choose a byte array instead of bit vectors.

The `nextIndex` array retains the hierarchy, i.e., it stores the node index of the following sibling or parent as discussed above. As parents have lower node indices than their children, the two kinds of references can be distinguished.

The string buffer contains Unicode sequences for progressive parsing. Data for all nodes are concatenated in document order. As node types and structure are already decoded, the delimiting character sequences `<`, `</`, `<!--`, `<?` and their counterparts are omitted as well as closing tags. To resolve strings for individual nodes, the `textOffset` array stores the index into the string buffer for every node.

This representation consumes 9 bytes per node, plus the space for the textual representation. In our implementation the input stream is decoded prior to structure analysis, i.e. the textual representation has to be stored as a Unicode string consuming 16 bits per character.

Assuming that document authors generally chose efficient character encodings, e.g., UTF-8 for English text or Shift-JIS for Japanese, we could refine lazy processing even further. In that scenario, storage can be saved by computing the token sequence from the raw input stream and deferring character decoding to progressive parsing. Due to the need for multiple scanners, we have not pursued this path.

4.2 Progressive Parsing

To fit into the measurement framework, we had to provide a DOM interface to the virtual document tree. The Java bindings of DOM are defined by the Java API for XML Processing, short JAXP [26]. As XSLT never writes on its input, we confined ourselves to a read-only DOM. Additionally, we included functions to measure input coverage by gathering access profiles.

5. MEASUREMENTS

We are interested in the relative performances of lazy tree construction and the traditional eager approach to parsing and building document trees.

Lazy tree construction avoids parsing and constructing parts of the tree that are never accessed. We expect its performance to be superior for sparse coverage. However, the dynamic adaptation mechanism requires additional bookkeeping and delegates numerous API calls to the central `Document` object. We expect these overheads to worsen performance for full coverage or even repeated accesses. As a weak lazy approach, Xerces deferred node expansion should fall somewhere in between the eager and lazy approaches.

To validate these expectations, we have to measure execution times for a full range of coverage. In these data, we are particularly looking for the break-even point between the lazy and traditional approaches.

5.1 Test Suite Generator

As stated in section 2.5, the benchmarks in existing test suites do not cover a full range of coverage. We thus developed a custom benchmark generator. Implemented in Java, it generates custom *test cases* for XSLTMark from configuration templates. Each test case consists of an XML source document, two sets of XSLT scripts and a configuration file.

The *source document* is generated from a template, an XML file with nested elements and text which describe the overall structure of the output document. Whereas text is simply copied from the template to the output, an element can be further configured by specifying the tag name and the number of times its content is to be repeated.

The *XSLT scripts* describe simple transformations that copy portions of the input document to the output document. Both sets of transformations span a full range of coverage. The first set uses limits on traversal breadth to achieve partial coverage, the second one employs limits on depth. In this way, we accumulate access profiles of different kinds for gradually increasing coverage.

The *configuration file* inserts test cases into the benchmark framework. It specifies a list of transformations to be executed and timed on varying inputs.

5.2 Benchmark

We compiled a benchmark running our lazy DOM implementation against an eager implementation. For this comparison, we choose Apache's Xerces Java Parser release 1.3.0, a freely available traditional XML parser that supports the DOM API and shows fairly good performance. Both traditional and deferred modes were tested.

We generated several disparate test cases with the tool described above. Within our Java benchmark framework, the

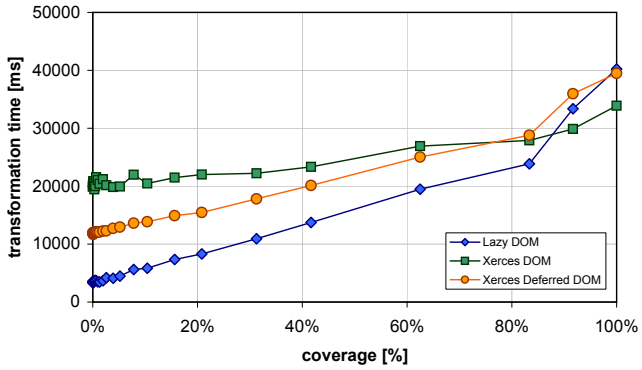

Figure 8: Depth-limited processing time

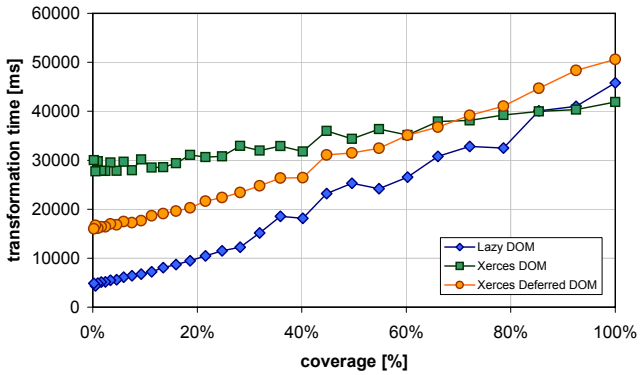

Figure 9: Breadth-limited processing time

individual transformations were applied using the Apache Xalan XSLT transformer [2]. This well established standard implementation is capable of operating on DOM input. Unfortunately, current versions of Xalan convert the entire input tree to an internal representation, resulting in full coverage. Therefore, we restricted ourselves to Xalan version 2.0.0. The input trees were provided by Xerces running in DOM mode and our lazy DOM parser, respectively.

For each transformation and DOM implementation respectively, the cumulative runtime of ten subsequent iterations was measured, thereby compensating for the coarse resolution of the system clock. In order to bypass disturbances caused by the JVM's just-in-time compilation, we conducted blind runs prior to the actually measured executions. Coverage was measured by gathering access profiles for the virtual document tree.

We ran our measurements on a 600 MHz Pentium III with 256 MB of memory under Windows 2000 Professional and Sun Java 2 SDK, Standard Edition, version 1.3.1.

5.3 Results

Fig. 8 and 9 show processing times for depth-limited and breadth-limited test cases, respectively. The depth-limited test case comprises a binary XML tree of 1,053 KB source and a nesting depth of 14 levels. The breadth-limited case contrasted this with a tree nesting to 4 levels with a branching factor of 30 per level and 1,417 KB source. The resulting graphs are similar in other cases.

For zero coverage, lazy parsing outperforms the eager approach by a considerable 80%. This lead continuously declines for increasing coverage. Break-even between lazy and eager processing occurs between 80% and 95% coverage. Lazy processing incurs an overhead of about 10-15% for full coverage. Xerces deferred node expansion consistently performs worse. Its trade-off point versus eager processing is lower, at around 70%. These overall results are independent of traversal limits, though the lazy strategy appears to be most convincing for shallow breadth limits. In all, the numbers validate our initial expectations.

Fig. 8 and 9 show memory consumption for the above benchmarks. For zero coverage, lazy processing again outperforms the eager approach by 65%. The lead declines with increasing coverage, reaching parity around 45%. For full coverage, lazy processing incurs a memory overhead of 70-100% over eager processing. Memory consumption for Xerces deferred node expansion lies near the average of lazy and eager processing, with a similar tradeoff point as for lazy processing.

6. CONCLUSIONS

For both depth- and breadth-limited transformations, processing times for the lazy approach are approximately linear in the percentage of nodes visited. In both traversal cases, the lazy approach breaks even with eager processing at 80% coverage. Thus, coverage uniquely determines the tradeoff between lazy and eager processing in our test cases.

Limits to breadth and depth are highly disparate cases. In general, path-based tree traversals like those induced by XSL transformations lie somewhere between the two ex-

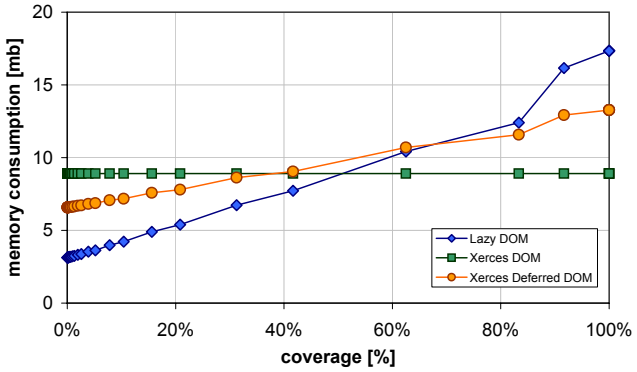

Figure 10: Depth-limited memory consumption

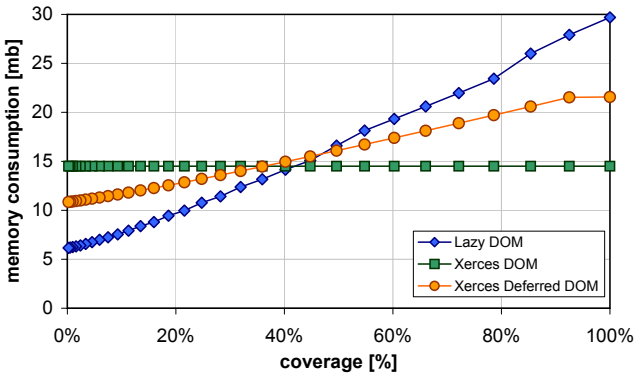

Figure 11: Breadth-limited memory consumption

tremes. We therefore propose the following strategy to maximize performance: Choose a lazy implementation if coverage does not exceed 80%, otherwise an eager one.

In low-footprint devices, the memory overhead of lazy processing at high coverage may be a problem. As sketched in the final paragraphs of section 4.1, extending the lazy approach to character set decoding can alleviate this problem if document authors use suitable character encodings.

Unfortunately, most XSLT scripts in standard test suites actually visit each and every node in a document, i.e., they exhibit full coverage. Partially, this is due to their dual purpose. These suites simultaneously benchmark and validate processors, so complete processing is actually desired. In many cases, disabling superfluous default processing rules goes a long way to ensure sparse coverage in XSLT scripts.

Moreover, in many cases the mapping of XPath steps to DOM operations destroys sparseness. DOM does not support proper filtering, so a sparse transformation on the XSLT level often maps to a complete DOM tree traversal. This problem is inherent to DOM and may only be eliminated by adopting a new interface.

Once we have completed implementing lazy XSL transformations, our future work will center on such a new interface. We aim to provide uniform access to both deserialization and transformation processors, while preserving sparseness to improve networked performance.

When lazy XSL transformation is available, we also aim to explore the use of infinite intermediate documents. How these structures can contribute to clean and efficient designs remains an open question for XML processing networks. Exploring the related idioms of functional programming languages may be a promising direction here.

7. ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers for their diligence and dedication to quality. Your insightful comments were really helpful in enhancing this paper.

8. REFERENCES

[1] *Xerces Java Parser*. Apache XML Project, `http://xml.apache.org/xerces-j/`.

[2] *Xalan Java XSLT Processor*. Apache XML Project, `http://xml.apache.org/xalan-j/`.

[3] *aXMLerate Project*. B2B Group, University of Karlsruhe, `http://i44pc29.info.uni-karlsruhe.de/B2Bweb/`.

[4] L. Bass, P. Clement, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[5] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[6] J. Clark. *XP - an XML Parser in Java*. `http://www.jclark.com/xml/xp/`, 1998.

[7] J. Clark. *XT*. `http://www.jclark.com/xml/xt/`, 1999.

[8] J. Dieterich. *Generierung von Graphersetzern als XML-Transformatoren*. Universität Karlsruhe, IPD Goos, Jun 2001.

[9] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[10] M. Kay. *The SAXON XSLT Processor*. `http://saxon.sf.net/`, 2001.

[11] S. Kleene. General recursive functions of natural numbers. *Math. Ann.*, 112:729–745, 1936.

[12] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[13] W. Löwe, M. L. Noga, and T. S. Gaul. Foundations of fast communication via XML. *Annals of Software Engineering*, 13(1–4):357–379, Jun 2002.

[14] W. Löwe and M. L. Noga. Scenario-based connector optimization. In *LNCS 2370, IFIP/ACM CD 2002*, pages 170–184. Springer, Jun 2002.

[15] J. McCarthy. *History of LISP*, pages 173–197. Academic Press, New York, 1981.

[16] M. L. Noga. *Erzeugung validierender Zerteiler aus XML Schemata*. Universität Karlsruhe, IPD Goos, `http://www.noga.de/markus/XMLSchema/Diplomarbeit.pdf`, Oct 2000.

[17] *Simple API for XML Processing*. Megginson et. al., `http://www.saxproject.org/`, 2002.

[18] T. Schmitt-Lechner. *Entwicklung eines XSLT–Übersetzers*. Universität Karlsruhe, IPD Goos, May 2001.

[19] M. Shaw and D. Graham. *Software Architecture in Practice – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[20] *Document Object Model*. W3C, `http://www.w3.org/DOM/`, 2000.

[21] *XML Information Set*. W3C Working Draft 26 July 2000, `http://www.w3.org/TR/2000/WD-xml-infoset-20000726`, 2000.

[22] *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502`, 2001.

[23] *XML Schema Part 2: Datatypes*. W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-2-220010502`, 2001.

[24] *Extensible Markup Language (XML) 1.0*. W3C Recommandation, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[25] *XSL Transformations (XSLT)*. W3C Recommandation, `http://www.w3.org/TR/xslt`, 1999.

[26] *Java API for XML Processing*. Sun Microsystems Inc., 1.1.3 edition, 2001.

[27] *XML Parser for Java*. IBM AlphaWorks, `http://alphaworks.ibm.com/aw.nsf/techmain/xml4j`, 2001.

[28] *XSLBench*. TFI Technology Ltd., `http://www.tfi-technology.com/xml/xslbench.html`, 2001.

[29] *XSLTMark*. DataPower Technology Ltd., `http://www.datapower.com/XSLTMark/`, 2001.