A Low-Memory Streaming Algorithm for XSLT Processing Implemented in Xord Framework

Jana Dvořáková and Filip Zavoral
Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague
{ Jana.Dvorakova, Filip.Zavoral } @mff.cuni.cz

Abstract

We present an implementation of the Xord framework for streaming processing of XSLT transformations based on .Net technologies. Within the framework, we implement an efficient streaming algorithm capable to process a significant subset of top-down XSLT transformations. We exactly characterize this class of transformations. The algorithm uses a stack of the size proportional to the depth of the input XML document. Such memory usage is highly efficient in practice, since real-world XML documents are shallow. The evaluation of the algorithm supports our expectations and shows that a transformation of large but shallow XML documents is processed using a constant size of memory.

1 Introduction

Common processors of XSLT [13] and XQuery [14] (Saxon, Xalan, AltovaXML) are tree-based, i.e., they store the whole input in the memory and then perform the transformation itself. Such approach is sufficient for small XML documents stored in files. However, this classical processing is apparently not suitable for large XML documents and XML data streams - in the former case, it is not acceptable or even possible to store the whole input document in the memory, while in the later one, the XML data become available stepwise and need to be processed “on the fly”. It is thus natural to employ the streaming processing, i.e., to read the input document sequentially in the document order as well as to generate the output document sequentially. It is easy to see that, for certain classes of XML transformations, such streaming processor is substantially less memory-consuming than the tree-based processors.

The simplest approach to obtaining a program for streaming XML transformation is to write it completely by hand in some common programming language using an event-based parser such as SAX or StAX. Another possibility is to specify the transformation in some low-level streaming transformation language such as STX [1] or XStream [7]. However, both of these approaches are cumbersome for two reasons:

- The user must be familiar either with an event-based API for XML processing or with a special streaming transformation language.
- An extra effort is needed in order to obtain efficient streaming algorithm as the result - it is necessary to analyze the transformation and to optimize the size of the memory buffers manually. Still, it is not guaranteed how efficient the designed algorithm is (the optimal solution might not be known).

In this paper, we are interested in automatic approach, i.e., we focus on the problem how to design an automatic streaming processor for commonly used transformation languages such as XSLT and XQuery. Since both languages are based on tree-manipulation, they contain several non-streaming constructs. The key problem is then to find a way how to handle processing of such constructs using input buffering.

The main contributions of this paper are the following:

- We present a modular implementation of the Xord framework for streaming processing of XSLT transformations based on .Net technologies. The implementation consists of the Template Model and the Algorithm Model. The framework has a solid formal base ([4, 5]). It is intended to contain several streaming algorithms. Each of them is first designed as a formal model called streaming XML transducer (SXT). For such model, the transformation class captured and the memory requirements are known.
- Using both the Template Model and the Algorithm Model, we implement an efficient streaming algorithm...
for processing a subset of top-down XSLT transformations. We exactly characterize this class of transformations. The algorithm uses a stack of the size proportional to the depth of the input XML document. Such memory usage can be considered constant in practice, since real-world XML documents are shallow [12].

- We mention experimental results for the algorithm implemented. The results show that the algorithm indeed consumes constant amount of memory when processing transformations from associated class on large and shallow XML documents.

The rest of this paper is organized as follows: In Section 2 we list automatic streaming processors for XSLT and XQuery. Section 3 presents the formal base of the framework. The class of top-down XSLT transformations considered is described in Section 4. In Section 5, we introduce the stack-based algorithm, its basic principles and implementation. The evaluation results are presented in Section 6. We conclude with summary and comments on future work.

2 State of the Art

Much of the previous work is devoted to streaming evaluation of XPath expressions, e.g. [2, 9]. The techniques used for XPath may be partially used also in the streaming algorithms for the transformations, but some significant problems are still to be solved (e.g., different order of the elements in the input document and in the output document). Several streaming processors for XSLT and XQuery have been implemented. However, their efficiency was demonstrated only by experiments on a small number of XML transformations and input XML documents. It is thus not known how much memory these processors consume for a given type of XML transformation. The only exception is the XSLT processor based on SPM [8] which by definition does not have any additional memory. Due to the lack of ability to store data temporarily the processor is significantly less powerful than the SSXT algorithm that we introduce in Section 5. Moreover, the class of transformations captured is not characterized.

XML Streaming Machine (XSM) [11] processes a subset of XQuery queries on XML streams. It is based on a model called XML streaming transducer. Authors have tested the processor on XML databases of various sizes against a simple XQuery query. They compared the time of the processing with common tree-based processors. The results show that using XSM the processing time grows linearly with the document size, while in the case of tree-based processors the time grows superlinearly. However, more complex XQuery expressions have not been tested.

BEA/XQRL [6] is a streaming XQuery processor that is a central component of BEA’s WLI product. It implements full XQuery. The authors have compared the processor with Xalan-J processor on the set of 25 typical WLI transformations as well as on XMark Benchmarks. BEA processor appeared to be fast on small databases, however, the processing of large databases was slower due to the fact that the optimizations specially designed for XML streams are limited in this engine.

FluXQuery [10] is a streaming XQuery transformation engine based on an internal query language called FluX extending XQuery with new constructs for streaming processing. XQuery query is converted into FluX and the buffer size is optimized by examining the query syntax and the input DTD. FluXQuery supports a subset of XQuery. According to the benchmarking against XQuery processors Galax and AnonX on selected queries of the XMark benchmark, the engine appeared to be less memory-consuming.

SPM (Streaming Processing Model) [8] is a one-pass streaming XSLT processor without an additional memory. Authors present a procedure that tries to converts a given XSLT stylesheet into SPM. No algorithm for testing the streamability of XSLT is introduced.

3 Formal base

The implementation framework for XSLT is based on the abstract framework introduced in [4, 5]. It consists of two groups of formal models. The basic general XML transducer (GXT) is used to model all (algorithmically computable) XML transformations and their tree-based processing. By imposing various restrictions on the GXT, different XML transformation classes can be defined. On the other hand, the basic streaming XML transducer (SXT) is used to model one-pass streaming processing without an additional memory. It can be extended by a memory to store temporary data or by allowing more passes over the input document. This way, we obtain models of several streaming processors.

When designing a new streaming algorithm within the framework, it is necessary to find a correspondence between a restricted GXT and an extended SXT. In this paper, we present one such algorithm in which we consider the simple order-preserving branch-disjoint GXT and the simple SXT (SSXT). The algorithm is called SSXT algorithm according to the streaming XML transducer employed.

The GXTs are language-independent and in case a conversion method between GXT and a specific language is found, an implementation framework for this language can be developed in a straightforward way starting from the abstract framework. In this paper, we focus on XSLT transformations and therefore we directly consider XSLT transformation classes instead of GXTs.

At the design level, the implementation framework for the XSLT language consists of two basic modules:
1. **Static analyzer** analyzes the given XSLT stylesheet \( xsl \) and determines to which of the known classes of XSLT transformations it belongs.

2. **Transformer** processes \( xsl \) using the algorithm associated with the determined transformation class.

### 4 Simple XSLT transformations

In the SSXT algorithm, we only consider simple XSLT transformations which are order-preserving and branch-disjoint.

Simple XSLT stylesheet contains an **initializing template** and several **transforming templates**. The initializing template sets the initial mode to \( m0 \) and call processing of the root element (node). It is of the form:

\[
\text{<xsl:template match="/"} >\text{<xsl:apply-templates mode="m0"/}>\text{</xsl:template>}
\]

A transforming template is called by an element name and a mode:

\[
\text{<xsl:template match="a" mode="m1">... template body ... </xsl:template>}
\]

The template body consists of output elements (possibly nested) and template calls which call application of other templates by an XPath expression and a mode. The output elements are of the form:

\[
\text{<element name> ... element body ... </element name>}
\]

The template calls are of the form:

\[
\text{<xsl:apply-templates select="child::a/child::b/descendant::c" mode="m2"/>}
\]

A subset of XPath expressions is allowed in transforming templates - they may contain only child and descendant axes, and they may select nodes by name only:

\[
\begin{align*}
\text{XPath} & \ := \ \text{Step} \mid \text{Step/XPath} \\
\text{Step} & \ := \ \text{Axis::name} \\
\text{Axis} & \ := \ \text{child} \mid \text{descendant}
\end{align*}
\]

where \( \text{name} \) refers to an element name. The evaluation function for XPath expression \( \text{exp} \) with respect to the XML document \( d \) and one of its nodes \( u \) is denoted by \( \text{eval}(\text{exp}, d, u) \). The semantics of the \( \text{eval} \) function directly follows the semantics of evaluation of XPath expressions, the only difference is that in our case it is sufficient to consider a single node as the current context set.

In order to determine whether a stylesheet is simple, it must be checked that it conforms the structure described. Now we described order-preserving simple XSLT transformations and branch-disjoint simple XSLT transformations.

In case a simple XSLT transformation does not conform to these conditions, the SSXT algorithm is not applicable - moreover, in majority of cases, additional memory buffers are needed in order to process the transformation.

**Order-preserving simple XSLT.** Let \( \text{tmp} \) be a transforming template, \( d \) be an XML document, and \( u \) be a node of \( d \). We define an auxiliary function \( \text{eval-exp} \) by

\[
\text{eval-exp}(\text{tmp}, d, u) = \text{eval}(\text{exp}_1, d, u) \ldots \text{eval}(\text{exp}_n, d, u)
\]

where \( \text{exp}_1, \ldots, \text{exp}_n \) is a sequence of XPath expressions appearing in the template calls of \( \text{tmp} \) (in this order). Thus, the \( \text{eval-exp} \) function returns the concatenation of the node sequences returned by individual XPath expressions.

A simple XSLT \( xsl \) is order-preserving on a set of XML documents \( \mathcal{D} \) if and only if,

- for each transforming template \( \text{tmp} \) of \( xsl \),
- for each XML document \( d \in \mathcal{D} \),
- for each node \( u \) of XML document \( d \),

it holds \( \text{eval-exp}(\text{tmp}, d, u) \) returns a sequence of nodes of \( d \) in document order.

**Branch-disjoint simple XSLT.** A simple XSLT \( xsl \) is branch-disjoint on a set of XML documents \( \mathcal{D} \) if and only if,

- for each transforming template \( \text{tmp} \) of \( xsl \),
- for each XML document \( d \in \mathcal{D} \),
- for each node \( u \) of XML document \( d \),

it holds \( \text{eval-exp}(\text{tmp}, d, u) \) does not contain two nodes located within the same branch of \( d \).

**Xord Template Model.** In the Xord framework, XSLT stylesheets are represented by a set of classes, an **Xord Template Model**. Each template from the XSLT contains a sequence of template calls. A template call consists of the parsed XPath expression and the template called by the \(<\text{apply-templates}>\) mechanism. The input template file is parsed into these structures before the analysis. Then the analysis algorithm directly traverses the DAG, evaluates the expressions etc.

### 5 SSXT algorithm

The simple streaming XML transducer has a single input head that reads the input document sequentially, and a single output head that generates the output document sequentially. The SSXT is equipped with a stack to store temporary data.

The SSXT takes an input document \( d_{in} \) and a top-down XSLT stylesheet \( xsl \) as the input. It reads \( d_{in} \) sequentially
in one pass and apply the stylesheet xsl stepwise. First, the template matching the root element of $d_{in}$ in the initial mode $\nu_0$ is set to be the currently processed template (current template). The processing proceeds in cycles. During a single cycle, a single template call of the current template is processed.

**Processing cycle.** All XPath expression within a template are evaluating concurrently. The evaluation is realized by deterministic finite automata (DFA)\(^1\). A single DFA is constructed for each expression. When the processing of a template starts, the sequence of the initial states of DFAs is pushed on the stack. The input head of SSXT reads the elements of $d_{in}$ in document order. When a start-tag is encountered, new sequence of DFAs is computed. Three situations may occur:

a) new sequence contains no final state - the input head continues in evaluation,

b) new sequence contains a single final state which belongs to the DFA evaluating the last matched expression or an expression located after the last matched expression - the corresponding template call is processed,

c) new sequence contains a final state which belongs to the DFA evaluating expression located before the last matched expression, or it contains two or more final states - error.

In case b), the current cycle configuration (template id, matched expression id) is pushed on the stack and new cycle for processing the called template starts. The cycle configuration is popped after the whole called template has been processed and the control moves back to the current template. In case a), the evaluation continues. Here if an end-tag is encountered, the sequence of the DFA states located at the top of the stack is popped. Hence, the XPath expression of the current template are evaluated on “branches” of $d_{in}$.

### 5.1 Implementation

The implementation of the SSXT algorithm follows the Xord Ssxt Model (see Fig. 1) and it uses both the Template Model and the Algorithm Model classes. The Template Model was described in Section 4. The Algorithm Model is an abstraction of all streaming algorithms to be designed within the Xord framework. Since the algorithm is stack-based, the main data structure used is a polymorphic stack $stk$ of type $XStack$ which stores sequences of DFA states ($SIDfaSequence$) and cycle configurations ($SICycleConfig$).

\(^1\)We refer the reader to [3] for a more detailed description of this evaluating method.

![Figure 1. The Xord Ssxt Model.](image)

Until the transformation is finished the top of stack is checked and the stack item is processed, see the method $Ssxt::Run$ below. In case of an empty stack and nonempty remaining input new DFA sequence is pushed on the stack.

```cpp
void Ssxt::Run(XfXml xml)
{
    XfTemplate currTemplate = xslt.Start();
    XfCall currCall = null;
    bool transformed = false;
    while(!transformed) {
        stk.Top.Process();
        if(!stk.Empty()) {
            stk.Top.Process();
            break;
        } else {
            switch(xml.currType) {
            case XmlNodeType.Element:
                stk.Push(new SIDfaSequence(currTemplate));
                xml.Advance();
                break;
            case XmlNodeType.EndElement:
                currTemplate.Generate(currCall, null); 
                transformed = true;
                break;
            }
        }
    }
}
```

The processing of the DFA sequences is accomplished by the method $SIDfaSequence::Process$ below. The core of the processing is accomplished when start tags of elements are encountered. A new DFA sequence is generated on the stack in case the current DFA sequence contains no final states. Otherwise the output is generated and a new cycle configuration is placed on the stack. In case of a template without calls, its output is generated immediately.
void SIDfaSequence::Process(XfXml xml)
{
    SIDfaSequence ds = stk.GetDfaSequence();
    switch (xml.currType)
    {
    case XmlNodeType.Element:
        SIDfaSequence new_ds = ds.Transition(xml.currName);
        if (!new_ds.HasFinalStates()) {
            stk.Push(new_ds);
            xml.Advance();
        } else {
            XfCall myCall = new_ds.GetCallWithFinalState();
            currTemplate.Generate(currCall, myCall);
            XfTemplate calledTemplate = xslt.SelectTemplate(xml.currName, myCall.mode);
            if (calledTemplate.Empty) {
                calledTemplate.Generate(null, null);
                currCall = myCall;
                if (xml.laType == XmlNodeType.Element)
                    stk.Push(new_ds);
                xml.Advance();
            } else {
                stk.Push(new SICycleConfig(currTemplate, myCall));
                currTemplate = calledTemplate;
                currCall = null;
            }
        }
        break;
    case XmlNodeType.EndElement:
        if (xml.laType == XmlNodeType.Element) {
            stk.Pop();
            xml.Advance();
        }
        break;
    default:
        stk.Pop();
        break;
    }
}

The cycle configuration processing (method SICycleConfig::Process below) depends on the current XML node type. A start tag pushes a new DFA sequence while an end tag generates output.

void SICycleConfig::Process(XfXml xml)
{
    SICycleConfig cc = stk.GetCycleConfig();
    switch (xml.currType)
    {
    case XmlNodeType.Element:
        if (xml.laType == XmlNodeType.Element)
            stk.Push(new SIDfaSequence(currTemplate));
        xml.Advance();
        break;
    case XmlNodeType.EndElement:
        currTemplate.Generate(currCall, null);
        currTemplate = cc.template;
        currCall = cc.call;
        stk.Pop();
        break;
    }
}

Memory usage. In the SSXT algorithm, a single sequence of DFA states is pushed on the stack when reading start-tags and no match is found, and a single sequence of DFA states is popped from the stack when reading end-tags and moving upwards in the element hierarchy. The sequences are obviously of constant length since the number of states in a sequence depends on the number of XPath expressions in the templates of xsl.

A new processing cycle starts when a match is found for some template call. Here an extra item - a cycle configuration - is pushed on the stack. However, when returning from processing the call, the cycle configuration is popped. Based on this observation, it is easy to see that the size of the stack never exceeds the number (depth of $d_{in} + 2$).

We treat two boundary situations in a special way - processing templates that do not contain any template calls and processing matches at leaves of $d_{in}$.

Static analysis. The static analyzer for SSXT algorithm currently checks the order-preservation and branch-disjointness of simple XSLT stylesheet in case the set of input XML documents is not restricted by a schema, i.e., any XML document over a given alphabet of element names may appear as the input.

6 Evaluation

We have evaluated the SSXT algorithm against the publicly available tree-based XSLT processors Saxon and Xalan using both synthetic and real data. The evaluation showed that the SSXT algorithm requires a memory proportional to a depth of the input XML document. Since this depth is generally not depending on the document size and documents are relatively shallow (99% of documents have fewer than 8 levels whereas the average depth is 4 according to [12]), our memory requirements for most of the XML documents are constant, independent to the document size. On contrary, XSLT processing using standard processors like Xalan or Saxon uses DOM for the whole document which implies memory requirements proportional to the document size. Our measurements confirmed this expectation, Fig. 2 shows a comparison of transformation memory requirements of 10000 to 1 million entities.

We have not included comparison to the streaming processor SPM [8] and to the transformation programs written manually using some event-based parser. In the first case, experiments were not possible since the SPM does not capture the class of top-down XSLT transformations considered. In the second case, the comparison would be pointless since the effectiveness of a hand-written program depends solely on the programmer skills and thus may vary.

7 Conclusion

We introduced an implementation of the Xord framework for efficient XSLT processing on .NET platform. The functionality of the framework is currently based on the stack-based streaming algorithm which is able process a class of top-down XSLT transformations using stack of the size proportional to the depth of the input document. In practice, the stack size can be considered constant since
real XML document typically contain only few levels of elements. We supported this observation by experiments - the results show that the algorithm is much less memory-consuming comparing to the standard DOM processing. Moreover, the algorithm is much more powerful than existing streaming algorithms for XSLT (SPM [8]).

Several issues are left for the future work. We intend to modify the current algorithm to a buffer-based streaming algorithm in order to make it possible to process more complex transformations. At the same time, we plan to implement static schema-based analyzer which, for a given XSLT stylesheet and schema, determines which algorithm is the most convenient one to apply.

Acknowledgements

This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838) and by the grant VEGA 1/3106/06. A part of the results presented comes from a PhD thesis of Comenius University in Bratislava, Slovakia.

References