View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

A Prototype for Translating XQuery Expressions into XSLT Stylesheets

Niklas Klein[1], Sven Groppe[1], Stefan Böttcher[1], and Le Gruenwald[2]

[1] University of Paderborn, Faculty 5,
Fürstenallee 11,
D-33102 Paderborn, Germany
{niklask, sg, stb}@uni-paderborn.de
[2] University of Oklahoma,
School of Computer Science,
Norman, Oklahoma 73019, U.S.A
ggruenwald@ou.edu

Abstract. The need for a user-friendly query language becomes increasingly important since the introduction of XML. The W3C developed XQuery for the purpose of querying XML data, but XQuery is not available in every tool. Because of historical reasons, many tools only support processing XSLT stylesheets. It is desirable to use tools with XQuery, the design goals of which are, among other goals, to be more human readable and to be less error-prone than XSLT. Instead of implementing XQuery support for every tool, we propose to use an XQuery to XSLT translator. Following this idea, XQuery will be available for all tools, which currently support XSLT stylesheets. In this paper, we propose a translator which transforms XQuery expressions into XSLT stylesheets and we analyze the performance of the translation and XSLT processing in comparison to native XQuery processing.

1 Introduction

1.1 Problem Definition and Motivation

With the wide-spread use of the Extensible Markup Language (XML) accompanied with increasing document sizes, there is an increasing need for user-friendly XML query languages. While the Extensible Stylesheet Language Transformations (XSLT) [11], which also can be used as a query language, is established in the market for years, XQuery [12] is relatively new.

Whereas XSLT is conceived as a transformation language, XQuery was aimed to be an easy human readable query language. Furthermore, both languages are used to grab, filter and associate data from XML-documents. There exists already a large repository of tools, especially commercial products, for supporting XSLT, but not the XQuery language. Examples of such products are BizTalk [8], Cocoon [1] and Xalan [2]. Whenever an application based on these tools is required to use XQuery as the XML query language, it is a big advantage to

J. Eder et al. (Eds.): ADBIS 2005, LNCS 3631, pp. 239–254, 2005.
© Springer-Verlag Berlin Heidelberg 2005

have a translation from XQuery expressions to XSLT stylesheets such that the XQuery language can be used.

Although both languages were developed with different aims, their application possibilities and expressive power are similar. Both languages use XPath as the path-language for retrieving XML node sets, and both languages have corresponding language constructs for the iteration on an XML node set, the definition of variables, XML node constructors and the definition and call of user-defined functions. However there are some differences between the two languages which we will discuss in Section 2.3.

In this paper, we propose a translation tool from XQuery expressions into XSLT stylesheets that covers the XQuery language except for a few exceptions.

The rest of this paper is organized as follows. Section 2 provides a comparison of XQuery and XSLT. Section 3 describes how we would translate XQuery expressions into XSLT stylesheets. Section 4 presents experimental results comparing the execution times of XSLT stylesheets translated by our approach with the execution times of direct executed XQuery expressions. Finally, Section 5 concludes the paper.

1.2 Related Work

There exists works that compare the languages XSLT and XQuery. [7] shows that many XQuery constructs are easily mappable to XSLT, but presents only examples of mappings and does not provide an algorithm for translating XQuery expressions into XSLT stylesheets. [6] introduces an algorithmic approach of translating XQuery expressions into XSLT stylesheets, but includes neither a detailed algorithm for a subset of XQuery nor a report on experimental results.

Saxon [5] is a processor for both, XQuery expressions and XSLT stylesheets. First, Saxon translates an XQuery expression or an XSLT stylesheet into an object model, where most but not all components are common for XQuery and XSLT. After that, Saxon executes the objects of the object model in order to retrieve the results, but does not provide a source to source translation so that XQuery can be used in XSLT tools.

In this paper we describe a detailed algorithm for translating a subset of XQuery expressions into XSLT stylesheets. Furthermore, we give a detailed performance analysis of the execution of the original XQuery expession compared to the execution of the translated XSLT stylesheet.

2 Comparison of XQuery and XSLT Features

2.1 XQuery Essentials

XQuery is a *functional language*,which means that expressions can be nested with full generality. XQuery is also a *strongly-typed language* in which the operands of various expressions, operators, and functions must conform to the expected types.

XQuery embeds XPath as the path language to locate XML nodes in XML structures. An XPath expression itself is a simple XQuery expression. Furthermore, the XQuery language extends the XPath language by constructors for XML structures like elements and attributes, by FLWOR expressions, which can combine and restructure information from XML documents, by user-defined functions and many more language elements.

FLWOR is an acronym, standing for the first letters of the clauses that may occur in an FLWOR expression:

- `for` clauses associate one or more variables to expressions, creating a tuple stream in which each tuple binds a given variable to one of the items to which its associated expression evaluates. There can be an arbitrary amount of `for` clauses.
- `let` clauses bind variables to the entire result of an expression. There can be an arbitrary number of `let` clauses, but there must be at least one `let` or `for` clause.
- `where` clauses filter tuples, retaining only those tuples that satisfy a condition. The `where` clause is optional.
- `order by` clauses sort the tuples in a tuple stream. The `order by` clause is optional.
- `return` clauses build the result of the FLWOR expression for a given tuple. The `return` clause is required in every FLWOR expression.

2.2 XSLT Essentials

The W3C developed the declarative language XSLT, which describes the transformation of XML documents into a document formulated in XML, HTML, PDF or text by template rules. An XSLT stylesheet itself is an XML document with the root element `<xsl:stylesheet>`. The `xsl` namespace is used to distinguish XSLT elements from other elements. Template rules are expressed by an `<xsl:template>` element. Its `match` attribute contains a pattern in form of an XPath expression. Whenever a current input XML node fulfills the pattern of the `match` attribute, the template is executed. An XSLT processor starts the transformation of an input XML document with the current input XML node assigned to the document root. Using a short form, the output of the executed template is the XML nodes, which are not XSLT instructions, and the text inside the executed template. This output can also be described by a long form with the XSLT instructions `<xsl:element>` for generating XML elements, `<xsl:attribute>` for generating attributes of an XML element and `<xsl:text>` for generating text. Output is also described by the XSLT instruction `<xsl:value-of>`, which converts the result of an XPath expression to a string. The XSLT instruction `<xsl:apply-templates>` recursively applies the templates to all XML nodes in the result node set of the XPath expression given by its `select` attribute. We refer to [11] for a complete list of XSLT instructions.

2.3 Comparison of the XQuery and the XSLT Data Model and Language Constructs

XSLT 2.0 and XQuery 1.0 are both based on the XPath data model [3] and both embed XPath as the path language for determining XML node sets. Therefore, a majority of the XQuery language constructs can be translated into XSLT language constructs and vice versa. For example, `xsl:for each` has similar functionality as `for`, `xsl:if` has similar functionality as `where`, and `xsl:sort` has similar functionality as `order by`. However there are some differences between the two languages which we will discuss here.

Differences in handling intermediate results: XQuery and XSLT handle intermediate results differently.

- Whereas XQuery expressions can be nested with full generality, most XSLT expressions cannot be nested. Therefore, nested XQuery expressions must be translated into a construct, where the intermediate results of the nested XQuery expression are first stored in an intermediate variable using the `xsl:variable` XSLT instruction. After that the intermediate variable is referred for the results of the nested XQuery expression. XSLT variables, which are defined by `xsl:variable`, can only store element nodes. In particular, XSLT variables cannot store attribute nodes, comment nodes and text nodes. Whenever the translated XSLT stylesheets have to store other XML nodes besides element nodes, the translation process can use the work-around presented in Section 2.4.
- Both XQuery and XSLT embed XPath 2.0, which contains the `is` operator. This operator compares the two nodes identities. In the underlying data model of XQuery and XSLT, each node has its own identity. XQuery expressions never copy XML nodes, but always refer to the original XML nodes. Contrary to XQuery expressions, XSLT expressions can only refer in variables to original XML nodes, which can be described by an XPath expression `XP` and when using the `<xsl:variable select="XP">` instruction. While computing the result of more complex XSLT expressions, which contain functionality outside the possibilities of XPath like iterating in a sorted node set `XP` by `<xsl:for-each select="XP"><xsl:sort/>...</xsl:for-each>`, XSLT expressions have to copy XML nodes by using `xsl:copy` or `xsl:copy-of`, where the copied XML nodes get new identities different from those of the original XML nodes or other copied XML nodes. Therefore, whenever an XQuery expression uses the `is` operator and variables store a node set that cannot be expressed by an XPath expression, the translation process must offer a work-around, which ensures that the identities of XML nodes in the translated XSLT stylesheet to be considered in the same way as the identities of XML nodes in the original XQuery expression. Section 2.5 describes such a work-around.

Differences in language constructs: The translation process must consider the following differences in the language constructs of XQuery and XSLT:

- Whereas XQuery binds parameters in function calls by order of appearance, XSLT binds parameters of calls of functions and of named templates by parameter names.
- The `order by` construct of XQuery corresponds to `xsl:sort`. XQuery supports four order modifiers: `ascending`, `descending`, `empty greatest` and `empty least`. XSLT supports only `ascending` and `descending`. Therefore, `empty greatest` and `empty least` can not be translated yet. Furthermore `xsl:sort` has to be the first child of the surrounding `xsl:for-each` XSLT instruction. The `order by` clause can contain a variable `$v`, which is defined after the `for` expression. Therefore, the translated variable definition of `$v` occurs after the `xsl:sort` instruction, which must be the first child of `xsl:for-each`, but translation of `$v` is defined later in the translated XSLT stylesheet and cannot be used in the `xsl:sort` instruction. In the special case where the variable `$v` is defined by an XPath expression `XP`, we can replace the reference to the translation of `$v` in the `xsl:sort` XSLT instruction by `XP`. Furthermore, nested variables in `XP` must be already defined before the `xsl:for-each` XSLT instruction or, again, must be defined by an XPath expression such that the nested variables can be replaced in `XP`. In all other cases, the `order by` clause cannot be translated into equivalent XSLT instructions.

2.4 The Transforming XML Nodes to Element Nodes Approach

Whenever XML nodes, which are not element nodes, must be stored as intermediate results, a preprocessing step of the original XML document is needed to transform these XML nodes into element nodes as only element nodes can be stored in XSLT variables of the translated XSLT stylesheet. We use a namespace `t` in order to identify element nodes, which are transformed from not element nodes. Tests on XML nodes, which are not element nodes, are translated into tests on the corresponding element nodes (see Figure 1). As the result of the translated XSLT stylesheet contains copied element nodes, which are not element nodes of the original document, a postprocessing step must be applied to the result of the XSLT stylesheet, which then transforms these element nodes back to the corresponding XML nodes.

```
site/people/person/@name
```

is translated into

```
site/people/person/t:name
```

Fig. 1. Translating tests on attribute nodes

2.5 The Node Identifier Insertion Approach

In the following, we summarize the work-around presented in [6], which ensures that the identities of XML nodes in the translated XSLT stylesheet are considered in the same way as the identities of XML nodes in the original XQuery expression.

Whenever the `is` operator occurs in the XQuery expression, it is necessary to preprocess the source-document in order to add a new attribute `t:id` containing an unambiguous identifier to every XML element and postprocess the result of the XSLT stylesheet in order to remove the attribute `t:id`. Then the `is` operator can be translated into the `=` operator evaluated on the attribute `t:id`.

When elements are created as intermediate results, the translated XSLT stylesheet does not provide a mechism to set the `t:id` attributes of these elements. Using the `is` operator would work in these cases (see Figure 2). In order to consider both, the case that we have to consider the identity of XML nodes of the input XML document and of intermediate results, we will translate the `is` operator into two operations concatenated with the `or` operator (see Figure 3). One operation compares the `t:id` attributes the result of which is false in the case that there are no `t:id` attributes. The other operation uses the `t:is` operator the result of which is false if two copied XML nodes are compared.

The result of

```
let $a:= <z/>
return $a is $a
```

is "true", but the result of the wrong translation

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version="2.0">
<xsl:template match="/">
        <xsl:variable name='let0a'>
                <xsl:element name='z'/>
        </xsl:variable>
        <xsl:copy-of select="$let0a/@t:id = $let0a/@t:id"/>
</xsl:template>
</xsl:stylesheet>
```

is "false".

Fig. 2. Problems when translating the `is` operator in the case that elements are created as intermediate results

```
. is /site[last()]
```

is translated into

```
(./@t:id = /site[last()]/@t:id) or (. is /site[last()])
```

Fig. 3. Translating the `is` operator

2.6 Optimization

Our proposed translation algorithm checks

- whether non-element nodes must be stored as intermediate results and, only then, applies the transforming XML nodes to element nodes approach discussed in Section 2.4, and, otherwise, optimizes by avoiding the processing of this approach.

– whether the `is` operator is used and, only then, applies the node identifier insertion approach discussed in Section 2.5, and, otherwise, optimizes by avoiding the processing of this approach.

Furthermore, if necessary, both the preprocessing steps and postprocessing steps presented in the transforming XML nodes to element nodes approach and in the node identifier insertion approach can be applied in one step.

2.7 Handling Intermediate Results and Function Calls

XQuery supports closure by allowing nesting XQuery expressions with full generality. Due to the lack of closure in XSLT, query results must be stored in XSLT variables. The results can then be referenced by the names of the variables (see Figure 4).

```
for $i in doc("auction.xml")/site/closed_auctions/closed_auction
        where  $i/price/text() >= 40
        return $i/price
```

is translated into

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0"
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
    <xsl:variable name='rootVar1'>
        <xsl:copy-of select='document("auction.xml")'/>
    </xsl:variable>
    <xsl:template match="/">
        <xsl:variable name="for0_aux">
            <xsl:copy-of select='$rootVar1/site/closed_auctions/closed_auction'/>
        </xsl:variable>
        <xsl:for-each select="$for0_aux/*">
            <xsl:variable name="for0i" select="."/>
            <xsl:if test='$for0i/price/text()>=40'>
                <xsl:copy-of select="$for0i/price"/>
            </xsl:if>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Fig. 4. Translating a query with intermediate results

While translating a function, we store the function name, the names of its parameters and their order in a global data structure. Whenever we translate a function call, we access this global data structure in order to retrieve the necessary information of the names and the order of the parameters. Then the problem of parameter binding can be solved by mapping the names in the order of their appearance in the function call to the corresponding `xsl:param` tags (see Figure 5).

```
declare function local:mult($y, $x){ $y * $x };
local:mult(10, 10)
```

is translated into

```
<xsl:template name='mult'>
    <xsl:param name='y'/>
    <xsl:param name='x'/>
    <xsl:copy-of select='$y*$x'/>
</xsl:template>
<xsl:template match="/">
    <xsl:call-template name='mult'>
        <xsl:with-param name='y' select='10'/>
        <xsl:with-param name='x' select='10'/>
    </xsl:call-template>
</xsl:template>
```

Fig. 5. Translating a function

3 Translating XQuery Expressions into XSLT Stylesheets

In this section, we describe the algorithm to translate XQuery expressions into XSLT stylesheets.

3.1 Translation of an XQuery Expression

The translation from an XQuery expression into an XSLT stylesheet is done in two phases. In phase one, we parse the XQuery expression in order to generate the abstract syntax tree of the XQuery expression. For an example, see the XQuery expression in Figure 4 and its abstract syntax tree in Figure 6. In phase two, we evaluate the attribute grammar, which we do not present here due to space limitations. After evaluating the attribute grammar, a DOM [4] representation of the translated XSLT stylesheet is stored in the attribute `MainModul.docFrag` (see Figure 6). Figure 6 presents the evaluation of attributes for every node in the abstract syntax tree of the XQuery expression in Figure 4. Figure 4 presents also the final results of the translation process.

3.2 Processing of the Translated XSLT Stylesheet

See Figure 7 for an example of the entire translation process (step 1) and transformation process, which consists of the preprocessing step of the input XML document (step 2), the execution of the translated XSLT stylesheet (step 3) and the postprocessing step (step 4) of the results of the XSLT stylesheet.

If we can optimize according to what we have discussed in Section 2.6, then we will avoid the preprocessing step (step 2) and the postprocessing step (step 4).

4 Performance Analysis

This section describes the experiments that we have conducted to compare the execution time of translated XQuery expressions (i.e. XSLT stylesheets) in-

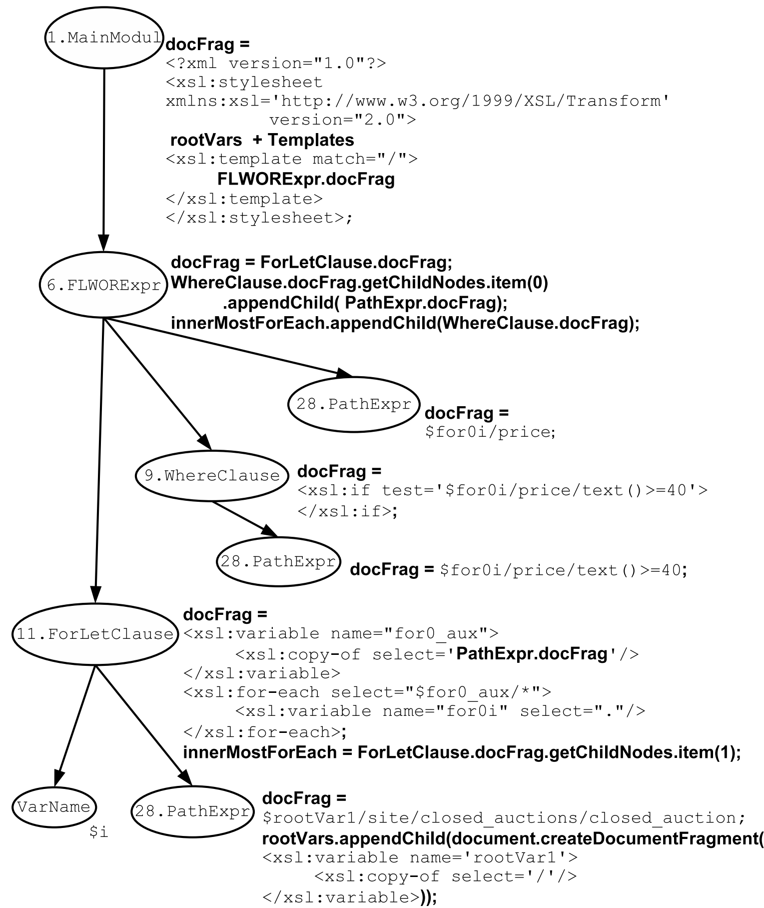

Fig. 6. The abstract syntax tree including computed attributes showing the translation of the XQuery expression in Figure 4

cluding the translation time with the time for executing the original XQuery expression.

4.1 Experimental Environment

We have used the XMark benchmark [10] for all our experiments. This benchmark consists of 20 XQuery queries and an XML data generator. This generator generates XML documents, the size of which can be scaled, containing auction data. The XMark developers chose the 20 XQuery queries that cover many aspects of XQuery. Furthermore, XMark is one of the most used benchmarks for XQuery in research. We have used documents of size 0.317 MB, 0.558 MB, 1.2 MB, 1.7 MB, 2.3 MB, 2.8 MB, 5.5 MB and 12 MB for the experiments.

We have used three different query evaluators: Saxon [5], Xalan [2] and Qexo [9]. Saxon has the capability to evaluate both XQuery expressions and XSLT

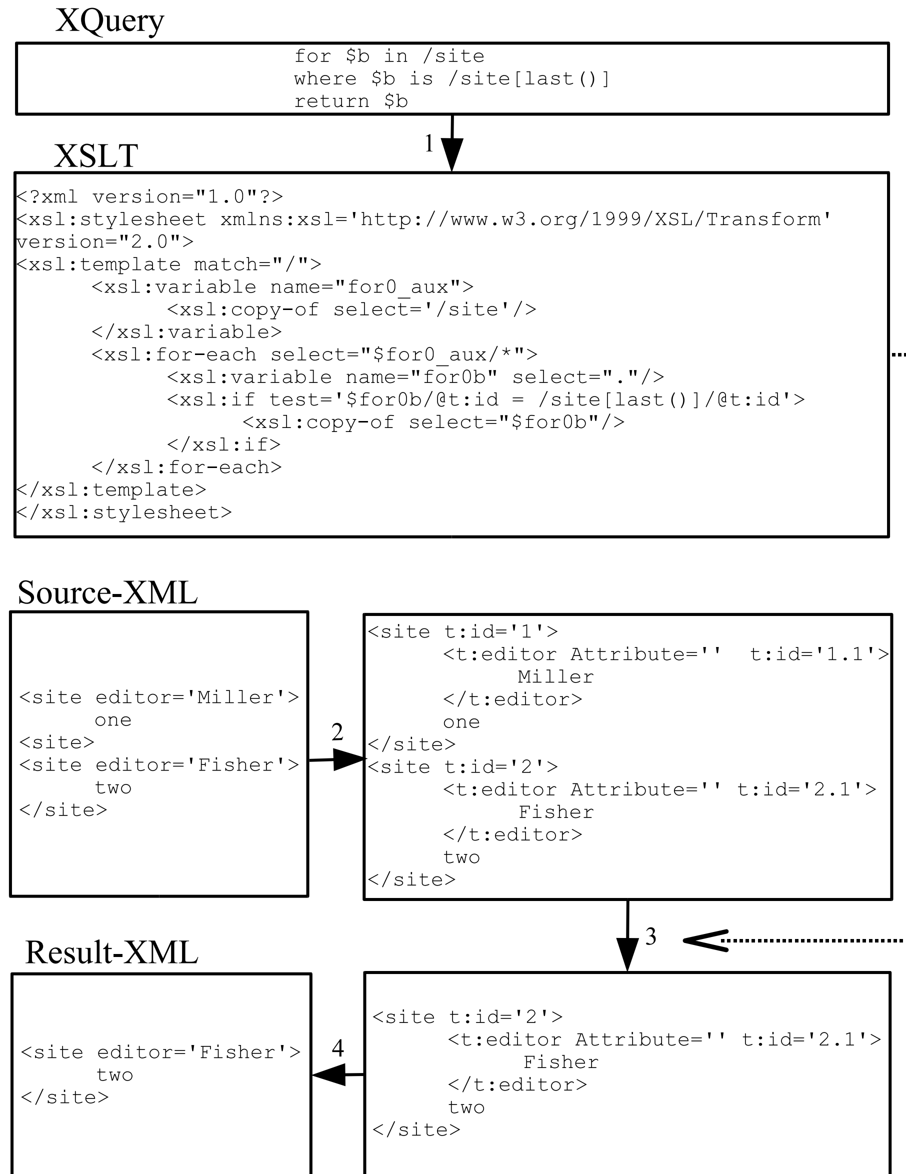

Fig. 7. The transformation process

stylesheets. Whereas Xalan is an XSLT evaluator, which is integrated in the Sun Java Development Kit, Qexo is an XQuery evaluator. Qexo is one of the few XQuery evaluators developed in Java, which implements most language constructs of the current XQuery specifications.

We present the average execution times of 20 executions for every XMark query in combination with every query evaluator.

We have run the experiments on an AMD Athlon 2 Gigahertz with 1 Gigabytes main memory, where 800 Megabytes are assigned to the Java virtual machine. The system runs a Linux kernel 2.6.4 and Java version 1.4.2.

4.2 Analysis of Experimental Results

We present the average execution times of twenty experiments of all 20 XMark queries and their translated XSLT stylesheets for a fixed document file size of 5.5 Megabytes in Figure 14. For the XSLT stylesheets in Figure 14 we have used optimized evaluation (i.e. without the pre- and postprocessing step) except for the unoptimized XSLT stylesheet of query 10, which cannot be optimized because query 10 uses the *is* operator. Saxon processes the XQuery queries 22 % faster on average compared to evaluateing the translated XSLT stylesheets. When we only consider queries without joins, i.e. all queries except the queries 8, 9 and 10, Saxon evaluates the XQuery queries 12 % faster on average. Figure 14 shows that Saxon evaluates the translated XSLT stylesheets of the queries 8, 9 and 10 up to 132 times slower compared to evaluating the original XQuery queries, which shows that Saxon does not optimize the execution of the translated XSLT stylesheets of the queries 8, 9 and 10 with joins. The execution of the translated XSLT stylesheets of the Xalan XSLT processor is 0.8 % faster on average compared to the execution of the Qexo XQuery evaluator of the XMark queries. Note that Xalan and Qexo can not evaluate all queries because they do not implement all used XPath functions. Furthermore, the evaluation time of the Qexo XQuery evaluator is large when evaluating queries containing joins. In fact, Qexo evaluates query 9 (containing two joins) over 7000 times slower than query 1 (no join). The translation needs linear time in the size of the input XQuery query, which is under 7 msec in all cases and can be neglected.

Optimized processing is on average 13 % faster than processing with the preprocessing step and postprocessing step. Only the XMark query 10 cannot be optimized (because it uses the *is* operator), such that the preprocessing step and the postprocessing step must be performed.

Furthermore, we present the average results of twenty experiments of six queries (XMark query 1, 2, 3, 8, 9 and 18) where we vary the document file sizes: Figure 8 shows the execution times of query 1, where the Saxon XQuery evaluator is 18 % faster on average compared to the Saxon XSLT processor with optimization (about 6 sec in the slowest case) and 33 % faster than Saxon XSLT without optimization. Using Xalan XSLT (optimized) is faster than Saxon XQuery for document sizes less than 10 MB, the Xalan XSLT processor is 32 % faster on average compared to Saxon XQuery. The Qexo XQuery evaluator is again 58 % faster than Xalan.

We have retrieved similar results for query 2 (see Figure 9), query 3 (see Figure 10) and query 18 (see Figure 13) compared to query 1. Note that Qexo cannot evaluate query 3 and query 18.

XMark Query 8 contains one join and XMark query 9 contains two joins. The execution times of query 8 (see Figure 11) and the execution times of query 9 (see Figure 12) show that XSLT processors evaluate the translated XSLT stylesheets

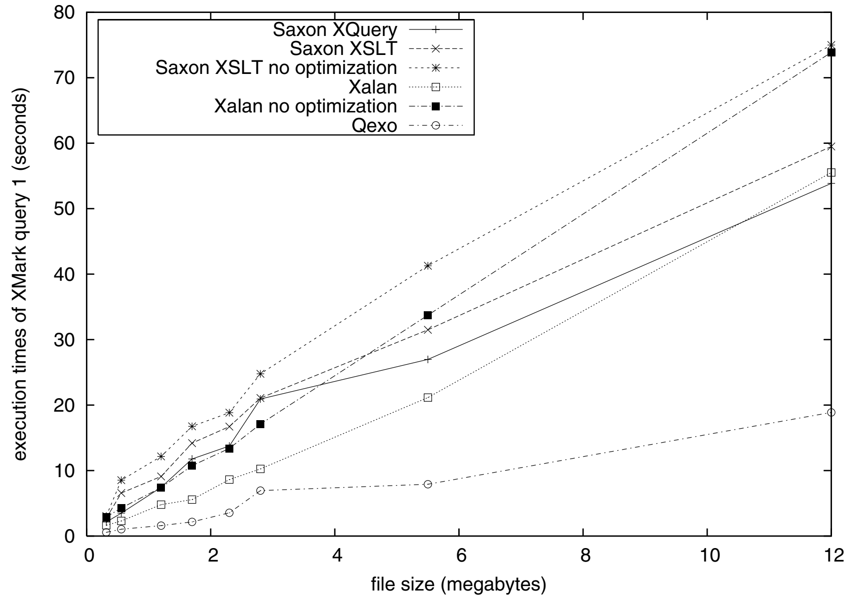

Fig. 8. Execution times (y-axis) of XMark Query 1 and of its translated XSLT stylesheet depending on the file size (x-axis)

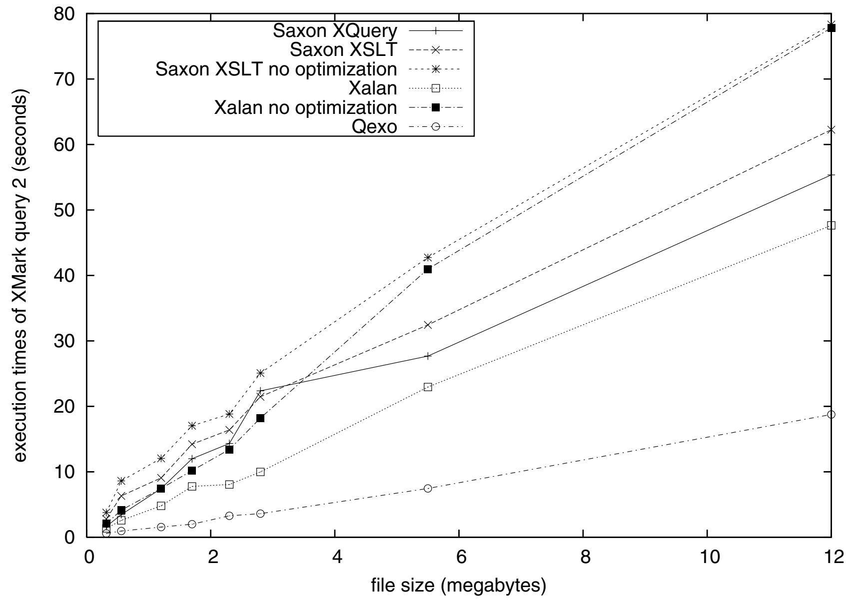

Fig. 9. Execution times (y-axis) of XMark Query 2 and of its translated XSLT stylesheet depending on the file size (x-axis)

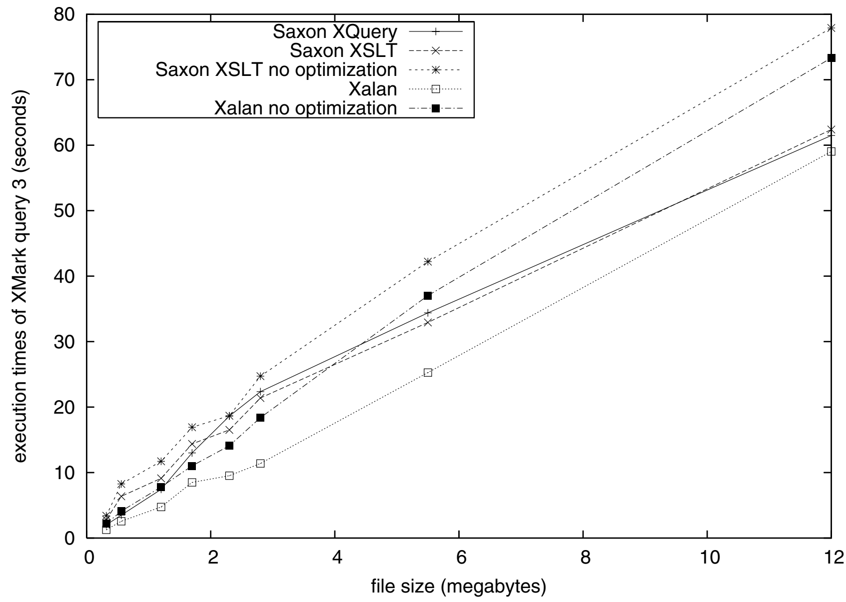

Fig. 10. Execution times (y-axis) of XMark Query 3 and of its translated XSLT stylesheet depending on the file size (x-axis)

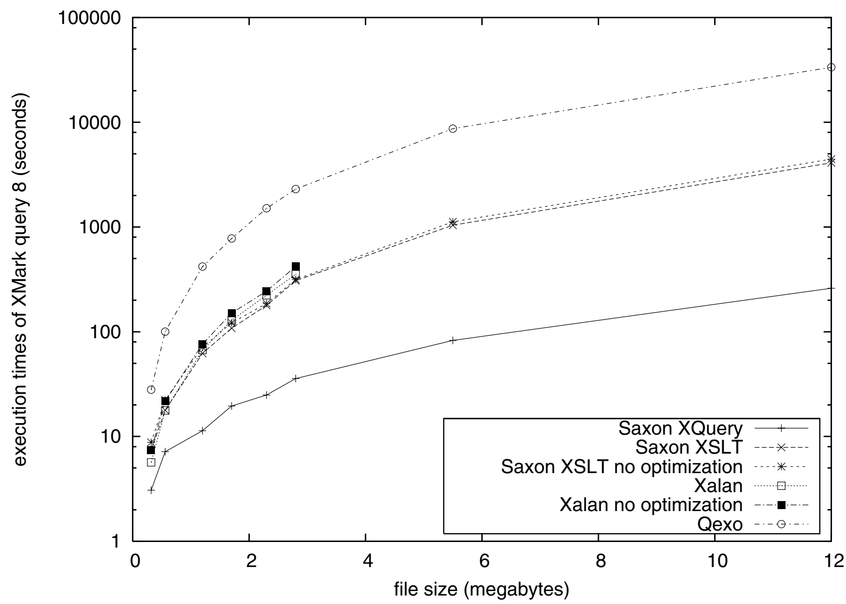

Fig. 11. Execution times (y-axis) of XMark Query 8 and of its translated XSLT stylesheet depending on the file size (x-axis)

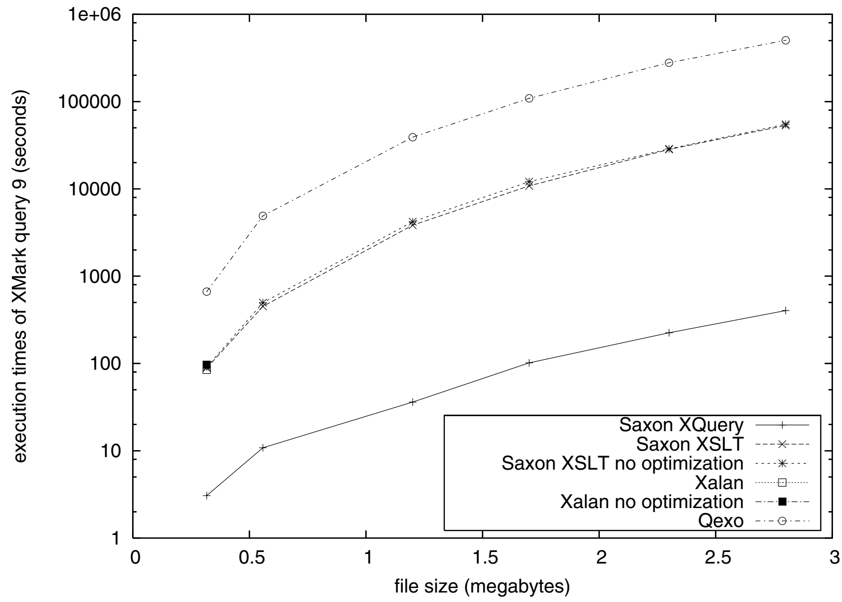

Fig. 12. Execution times (y-axis) of XMark Query 9 and of its translated XSLT stylesheet depending on the file size (x-axis)

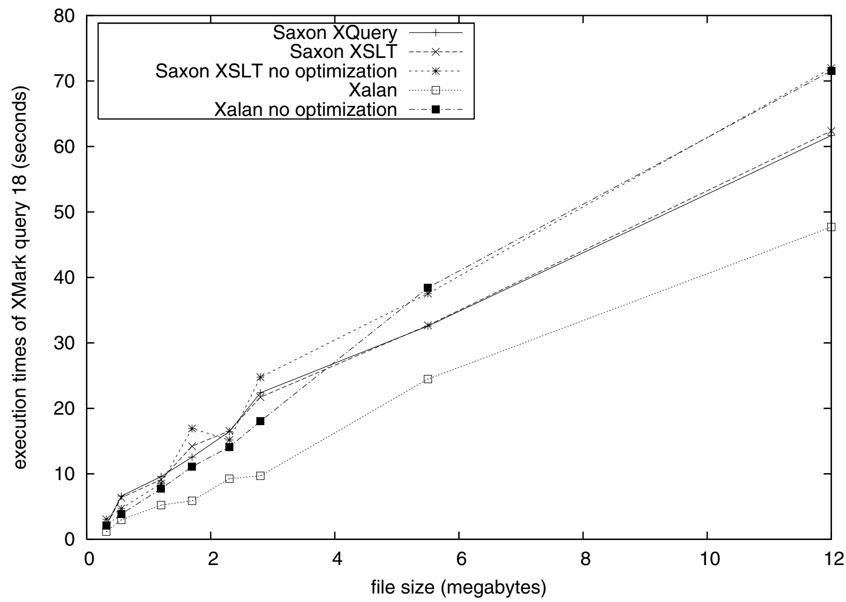

Fig. 13. Execution times (y-axis) of XMark Query 18 and of its translated XSLT stylesheet depending on the file size (x-axis)

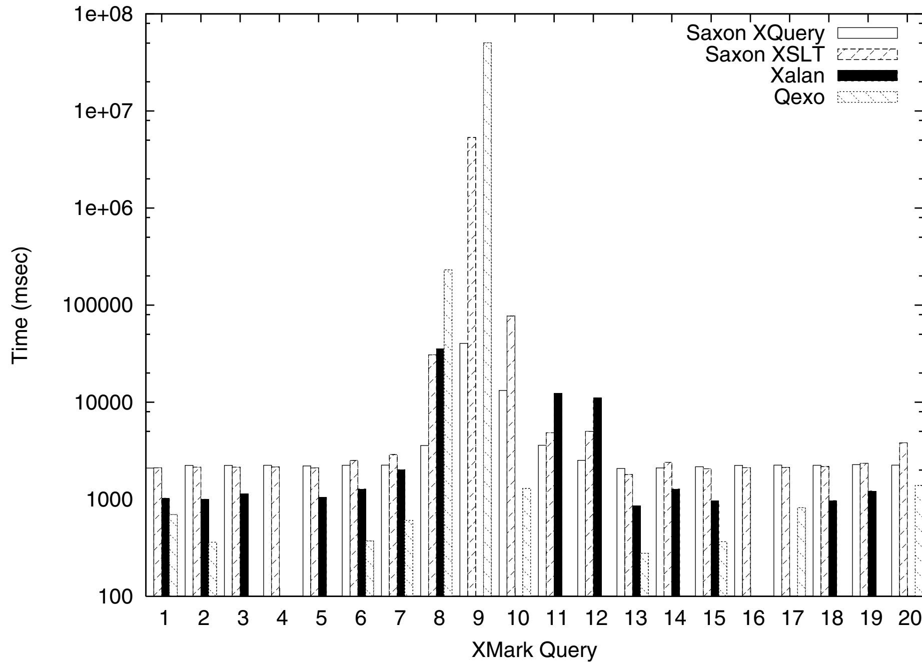

Fig. 14. The execution time of all XMark queries for a file size of 5.5 Megabytes

of queries containing joins much slower compared to the execution of XQuery evaluators of the XQuery queries.

The Saxon XQuery evaluator processes both queries much faster, 80 % for query 8 and 98 % for query 9, compared to the execution of the optimized translated XSLT stylesheets of the Saxon XSLT processor. Contrary, the evaluation of the translated queries of the Saxon XSLT processor is 85 % faster for query 8 and 90 % faster for query 9 compared to the Qexo XQuery evaluator.

5 Summary and Conclusions

We have presented an approach for translating XQuery expressions into XSLT stylesheets. We described the algorithm for the translation process in terms of an attribute grammar, which we do not present here due to space limitations. In general, there must be a preprocessing step for the original XML document before executing the translated XSLT stylesheet and a postprocessing step for the result of the translated XSLT stylesheet.

The experiments considering the XMark queries showed that executing the translated XSLT stylesheet is 12 % slower than native XQuery processing (except queries containing joins). We show that in most cases, but at least for all XMark queries except one XMark query, we can optimize and avoid the preprocessing

step and the postprocessing step. Optimized processing is on average 13 % faster than processing with the preprocessing step and postprocessing step. Therefore, we have achieved the goal to make XQuery practically useable for the broad field of XSLT tools.

References

1. apache.org. Cocoon (2004) http://cocoon.apache.org
2. apache.org. Xalan (2004) http://xml.apache.org/xalan-j
3. Fernandez, M., Robie, J. (Eds): "XQuery 1.0 and XPath 2.0 Data Model". W3C Working Draft, June (2001) http://www.w3.org/TR/2001/WD-query-datamodel/
4. Hors, A. L., Hegaret, P. L., Nicol, G., Robie, J., Champion, M., Byrne, S. (Eds): "Document Object Model (DOM) Level 2 Core Specification Version 1.0". W3C Recommendation, Nov. (2000) http://www.w3.org/TR/DOM-Level-2-Core/
5. Kay, M. H.: Saxon (2004) http://saxon.sourceforge.net
6. Lechner, S., Preuner, G., Schrefl, M.: Translating XQueryinto XSLT. In Revised Papers from the HUMACS, DASWIS, ECOMO, and DAMA on ER 2001 Workshops, Springer-Verlag (2002) 239–252
7. Lenz, E.: XQuery: Reinventing the Wheel? (2004) http://www.xmlportfolio.com/xquery.html
8. Microsoft. Biztalk (2004) http://www.biztalk.org/
9. qexo.org. Qexo (2004) http://www.gnu.org/software/qexo
10. Schmidt, A., Waas, F., Manolescu, I., Kersten, M., Carey, M. J., Busse, B.: XMark: A benchmark for XML data management. In Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002), Hong Kong, China, July 02 (2002)
11. W3C. XSL Transformations (XSLT) (2003) http://www.w3.org/TR/xslt
12. W3C. XML Query (2004) http://www.w3.org/XML/Query