PERFORMANCE EVALUATION FOR XSLT PROCESSING

by

Torsten Bittner

University of Rostock

2004

Approved by: Dr. Meike Klettke

Date: 07/15/2004

University of Rostock

Abstract

PERFORMANCE EVALUATION FOR XSLT PROCESSING

by Torsten Bittner

The increasing popularity of XSLT for transforming XML data brings up the question of the performance of the transformations. This paper gives an introduction into the topic of XSLT performance regarding to the relationship of the three components XSLT processor, stylesheet and input data and their major influence on the transformation time. These three factors are described and their impact is analyzed. Different approaches to measure XSLT performance and the testing environment are discussed.

Various XSLT processors are introduced and their functionalities are compared to each other. Their performance is measured using various test simulation models.

The performance of different XSLT stylesheets is evaluated. The data mapping tool prototype Clio is used to automatically generate these XSLT stylesheets.

Finally some ideas that help XSLT developers to improve the speed of XSL transformations are discussed.

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

Since 1999 the popularity of XSLT increased continuously as it is becoming a common means for XML data transformations. XSLT is a powerful and flexible language that often offers developers more than one way to solve a problem. Since XSLT is a new language the performance has not been fully analyzed and explored to provide the best perspectives into the usage of the technology.

As a declarative programming language, XSLT gives developers the flexibility to write code to describe what is wanted but does not answer which is the best way to perform the same transformation in relevance to speed, time, and process consumption.

For practical XSLT programming the execution time of an XSL transformation is often a crucial factor. Thus concrete problems are a topic of interest in XSLT newsgroups [mulb]. There XSLT developers exchange ideas of how performance can be improved.

The available reference on this topic is poorly documented. Although XSLT performance is covered in some of the available XSLT literature, these parts are usually pretty short and only give limited hints on how to improve XSLT performance. The "Guide to XSLT Performance" is missing. Points of interests are: Is there a "best" way to approach XSLT programming? How big is the performance difference of different XSLT stylesheets that produce the same result? What is "good" XSLT? How can performance be measured? How can tests be executed? How can results be compared? Are there any general hints that can be applied for most XSLT problems?

This paper covers some of these points and gives an introduction into the area of XSLT performance. It is for people who already know XSLT. Good sources for general information about XSLT programming are *Beginning XSLT* [Ten00] and *XSLT & XPATH – A Guide to XML Transformations* [GR02].

Figure 1-1 depicts the architecture of an XSL transformation. It shows that there are three components that impact the transformation and thus the transformation performance:

1. XML input data
2. XSLT stylesheet
3. XSLT processor

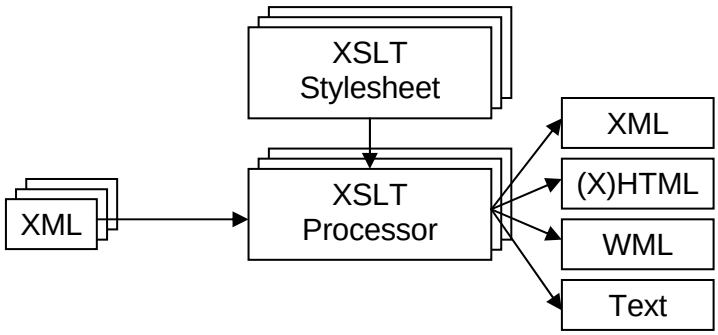

Figure 1-1 Architecture of an XSL Transformation

XML input data is transformed by the XSLT processor. The XSLT stylesheet contains rules that describe this transformation. The output data can have different formats e.g. XML, (X)HTML, WML and Text. The question how the three components affect the performance of the XSL transformation is discussed in this paper.

Chapter 2 gives an overview of currently available XSLT processors and compares their functionalities.

An environment to measure XSLT performance is characterized in Chapter 3. Different approaches to generate XML data are introduced. The software that is used for measuring XSLT performance is described as well as the hardware platform. In addition the parameters that affect XSLT performance and that can be tested and varied are compared to each other.

The transformation performance of some currently available XSLT processors is compared in Chapter 4. Two different benchmarks are used to find out which XSLT engine provides the best performance.

In order to ease the creation of XSLT stylesheets IBM currently develops the data mapping tool *Clio*. Clio is used to generate XSLT stylesheets for common XSL transformation operations. Chapter 5 introduces Clio and the performance of Clio-generated XSLT scripts is evaluated.

Using the XSLT performance testing environment different ways to speed up XSL transformations are presented in Chapter 6. Several approaches to execute a transformation are compared to each other. Some hints about general advantages of each approach are provided.

The paper finishes with a conclusion and an outlook on further investigation of the area of XSLT performance in Chapter 7.

2 XSLT processors

In 1999 the W3C released the XSLT 1.0 recommendation. Since then several companies started to develop XSLT processors. These companies like IBM, Microsoft, Oracle or Altova tried to put their feet into the future market for XML processing tools. Other products of these companies already supported XML data. XSLT extends the possibilities to take use of this data. Thus the integration of XSLT engines into data management tools becomes a necessity. However, the stand-alone XSLT engines are entirely available for free usage.

Apart from companies private people started to implement XSLT processors as well. Michael Kay, author of "XSLT Programmers Reference" [Kay00], and James Clark as a member of the W3C committee are the most popular examples. Michael Kay developed *Saxon* and James Clark the XSLT processor *XT*. In addition there is another set of XSLT processors developed by open source communities.

The following chapter provides an overview of the most popular XSLT processors. They are described and compared by their functionalities and characteristics. The main criterion is the implementation language. The supported platforms are usually implied by this language.

Another characteristic is the support of EXSLT functions. During the usage of XSLT processors it turned out that the capabilities of XSLT 1.0 functions were limited – especially

when handling dates, time, mathematic functions, string functions and regular expressions. Those functions were often needed by XSLT programmers. Hence the developers of XSLT processors started to implement a proprietary support of enhanced functions. However this limited the XSLT code to be executed with only one processor and the great flexibility of XSLT was lost. Thus the implementers of XSLT processors founded the EXSLT community [exsl] to provide a standardized set of functions that is used by all processors to keep the XSLT code portable. The initiative's goal is also to include EXSLT functions in future XSLT standards.

The up-to-dateness of XSLT processors is very important, too. In the past a couple of projects were not continued while at the same time new XSLT processors came into existence. Some of them are constantly improved and enhanced. Until today the integration of XSLT 2.0 functionalities is not completed. So the major focus of most XSLT implementers is on the features rather than the performance of the processors.

2.1 Java Processors

The idea of XML is to deliver a platform independent format to exchange data. That perfectly matches the idea of Java as a platform independent programming language. Hence a couple of XSLT processors are implemented in Java.

The Java API for XML Processing (JAXP) provides a standard interface for parsing and processing XML data. The XSLT processors implement these functions according to the JAXP interface which is available in different versions. Developers who build their application on top of the JAXP interface can easily exchange the XSLT processing engine.

The following section introduces the most popular Java XSLT processors in alphabetical order.

2.1.1 jd.xslt

The Java processor jd.xslt is an open source XSLT engine. It has been developed mostly by Johannes Döbler. The processor implements the XSLT 1.1 Working Draft. Currently the work on the processor is not continued.

Developer	Johannes Döbler	Info	[jdxs]
Platform	Java	Code	Java
License	Mozilla Public Licence	XSLT	1.1
Version	1.5.5	JAXP	n/a
Version date	May 2003	EXSLT	yes

2.1.2 Oracle XDK

The Oracle XSLT processor is part of the Oracle XML Developer's Kit (XDK). This kit contains different tools for XML processing like XML Parsers (SAX and DOM), XML

schema processors, XML class generator, XML SQL utility, XSQL servlet, XML pipeline processor and the TransX Utility. Oracle offers full support for the package that is closely interacting with the Oracle database management system. Due to these close interrelations Oracle XSLT is especially useful for XML developers who are building their application on top of Oracle products.

The XDK also includes a C++ implementation of the XSLT processor. However, this one is only compliant to the XSLT 1.0 specifications.

Developer	Oracle	Info	[oxdk]
Platform	Windows / Unix	Code	Java / C++
License	Oracle	XSLT	2.0 / 1.0
Version	10.1.0.2.0	JAXP	1.2
Version date	March 2004	EXSLT	n/a

2.1.3 Saxon

Saxon is developed by Michael Kay, who is an expert in the XSLT community, member of the W3C and author of XSLT literature [Kay00]. The name Saxon results from the original architecture of the processor. It is built 'on' top of the 'SAX' parser.

The XSLT processor is available in three different versions. Saxon 6.5.3 is a stable release from August 2003 that fully implements XSLT 1.0. Additionally, it offers some XSLT 1.1 features and a large set of the EXSLT extension library. Some of the extended functions started as proprietary Saxon features and were adopted by other XSLT processors later on.

Instant Saxon 6.5.3 provides the same functionality but comes without the source code and sample applications. Installation and usage are easier and it is limited to Windows systems because it is using the Microsoft Java Virtual Machine (JVM). It can be used for simple applications. However for professional usage the full Saxon 6.5.3 processor is recommended because it performs better due to the usage of the Sun JVM.

The latest development work is done on the 7.x -series of Saxon. Additional features of the XSLT 2.0 standard are implemented step by step. Saxon 7.9.1 is not only an XSLT processor but also an XPath 2.0 and XQuery 1.0 processor. Further development and the complete implementation of the XSLT 2.0 standard are planned. However, in March 2004 Michael Kay founded the company Saxonica Limited which will release future versions with more functionality as a commercial product.

Developer	Michael Kay	Info	[saxn]
Platform	Java	Code	Java
License	Mozilla Public License 1.0	XSLT	1.0 + 1.1 / 1.0 + 2.0
Version	6.5.3 / 7.9.1	JAXP	1.1 / 1.2

Version date	August 2003 / November 2003	EXSLT	yes

2.1.4 Xalan-J

The Apache project Xalan-J came into existence as a donation of the former LotusXSL engine developed by IBM [lxsl]. Later Sun's XSLT Compiler became part of the open source project as well. The name Xalan is derived from an African music instrument [xalm].

The XSLT processor Xalan is closely interrelated to the XML parser Xerces. Both are well maintained and constantly improved. One peculiarity about Xalan is the availability of Document Table Models (DTM) in addition to the common Document Object Model (DOM). The DOM is memory consumptive. The DTM approach tries to save memory and promises better performance.

The XSLT Compiler enables the user to translate XSL stylesheets into so-called translets. The translet contains Java byte code and can be applied to XML documents for a transformation. The idea is also to improve the transformation performance. However, this aspect is not part of the transformations tested in this paper.

Developer	Apache Software Foundation	Info	[xalj]
Platform	Java	Code	Java
License	Apache License 2.0	XSLT	1.0
Version	2.6.0	JAXP	1.2
Version date	February 2004	EXSLT	yes

2.1.5 XT

James Clark, member of the W3C committee, developed XT with the main goal to create a fast XSLT processor. The project is now maintained by Bill Lindsey. Due to the concentration onto high performance not every function of the XSLT 1.0 standard is implemented. However, XT offers some extension functions and can be the right choice if performance is more important than full conformance to the standard.

Developer	James Clark / Bill Lindsey	Info	[xt]
Platform	Java	Code	Java
License	Open Source	XSLT	partly 1.0
Version	20020426a	JAXP	n/a
Version date	April 2002	EXSLT	partly

2.2 C/C++ Processors

Java is known for platform independence but also for slower runtime performance than C/C++ implementations. Thus there are a couple of C/C++ XSLT processors available. They try to outperform their Java competitors.

2.2.1 Altova XSLT Engine

Altova has released the XSLT engine that is used in their products (e.g. XML Spy) as a stand-alone version. It is available for free download [alto]. It completely implements the XSLT 1.0 standard and is written in C++. The package only includes one executable file and the documentation. The usage is very easy on Windows systems.

Developer	Altova GmbH	Info	[alto]
Platform	Windows 98/ME/2000/XP/2003	Code	C++
License	Altova XSLT Engine developer license	XSLT	1.0
Version	XSLT Engine 2004 Release 3	JAXP	n/a
Version date	October 2003	EXSLT	n/a

2.2.2 FastXML

FastXML is developed by Helena Kupkova as a result of her master's thesis. With optimizations and efficient memory usage the XSLT processor has a good performance. However, it is not fully compliant to the XSLT 1.0 standard and is currently not further developed.

Developer	Helena Kupkova	Info	[fast]
Platform	Windows	Code	C++ / Assembler
License	n/a	XSLT	most of 1.0
Version	n/a	JAXP	n/a
Version date	April 2001	EXSLT	n/a

2.2.3 Libxslt

Libxslt is an XSLT library that is developed for the Gnome project. It is open source, written in C and includes the Libexslt library that implements some of the EXSLT extension functions.

Developer	Gnome Project	Info	[libx]

Platform	Windows /Linux / Solaris / MacOS	Code	C
License	MIT License	XSLT	1.0
Version	1.1.4	JAXP	n/a
Version date	March 2004	EXSLT	yes

2.2.4 Microsoft Msxml

Microsoft Core Services 4.0 is a package of XML tools. The Dynamic Link Library (DLL) that includes the processor can be used to develop XML applications. It can also be invoked from the command line using an additional executable file.

Developer	Microsoft	Info	[msxm]
Platform	Windows	Code	C++
License	Microsoft	XSLT	1.0
Version	4.0 SP2	JAXP	n/a
Version date	April 2003	EXSLT	n/a

2.2.5 Sablotron

Sablotron is an open source project of the Ginger Alliance. It is implemented in C++ and depends on the XML parser Expat which is developed by James Clark. Developers can use the processor in C++ applications and in other languages e.g. Perl, PHP, Object Pascal and Ada. This is possible because of wrappers that were developed for these languages.

Developer	Ginger Alliance	Info	[sabl]
Platform	Linux / Windows 98/ME/2000/XP / Solaris / HP-UX / Irix / FreeBSD / OpenBSD	Code	C++
License	Mozilla Public License 1.1	XSLT	1.0
Version	1.0.1-2	JAXP	n/a
Version date	November 2003	EXSLT	n/a

2.2.6 Xalan-C++

Xalan-C++ is an alternative for the Xalan-J engine that is also offered from the Apache Software Foundation. One goal of the C++ version is to improve performance and memory usage. The support of EXSLT extensions is currently in Beta stadium.

Developer	Apache Software Foundation	Info	[xalc]
Platform	Windows / Linux (RedHat, SuSE) / AIX / HP-UX / Solaris	Code	C++
License	Apache License 2.0	XSLT	1.0
Version	1.7	JAXP	n/a
Version date	January 2004	EXSLT	yes (beta)

2.3 Other Processors

A major field for XSLT is the transformation of XML into HTML on web servers. Perl and Python are programming languages that are often used on web servers. Hence the XSLT processor Sablotron provides wrappers for these languages. In addition there are projects which implement XSLT processors directly in these languages.

2.3.1 4Suite 4XSLT

4Suite is an XML development toolkit that implements technologies like XSLT, DOM, RDF, XPath and XPointer entirely in Python.

Developer	Fourthought	Info	[4xsl]
Platform	Windows 98/ME/2000/XP / Linux / Solaris / MacOS /FreeBSD	Code	Python
License	4Suite License 1.1	XSLT	1.0
Version	1.0a3	JAXP	n/a
Version date	July 2003	EXSLT	yes

2.3.2 XML::XSLT

Many web servers, especially those of web space providers for private people, do not support Java servlet technologies. Perl is supported much more often. Hence the Perl implementation of XML::XSLT targets people who want to execute XSL transformations on their private homepage without Java support.

Developer	Geert Josten, Egon Willighagen	Info	[xmlt]
Platform	Windows 98/ME/2000/XP / Linux / Solaris	Code	Perl
Licence	Open Source	XSLT	most of 1.0

Version	1.25	JAXP	n/a
Version date	February 2004	EXSLT	n/a

3 XSLT Performance Test Environment

In order to run performance tests for XSL transformations several components are required:

1. Input data
2. XSLT stylesheet
3. XSLT processor
4. Parameters that affect performance and are modified during the tests
5. Parameters that are measured during the tests
6. Software to measure performance (benchmarks)
7. Hardware platform that is fixed for all tests

The following chapter introduces the test environment that is used for the XSLT performance tests in this paper. Most of the provided information is general and is helpful for any type of XSLT performance tests.

All components are described in the following sections. However the creation of XSLT stylesheets (Chapter 5 and 6) and the introduction of XSLT processors (Chapter 2) are discussed in separate sections in more detail.

3.1 Generation of XML data

Input data is required for every XSL transformation. The data has to be well-formed XML. The presence of a matching XML schema for this data is helpful, because applications often use the schema to validate the processed XML data. Often the schema is required for the work with XML data. An example is the data mapping tool Clio that imports XML schema files (Chapter 5) as the base of every document mapping.

The need for XML schemas and valid XML data brings up the question: "Why not just using real data and see how good the transformation performs?" In many cases it is definitely the best idea to work with real data because this data exactly represents the scenario that the transformation is going to be used for. However, there are several problems with using real data. Firstly, there might not be enough real data available. Especially when creating new applications it is likely that there is hardly data existing to work with because it is only accumulated once the application is running. There are sources in the internet that offer XML data collections for download [shap] [trad]. However, this data is limited to certain schemas. So it does not match all possible requirements. Another downside of real data is that there is no control of the data structure and file size.

The best way to get input data for testing purposes is to create the data according to the needs of the application. There are different ways to create XML data. The following sections give an overview about the most common approaches.

3.1.1 General approaches

For simple purposes creating XML files can be done quite easily by writing an XML file according to a certain XML schema. This can be done either manually from the scratch or assisted by a software tool. Several programs offer this functionality e.g. Altova XML Spy and IBM WebSphere Studio Application Developer. Figure 3-1 shows how WSAD generates an XML file that is valid to given XML schema. The major problem with this way of creating XML files is the need for manually padding the files with data. The assistance of the tools is helpful but the creation of large XML files is still very cumbersome.

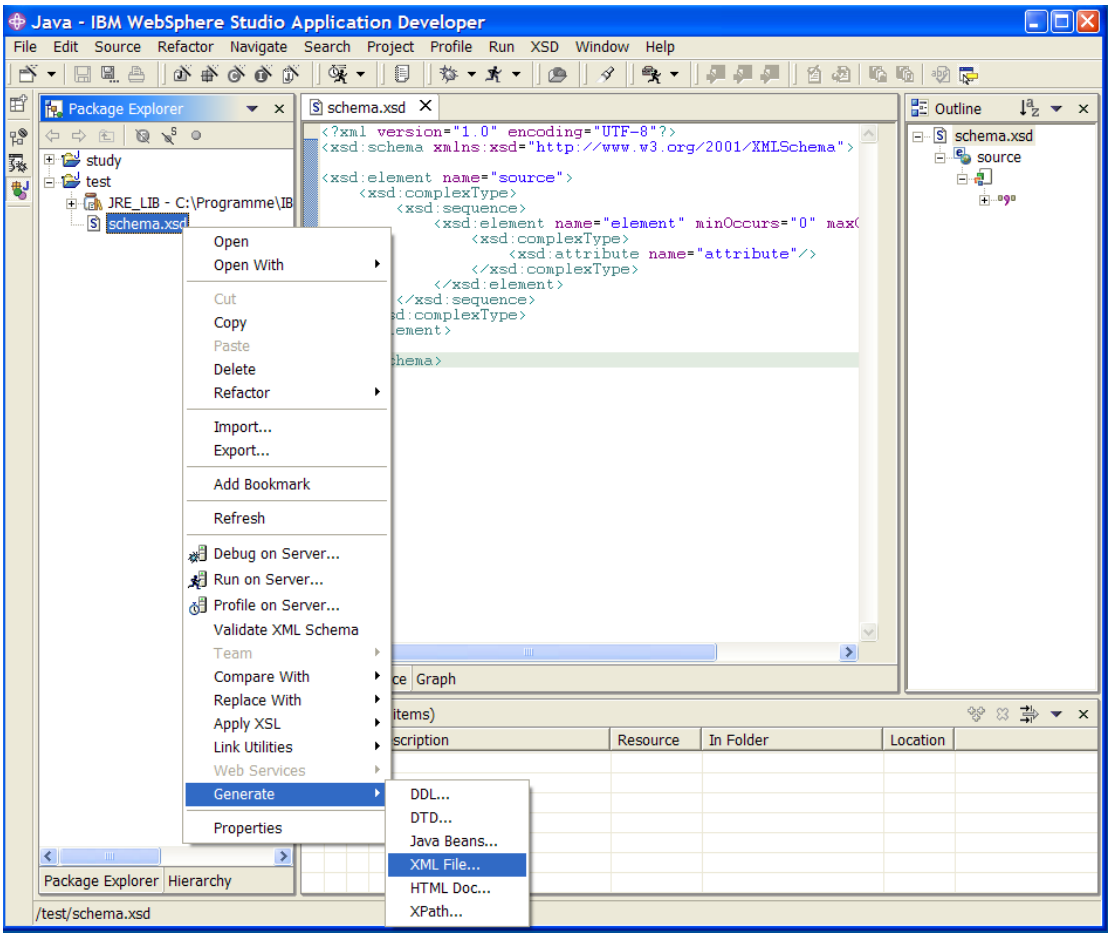

Figure 3-1 Generating an XML file using WSAD

A more powerful means to create XML files is to write a program or a script e.g. using Java or Perl. Java users benefit from the presence of XML APIs like DOM that ease the work with XML. Perl is particularly useful because of its powerful support of string functions and regular expressions. With the flexibility of loops and randomized functions it is much easier to create documents with different elements that have random content. The number of elements and the document size can be varied easily. Further down in this chapter it turns out that these are important factors which affect the performance of the transformation process. So it is important to have control of these factors when creating input data.

The downside of an application or a script is their inflexibility of being adapted to different XML schemas. Modifications of XML schemas result in continuous adjustments of the program code.

A much more intuitive way is to take a DTD or an XML schema and create valid sample data out of it. However, the realization of this functionality is not trivial. In different projects people tried to provide a solution for this problem.

3.1.2 Synthetically generation from DTD – IBM XML Generator

The XML Generator was developed by IBM to synthetically generate XML data based on a DTD. It is written in Java and available for free on the IBM Alphaworks website.

Given a well-formed DTD and a valid XML tree it creates random instances of valid XML data. Figure 3-2 shows the imported DTD, certain probabilities that can be defined by the user for the appearance of elements or attributes and the resulting XML document that is generated based on these parameters.

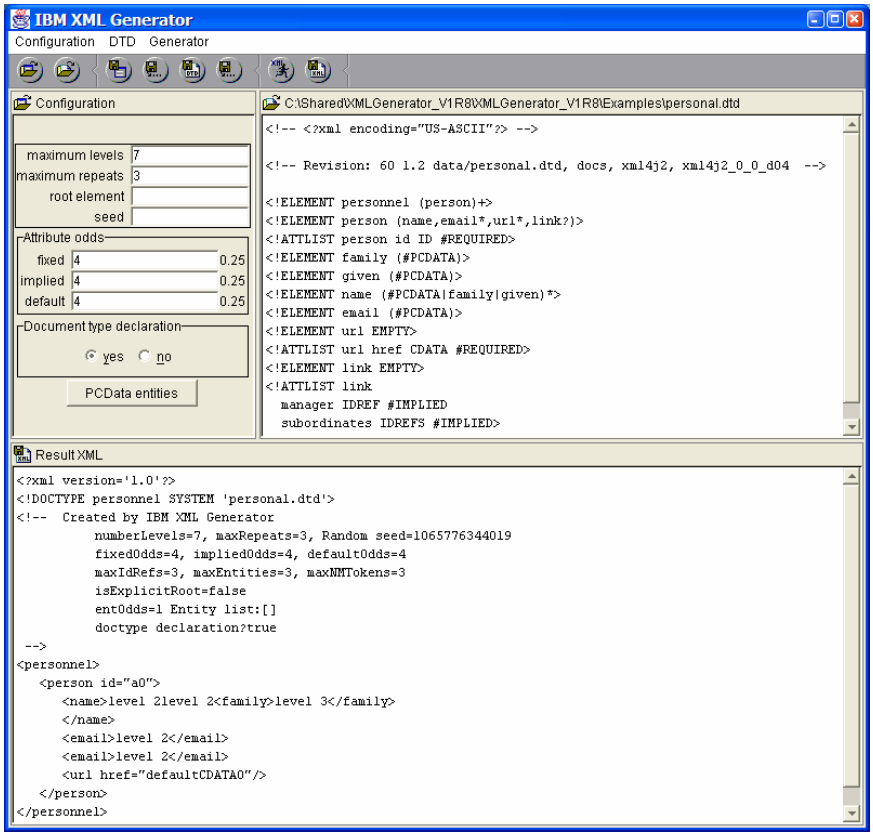

Figure 3-2 IBM XML Generator GUI

However, the XML Generator provides only limited control of the data generation process. For example, it is impossible to specify the exact number of levels for the generated XML documents; only the maximum level can be defined. Apart from that there is a limitation to uniform frequency distributions with no opportunity for generating skewed data. In addition the XML Generator can only process DTD files. It is not equipped to handle XML schemas that have more power to describe the structure of XML files and define different data types.

3.1.3 Synthetically generation from XML schema - ToXgene

The limitations of the XML Generator are eliminated in ToXgene. ToXgene is a Java based tool that is developed mainly by Denilson Barbosa at the University of Toronto [toxg]. It is part of the Toronto XML Server (ToX) project. The Toronto XML Server is a heterogeneous repository for XML data and metadata, which supports real and virtual XML documents [toxx].

ToXgene itself is a template-based generator for large, consistent collections of synthetic XML documents. It is declarative, powerful, very flexible and easy to use. It defines its own language to describe templates. This ToXgene Template Specification Language (TSL) is a subset of the XML schema notation that is enhanced with content-describing annotations. The language is used to describe the structure of an XML file as well as its content.

The consequence of the similarity of the TSL-file to an XML schema is that the user can basically take the XML schema and add the description of the content to get a TSL-file.

The following example illustrates this procedure. The XML schema *states.xsd* is the base:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="source">
  <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="state" minOccurs="0" maxOccurs="unbounded">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="state_name" type="xsd:integer"/>
     <xsd:element name="country_name" type="xsd:integer"/>
     <xsd:element name="continent_name" type="xsd:integer"/>
   </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Assuming that the names for states, countries and continents are integer values a manually written TSL-file *states.tsl* looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tox-template
          SYSTEM "http://www.cs.toronto.edu/tox/toxgene/ToXgene2.dtd">
<tox-template>
<tox-document copies="1" name="states" pad="no" starting-number="1">
 <element name="source">
  <complexType>
   <element name="state" minOccurs="1000" maxOccurs="1000">
    <complexType>
     <element name="state_name">
      <simpleType>
       <restriction base="integer">
        <tox-number minInclusive="0" maxInclusive="9"/>
       </restriction>
      </simpleType>
     </element>
```

```
      <element name="country_name">
       <simpleType>
        <restriction base="integer">
         <tox-number minInclusive="0" maxInclusive="9"/>
        </restriction>
       </simpleType>
      </element>
      <element name="continent_name">
       <simpleType>
        <restriction base="integer">
         <tox-number minInclusive="0" maxInclusive="9"/>
        </restriction>
       </simpleType>
      </element>
     </complexType>
    </element>
   </complexType>
  </element>
</tox-document>
</tox-template>
```

The ToXgene engine generates one file *states.xml* based on the descriptions in *states.tsl*. This XML file contains exactly 1000 `<state>` elements. Each `<state>` element has a `<continent_name>`, a `<country_name>` and a `<state_name>` element. These three elements have random integer values between 0 and 9. The file is valid to the schema *states.xsd*.

The example shows the similarity of the XML schema file and the TSL-file. The basic concept of ToXgene becomes obvious. However, ToXgene offers much more functionalities. It supports the generation of complex XML content like CDATA, element, attributes and mixed. CDATA values can be generated according to a type declaration. Various string, non-gibberish text, numeric and date types are supported. Document collections that share the same elements can be generated to enable consistent ID and IDREF connections. Real application data can be imported to be used as the base for randomized element or attribute content [toxg].

Using ToXgene it is possible to synthetically generate XML data, which can easily be adapted to unique requirements by varying the input parameters of the data generator template file. The structure of the data is well-understood and it is easy to generate complete collections of files with different sizes. All the user has to make sure is that the complexity of the synthetic data reflects the complexity of the real-life scenario. Due to these reasons ToXgene is used to generate the input data for the XSL transformations in this paper.

3.2 Parameters that affect XSLT performance

There are different parameters that can be varied when measuring XSLT performance. The following section introduces several aspects that are important for choosing the parameter for performance evaluation that matches the needs for a testing scenario best.

3.2.1 XSLT processor

The XSLT processor has a major impact on the performance of the transformation. Supported by an XML parser it applies the template rules of the stylesheet to the data and creates the output data.

XSLT processors use different XML parsers. Some processors use external parsers while others implement their own. For example Xalan-J uses Xerces and XT uses Expat. Msxml does not use an external parser. However, the speed of XML parsers is neglected in this paper. The open source project *XML Benchmark* [xmlb] provides details about XML parser performance. The combined speed of XML parsing and XSLT processing is covered in the benchmark comparison of XSLT processors in Chapter 4.

3.2.2 XSLT stylesheet

The complexity of the XSLT stylesheet strongly affects the transformation time. The usage of different XSLT code that creates the same output often results in big differences of the processing time. Chapter 5 and 6 illustrate the effect of the XSLT stylesheet in more detail.

3.2.3 XML input data

For many data processing benchmarks the input data is a crucial factor. Mostly it is the file size that determines the processing speed. So for XSL transformations one could vary the file size in order measure performance. However, there is one big downside to this approach.

XML files can be document-centered or data-centered. Document-centered files contain a lot of content, e.g. there are complete books written in XML. These documents can have a very simple structure consisting of chapters and paragraphs. The text – the content of the document – determines the size of the XML file.

The size of data-centered XML files, however, is less dominated by the element content. The elements usually contain much smaller data values. Hence the impact of markups increases – the longer the markup tags for the document, the bigger the file. The test in section 6.2.3 shows that the transformation of two files with the same structure but different length of markup tags are processed in almost the same time. Thus it is better to take the number of XML elements as the measure for the input data.

So the number of elements in the XML document has a major impact onto the processing time. However, if just a very limited number of elements are relevant for the XSL transformation the overall number of elements in the document is not as important. The number of transformed elements matters more.

The tests in this paper are all based on the number of transformed elements in the source document.

3.3 Parameters that can be used to measure performance

When determining the performance of XSL transformations there are two major measures that can be used – the data throughput and the processing time.

3.3.1 Data throughput

The data throughput is used by many benchmarks as a measure for performance. Because of the file size issues mentioned in section 3.2.3 this approach has some disadvantages when working with XML data. Imagine that two files with the same structure but different file sizes are processed. The one with the bigger file size (due to longer markup tags) would always cause a higher data throughput value disregarding the fact that both transformations were executed in exactly the same time. The XSLT stylesheets for this example would be almost identical except from tag names. Hence two identical stylesheets could produce very different results when measuring the data throughput. Obviously the data throughput is not a good measure when comparing the impact of the XSLT stylesheet on the transformation time.

Nevertheless, data throughput can be a reasonable measure when comparing the performance of different XSLT processors to each other. Since input data and stylesheets are fixed the data throughput is an indicator of the XSLT processor speed. The Sarvega XSLT Benchmark Study [sarp] uses data throughput as the measure to compare XSLT processors.

3.3.2 Processing time

The processing time for an XSL transformation is a simple yet accurate measure. Most XSLT programmers are interested in how long their transformation runs. For XML to HTML transformations on web servers the transformation time implies how many users are able to simultaneously browse a website without delays.

In this paper the processing time is used as the measure for all tests.

3.4 Software to measure XSLT performance

In order to measure the performance of an XSL transformation appropriate software is required. The software has to provide the following basic functionality:

- measure the transformation time

The following functionalities are optional, but ease the test work:

- automated tests with different processors
- automated runs with different stylesheets
- structured presentation of results
- measuring of input file size
- measuring of data throughput

The following sections introduce three tools that provide most of these functionalities.

3.4.1 DataPower XSLTMark

The XSLTMark is developed by DataPower Technology in cooperation with the XSLT community. It is a free XSLT processor performance benchmarking application. The software including the source code can be downloaded and executed to test the performance of XSLT processors [xmar].

The XSLTMark is a testing suite of 40 XSL transformations that are applied to several XML files. The XSL files contain a collection of common scenarios for data transformation. The test cases have been designed to cover a variety of possible tasks and input conditions for transformations. The XSLTMark measures the performance in four major categories:

- Pattern Matching
- XPath Selection
- XPath Library Functions
- XSLT Control

Examples are: XML to HTML transformations; sorting, string, number functions; search of elements and key functions [xmar].

In order to execute the benchmark with an XSLT processor a driver for this processor is required. These drivers are usually provided by the vendors of the XSLT processors who want to participate in the tests. The benchmark package includes drivers for the most common XSLT processors. However, since the last release of the XSLTMark these versions are outdated. It is possible to adapt the drivers to newer versions which requires programming effort.

3.4.2 Sarvega XSLT Benchmark

An alternative to the XSLTMark is the XSLT Benchmark. It is developed by Sarvega [sarv] and its design is similar to the XSLTMark's. The benchmark package includes the source code and is available for free [sard].

The XSLT Benchmark includes 15 different XSL transformations. Just like the XSLTMark test cases they are considered to be representative for a variety of possible applications. A detailed description of the test cases and further details about the benchmark are provided in the paper "The Sarvega XSLT Benchmark Study" [sarp].

The XSLT Benchmark can be used with different XSLT processors. Again a driver is required for each processor. Drivers for the most common processors are included in the package. For new drivers the XSLT Benchmark offers an extension mechanism that is easier to manage than the XSLTMark's.

Apart from extending the XSLT Benchmark with new XSLT processors it can also be extended with self-defined transformations by adding new XML input files and XSLT stylesheets. For each test the number of runs, the input XML file and the XSLT stylesheet are defined in a property file. New test cases can be added by adapting this property file.

The output documents are created in separate folders for each processor to enable a comparison whether the same results are created during the transformation. The time to process, data throughput, input file size, output file size and file names are saved in an XML

file. This XML file can be transformed into HTML or Microsoft Excel files to visualize the results.

Due to its good extensibility the Sarvega XSLT Benchmark is used for the tests in this paper.

3.4.3 CatchXSL – The XSLT profiler

When it comes to the in-depth analysis of an XSL transformation the XSLT code needs to be broken down into its atomic parts. A transformation consists of several steps. By separating the different steps from each other it is possible to measure the execution time of every single step separately. The final result of this procedure is the complete transformation time and the information which part of the XSLT stylesheet has the major impact on the processing speed.

CatchXSL is a free tool that provides these functionalities. It is developed by eCube [ecub]. It profiles XSL transformations. Every single XSLT instruction is recorded and the execution time is logged. There is no standard interface to monitor the transformation speed of XSLT processors available yet. Xalan-J and Saxon offer a proprietary interface which is used by CatchXSL. Hence CatchXSL is limited to execute tests with these two processors.

The front end of CatchXSL is a command line interface or a graphical user interface. Figure 3-3 shows the test configuration window. For each test project the processor, the XML input file, the XSLT stylesheet and the number of test runs have to be specified. Optionally, write events can be traced and the XSL output can be generated. If this option is not enabled the result of the XSL transformation is not written to a file.

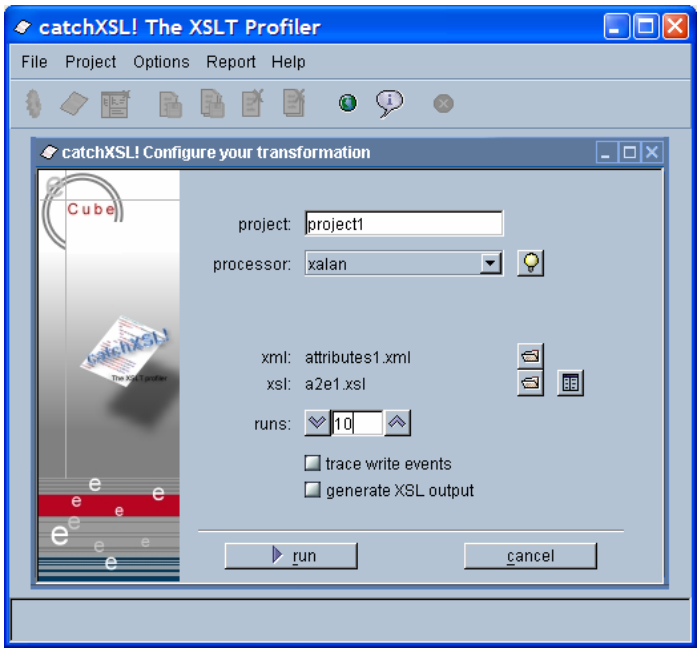

Figure 3-3 Running an XSL transformation with the CatchXSL GUI

At the end of the transformation process CatchXSL offers a detailed list of the times that the single transformations were running (Figure 3-4). The user can investigate these results in order to find the hotspots of the most time consumptive parts in the code.

During the tests it turned out that it is very useful to increase the number of runs up to the point where the measuring process takes up about three minutes. If the tests are repeated less times the results show many variations. With a higher number of transformations the results become more stable. XSLT caching also affects the results. Repeated runs are usually faster than one single run [catx].

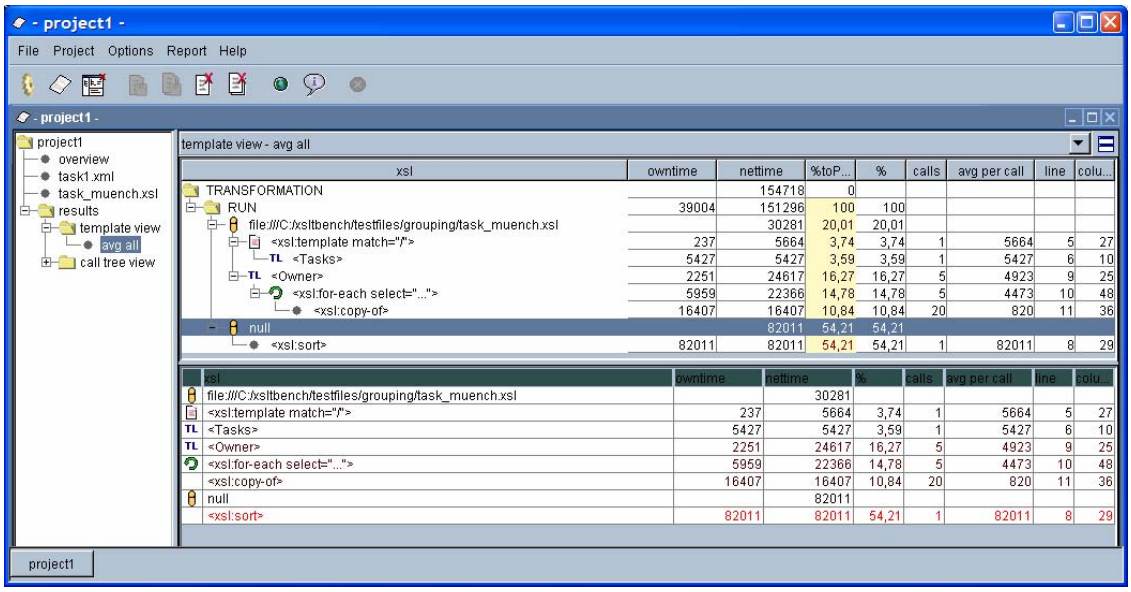

Figure 3-4 Result view of an XSL transformation with the CatchXSL GUI

CatchXSL can be used for a detailed analysis of XSL transformations. Due to its limitation to two XSLT processors it is only used for supplementary tests and the results are not reflected in this paper.

3.5 Used Hardware to measure XSLT performance

The performance of XSL transformation is very CPU-intensive. Complex transformations with large XML files can last up to several hours. Thus the hardware for the test execution has a major impact onto the transformation time.

For the tests in this paper the following machine is used:

Model	IBM Thinkpad A31
CPU	Pentium 4 mobile 1.8 GHz
Main Memory	768 MB
Harddisk	IBM Deskstar 40 GB
Operating System	Windows XP
Java SDK	1.4.2.

During the execution of the tests only the minimum of services and software is running to minimize the impact of concurrent tasks.

4 XSLT processor performance comparison

The XSLT processor has a major impact on the speed of an XSL transformation. The introduction of available XSLT processors in Chapter 2 provides an overview of their characteristics and specifications. Apart from those technical details their performance is important. The following questions have to be answered:

1. Are there differences between the processors?
2. If so – how big are these differences?
3. Are the differences dependent on the type of transformation?

The following chapter presents results of the DataPower XSLTMark and the Sarvega XSLT Benchmark.

4.1 DataPower XSLTMark results

The latest version of the XSLTMark was released in the year 2001. Since then most of the XSLT processors have been constantly improved. The drivers for the XSLT processors that are included in the benchmark package are bound to a certain version of the processor. In order to use the XSLTMark with newer versions of the XSLT processors the drivers have to be modified which includes programming effort for each processor. Hence the XSLTMark was not run with the latest versions of available XSLT engines. The benchmark results in Figure 4-1 are derived from the XSLTMark homepage [xmar].

The first part of the tests includes the parsing process of the input XML files. The second part only measures transformation time. Even though the processor versions are not the latest the results are still valuable. The test shows that there are big differences between the processors.

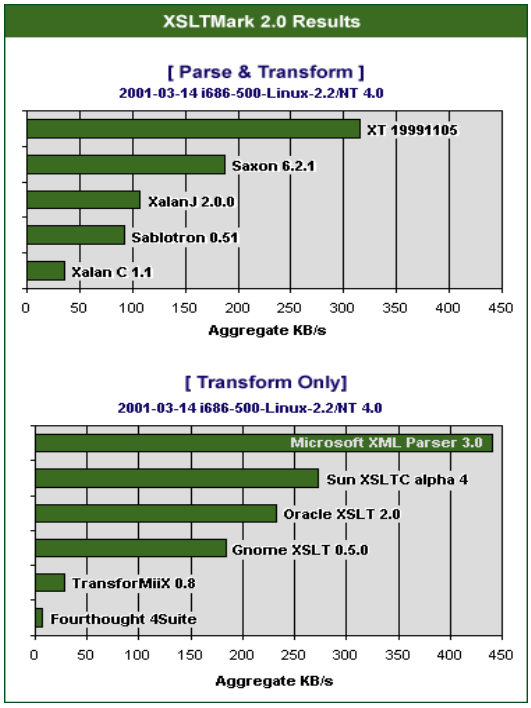

Figure 4-1 XSLTMark 2.0 Results

In the 'Parse & Transform' tests the fastest processor (XT) is almost ten times faster than the slowest processor (Xalan C). For the 'Transformation Only' tests the differences are even bigger. The fastest processor (Msxml) is about 80 times faster than the slowest (4Suite). In general C/C++ implementations are considered to have better performance than their Java pendants. Interestingly this is not the case in this test. The C implementation of Xalan is slower than the Java version.

However, it is important to point out that the benchmark rating also includes failed tests. Some processors do not produce any output with certain transformations because they do not entirely implement the XSLT standard. Those zero-values are reflected in the benchmark result and have a negative effect.

4.2 Sarvega XSLT Benchmark results

The Sarvega XSLT Benchmark was updated in 2003. When it was released the results of the latest XSLT processors were included in the benchmark documentation [sarp]. The overall results of this benchmark are depicted in Figure 4-2.

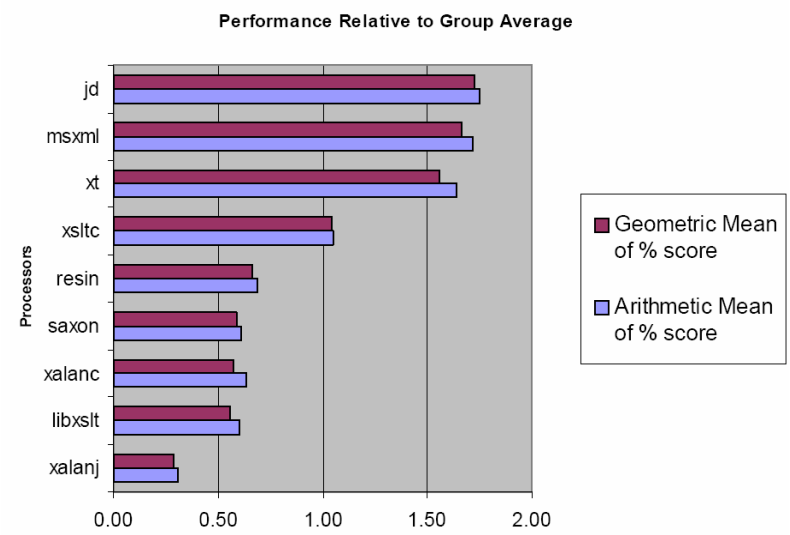

Figure 4-2 Sarvega XSLT Benchmark Results[1]

The results are similar to the XSLTMark. Msxml and XT are the fastest processors. The new processor jd.xslt got the best score. The performance of the popular processor Xalan-J is still the poorest. Unlike in the XSLTMark it is also outperformed by Xalan-C++. Since version 1.1 of the XSLTMark Xalan-C++ experienced many improvements. Hence, the version 1.5 is more optimized and obviously gains benefits from its implementation in C. However, it is still not able to keep up pace with some other Java processors like jd.xslt.

Since the XSLT Benchmark's last release some processors have been improved. In order to test them the Sarvega XSLT Benchmark is executed on the testing configuration described in Section 3.5. These tests are limited to six XSLT processors: jd.xslt 1.5.5, Msxml 4.0, Saxon

[1] derived from [sarp]; used versions: jd.xslt 1.5.1, libxslt 1.0.30/libxml2 2.4.19, msxml 4.0, resin 3.0.1-beta, saxon 6.5.2, xalan-c++ 1.5, xalan-j 2.5, xsltc 2.3.1, xt 20020426a

6.5.3, Saxon 7.9.1, Xalan-J 2.6.0 and XT 20020426a. Figure 4-3 and Figure 4-4 show the results of the 15 test cases. For each test the transformation time of the six XSLT processors is represented by bars. The shorter the bar, the better the performance.

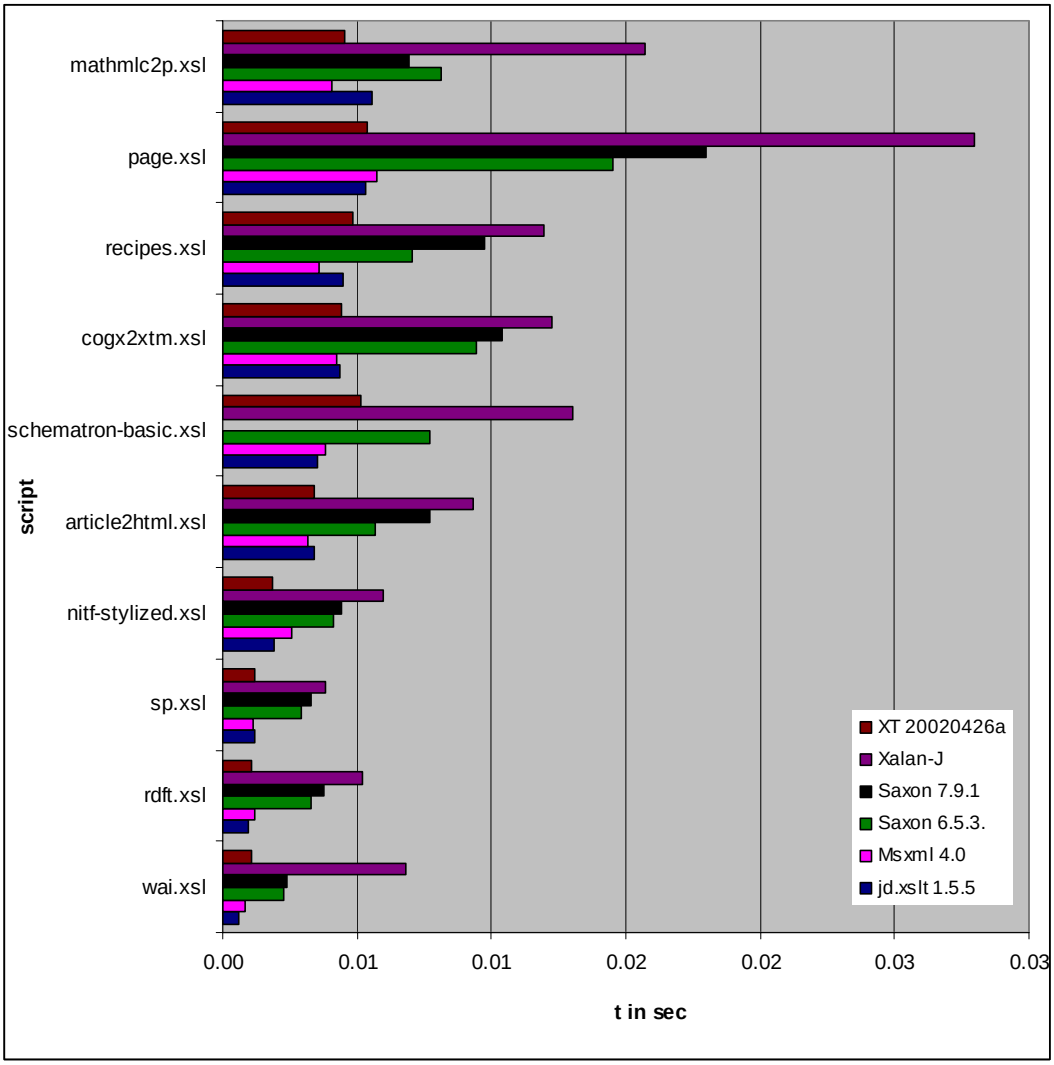

Figure 4-3 Sarvega XSLT Benchmark Results Set I

The first set of transformations, depicted in Figure 4-3, is executed in less than 0.03 seconds. All tests are executed properly except from the *schematron-basic.xsl* transformation that does not produce any output with Saxon 7.9.1.

The second set of transformations, shown in Figure 4-4 is still executed in less than one second. Again the processor Saxon 7.9.1 creates no output for one transformation. This time it is the stylesheet *chess.xsl* that does not produce any result. Hence this transformation as well as the *schematron-basic.xsl* is neglected in the results for this test case.

The behavior of the processors is very homogenous throughout the tests. In the benchmark tests it does not happen that one processor performs extremely good with one transformation but extremely bad with a different one. For most of the scenarios the

sequence of the best performing processors is the same. Xalan-J is the slowest processor for every test. The two versions of Saxon share the next positions. Usually Saxon 7.9.1 performs worse than the older version Saxon 6.5.3. This is probably because of the implementation of XSLT 2.0 functions which have a negative effect on the performance. In addition Saxon 7.9.1 fails to create a correct result for some transformations.

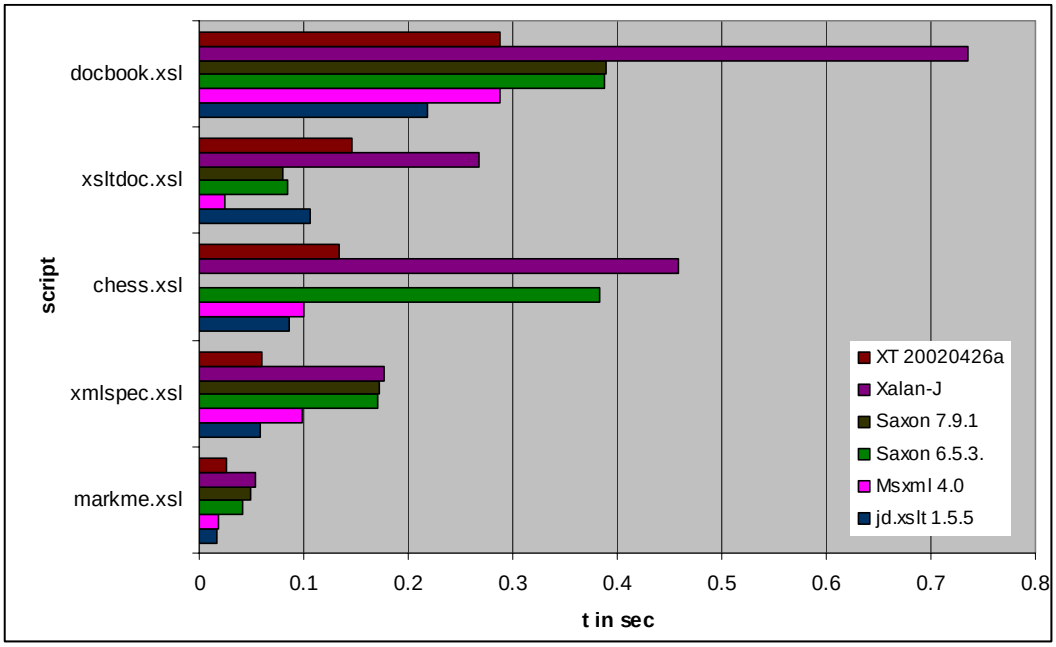

Figure 4-4 Sarvega XSLT Benchmark Results Set II

The three processors XT, jd.xslt and Msxml are pretty close in their performance as well. XT usually is slightly slower than the others, but sometimes a little bit faster.

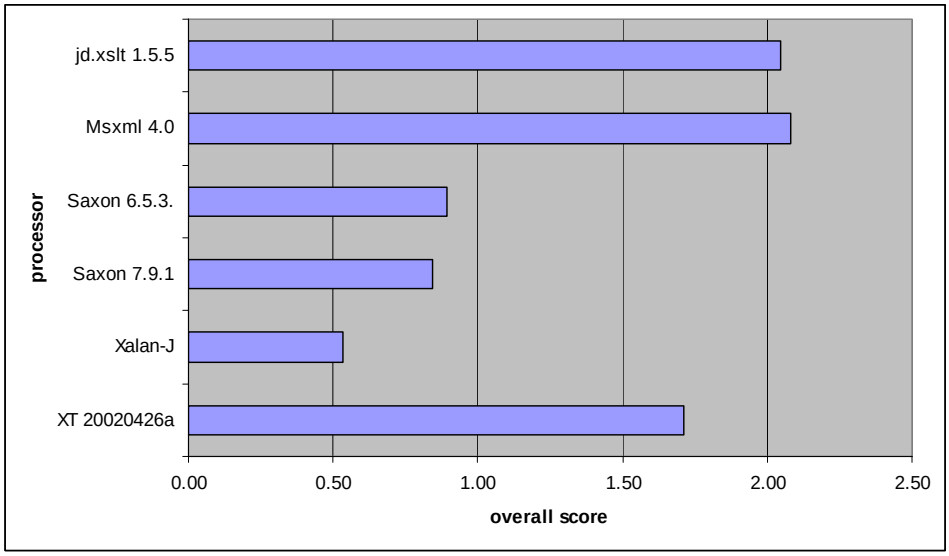

Figure 4-5 Sarvega XSLT Benchmark Overall Score

The overall scores are depicted in Figure 5-5. Msxml got the best score, jd.xslt with slightly worse performance places second. The difference between jd.xslt and Msxml is always very marginally. Sometimes jd.xslt is faster and sometimes Msxml. The example of jd.xslt shows that Java processors can keep up with the transformation speed of C++ pendants. Interestingly the very commonly used engine Xalan-J has only a poor performance.

Supported XSLT functionality of the XSLT processor must not be neglected when comparing the results. If certain XSLT functions are required for an application, the performance is less important. In those cases Saxon can be a better choice than e.g. XT.

5 Performance of Clio-generated XSLT

The following chapter introduces the data mapping tool Clio. First the motivation for the development of Clio is provided. Its functionalities are described. Finally Clio is used to generate XSLT stylesheets and their performance is measured.

5.1 Motivation of the Clio project[2]

The amount of data that is produced worldwide is rapidly increasing. It exists on different media like magnetic and optical discs, tapes and flash memory chips. Storing, querying and integrating this data are major challenges for the information industry.

One challenge is the integration of heterogeneous data. Data intensive applications in electronic document interchange and commerce environments, global information systems and data warehousing integrate data from multiple, often legacy sources. Databases are built using different schemas and data types. Legacy data has to be integrated into new systems. A major problem is the mapping of heterogeneous schemas. XML data also uses different schemas. The data is used in various ways so it has to be transformed from one format into another. This brings up the need for mappings between source and target schemas. Creating the mappings manually is difficult and very time-consuming. Due to the growing amount of data it has to be done more quickly and still accurately. Hence tools that support the user in this process are required. The goal of these tools is the discovery of a query or a set of queries that map the data sources to their new structure.

IBM currently develops a software prototype that has the objective to fulfill these requirements. The name of the project is *Clio*. Clio got its name from the muse of history because it supports the integration of 'historic' legacy data.

5.2 Functionalities of Clio

Clio creates mappings between two data representations semi-automatically. User input is required throughout the mapping process to ensure a correct mapping. Clio supports relational database schemas and XML schemas.

This paper focuses on the mapping component for XML schemas. Figure 5-1 gives a schematic overview of the schema mapping process.

---

[2] Information is derived from [HMH01], [MHH00], [MHH01], [NHT01], [PHV02] and [YMH01].

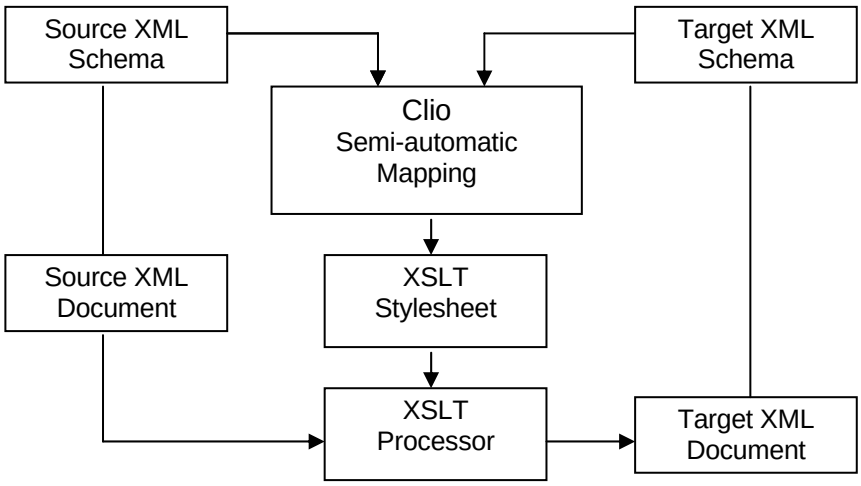

Figure 5-1 Generation of XSLT with Clio (schematic view)

At the beginning of each transformation there is a source XML document that is valid to a source XML schema. There is also a target XML schema. The source XML document has to be transformed into a target XML document that is valid to this target schema. The purpose of Clio is to realize this transformation and make any manual coding of XSLT superfluous.

The first step of the mapping process is the import of the source and the target schema into Clio. Then the user must create a mapping between them. Figure 5-2 shows the graphical user interface of Clio that supports the user during this process. It can easily be done by drawing arrows with simple mouse clicks. These arrows represent references from one schema to the other.

The number of possible mappings between two data sources can be enormous. This is why users are not able to conceive all of the possible alternatives, and hence may have difficulties to find the correct mapping for a specific application. Clio supports the user in finding a proper mapping. It systematically considers and manages alternative mappings. However the final choice of mappings must necessarily be made by a user who understands the semantics of the target application.

Clio uses mining techniques to discover and characterize the relationships between source and target schema and data. It automatically suggests correspondences between source and target attributes. Often the attribute names reveal hardly any information about the semantics of the data values. Only the data values in the attribute columns can convey the semantic meaning of the attribute. For each attribute, Clio analyzes these data values and derives a set of features. The overall feature set forms the characteristic signature of an attribute.

As the result of the mapping process Clio generates XSLT. This XSLT and the source document are processed by an XSLT processor. The output of this transformation is a target document that is valid to the target XML schema.

Dependent on source and target schema Clio also generates XQuery or SQL. For XML transformations the prototype XSLT generator engine creates two different sets of XSLT stylesheets:

- Ordinary stylesheet set
- Minimal union stylesheet set

Each set contains two XSLT scripts. These two scripts are applied sequentially to the input data. The transformation is broken down into these two steps because of performance benefits which became obvious during the development of Clio.

The difference between the ordinary stylesheets and the minimal union stylesheets is their behavior when processing hierarchical data. The minimal union scripts do an additional merge of data in the following scenario:

There is a set of elements where two elements have

1. the same values for their atomic sub-elements and attributes and
2. different set-valued components.

These two elements will be merged into one by merging their subsets of data. The minimal union is a recursive operation.

Example: There are two `<state>` elements, which have three sub elements `<state_name>`, `<country_name>` and `<continent_name>`. The element `<country_name>` and `<continent_name>` are the same, but they have different `<state_name>` elements. The two `<state>` elements would be merged into one by merging their `<state_name>` elements.

Due to this difference for certain input data and transformations the minimal union stylesheets create a different result than the ordinary stylesheets. However, for many cases the result is exactly the same. Hence it is a good example that XSLT is very flexible to use and offers different alternatives to achieve one goal.

5.3 Clio Test cases

There are a lot of different possible Clio transformations. However, there is a certain amount of basic transformations that are part of many different transformation scenarios. The combination of these basic transformations makes up the complete transformation.

The following tests are limited to certain basic transformations. The purpose of the tests is to get information regarding the following questions:

1. Are there performance differences between the ordinary and minimal union scripts when the same output data is created?
2. How big are these differences?
3. Is there a type of transformation that consumes the major amount of transformation time?
4. How do the transformations scale with bigger input files?

5. How feasible are those transformation in general? Are there certain transformations that are unlikely to be executed in reasonable time at all?
6. Are there XSLT processors that perform much better than others and therefore should be used preferably for certain transformations?

The Sarvega XSLT Benchmark is enhanced with the Clio-generated scripts to measure their performance. The Java XSLT engines jd.xslt, Xalan-J, XT, Saxon 6.5.3 as well as Saxon 7.9.1 and the C++ XSLT processor MSXSL 4.0 are used.

The input data is generated with ToXgene. The files size grows – the number of elements doubles from one file to the other. Hence the charts do not represent a linear curve if the transformation time grows linearly. However this presentation style was chosen to show the development of the transformation speed with small and big XML files without executing many tests. The complete test result numbers are listed in the appendix.

5.3.1 Transforming attributes to elements

The transformation of an XML element into an XML attribute is a commonly used data transformation. Figure 5-2 shows a simple transformation of an XML element into an attribute.

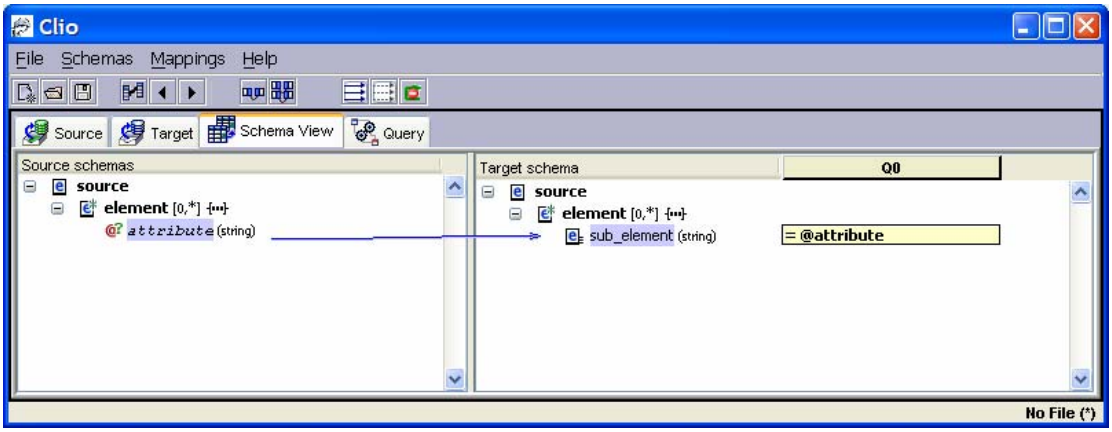

Figure 5-2 Clio mapping attributes to elements

The number of `<element>` elements has the major impact on the transformation speed because it is the only element that is processed during the transformation.

The intermediate output that is generated by the first script has more elements than the original document because for every source attribute an additional element `<sub_element>` is created. During the second transformation only these elements appear in the final result. Their number is equivalent to the number of `<element>` elements from the input file.

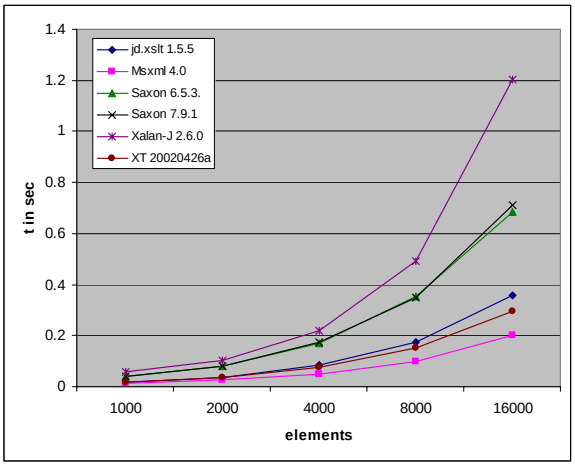

Figure 5-3 Results a2e1.xsl

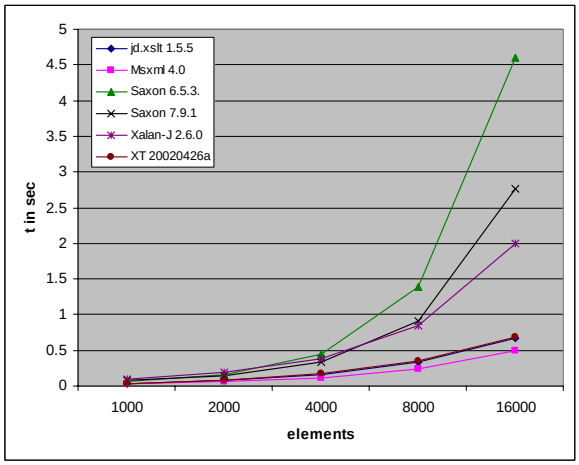

Figure 5-4 Results a2e2.xsl

For the first script the transformation is finished in less than two and for the second script in less than five seconds. Obviously the processing time grows linearly with an increased number of elements.[3]

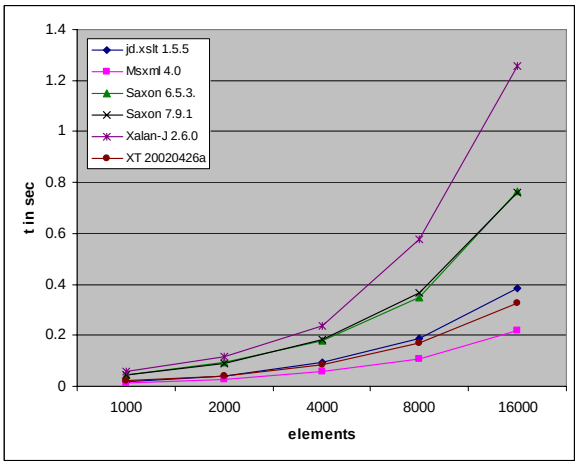

Figure 5-5 Results a2emu1.xsl

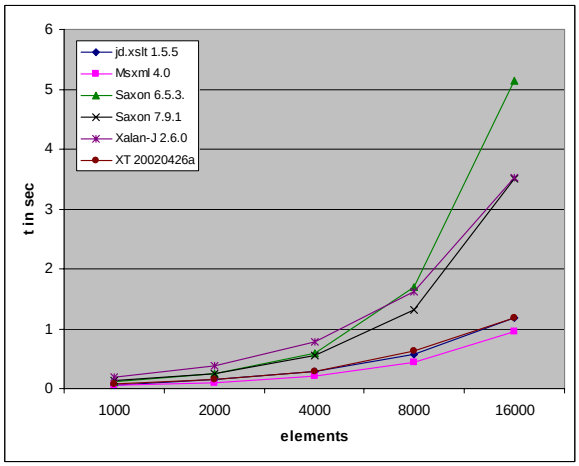

Figure 5-6 Results a2emu2.xsl

The transformation using the minimal union stylesheets shows a worse performance than the ordinary transformation. The first script is completed in almost the same time as before. The intermediate result file is bigger because it has additional attributes for the `<element>` element.

The second transformation of the minimal union stylesheet set is processed slower than the ordinary second script. The XSLT code is more complex. This is a good example for two stylesheets that create the same results but are processed within different times.

[3] Once again the hint: Due to doubling the file size the curve looks square even if the growth is linear.

5.3.2 Transforming elements to attributes

The opposite direction of the preceding transformation converts XML elements to XML attributes. The mapping is shown in Figure 5-7.

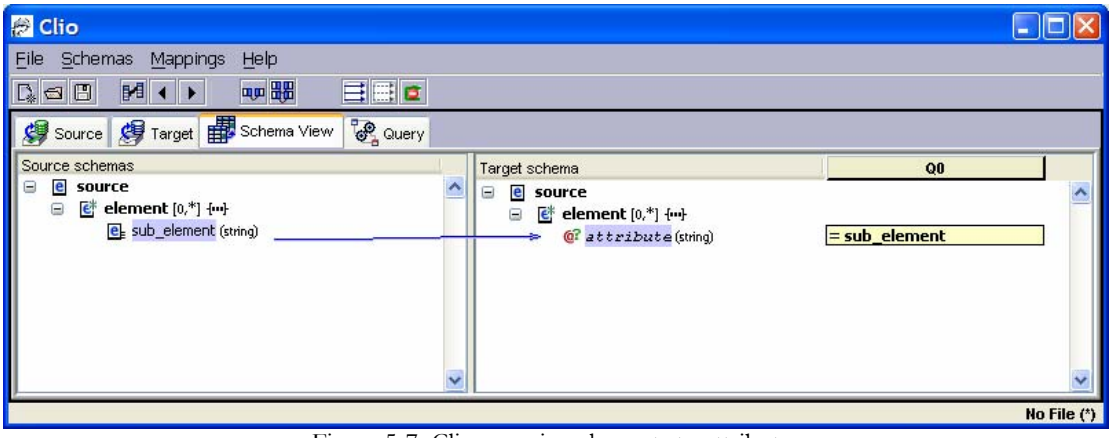

Figure 5-7 Clio mapping elements to attributes

The ordinary transformation is completed in less than five seconds. The processing time is approximately the same as in the transformation of attributes to elements.

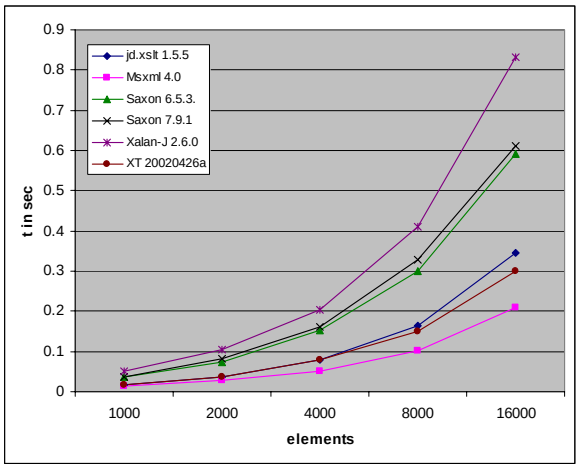

Figure 5-8 Results e2a1.xsl

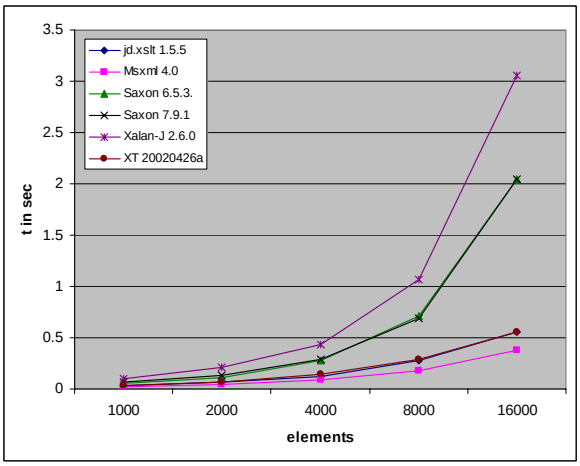

Figure 5-9 Results e2a2.xsl

For the transformation with the minimal union option the behavior is comparable to the attributes-to-elements transformation, too. The time consumption of the second script is higher which makes the overall performance of the minimal union transformation worse than the ordinary transformation.

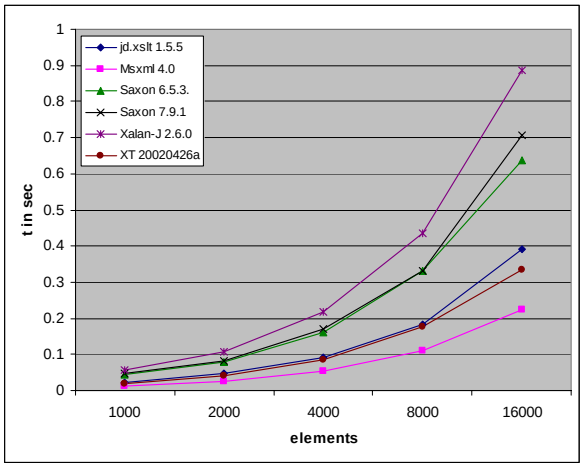

Figure 5-10 Results e2amu1.xsl

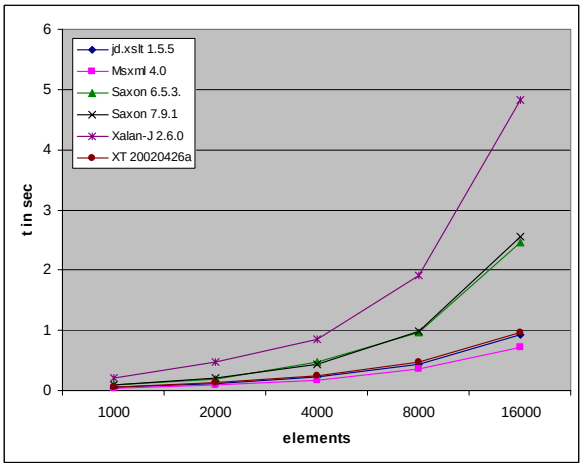

Figure 5-11 Results e2amu2.xsl

The elements-to-attributes transformation is another example how two different ways of producing the same result can consume different times.

5.3.3 Flat hierarchy to flat hierarchy

The following mapping is a simple copy of XML elements. The source schema is equal to the target schema. The purpose of this transformation is to check the performance of a very basic copying process without any structural changes of the XML document. This knowledge is helpful for the upcoming transformations.

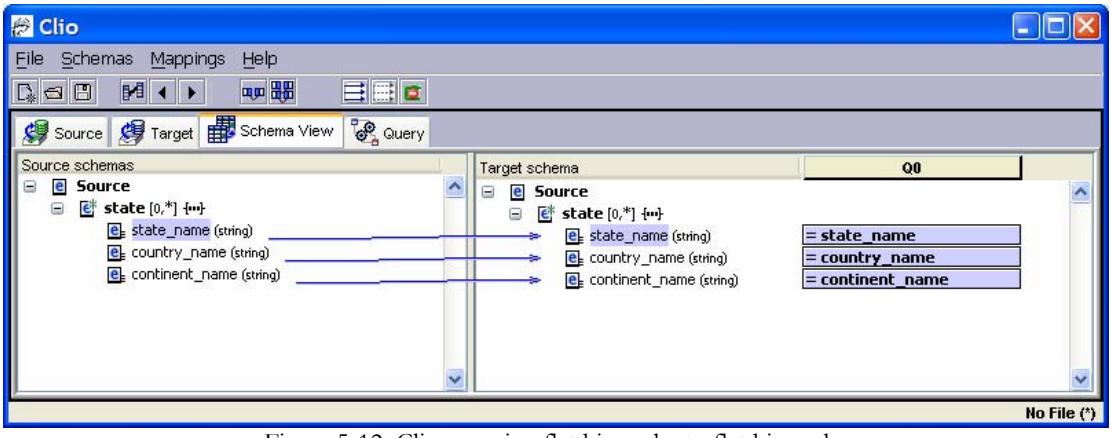

Figure 5-12 Clio mapping flat hierarchy to flat hierarchy

The input document is generated with ToXgene. It has 100 to 1600 `<state>` elements. Each of them has the three sub-elements `<state_name>`, `<country_name>` and `<continent_name>`. The ordinary transformation is executed very fast for the first as well as the second script.

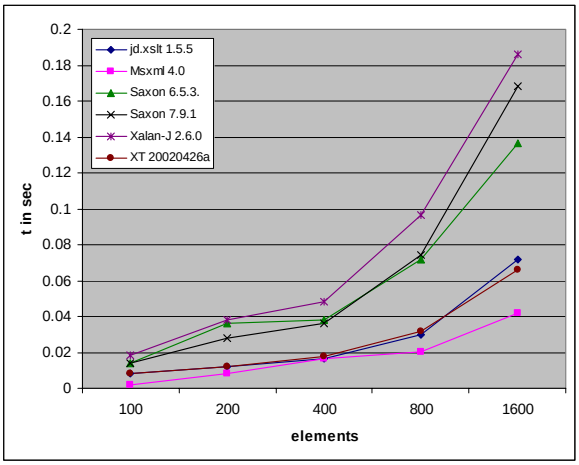

Figure 5-13 Results f2f1.xsl

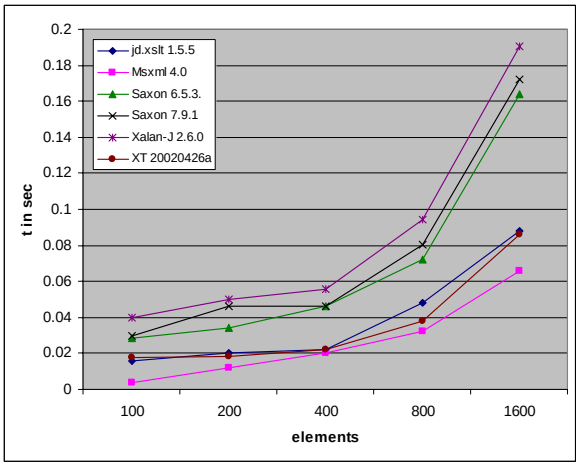

Figure 5-14 Results f2f2.xsl

The transformation with the minimal union option is executed slightly slower than the ordinary transformation.

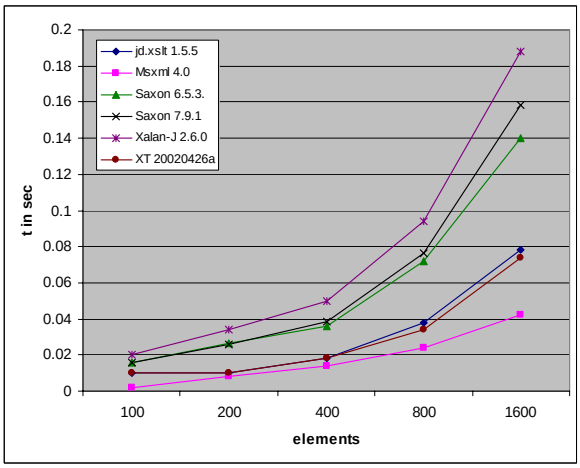

Figure 5-15 Results f2fmu1.xsl

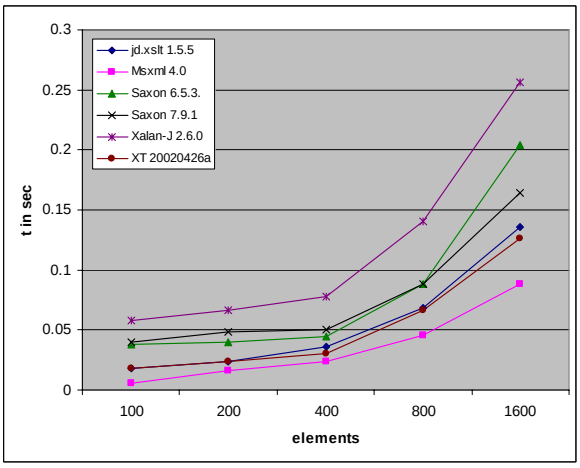

Figure 5-16 Results f2fmu2.xsl

Obviously the execution time for the ordinary transformation as well as the minimal union transformation grows linearly with an increasing number of input elements.

5.3.4 Flat hierarchy to nested hierarchy

For this test Clio is used to generate a mapping from a flat hierarchy to a nested hierarchy. The number of nesting levels is varied from two to four. The sample scenario is represented by a hierarchy of continents that contain countries. These countries contain states. For the sample XML files the names of the elements are represented by integer values to keep the file structure simple. It is possible that e.g. one country name shows up multiple times, where each time it belongs to a different continent. This might not match reality however it keeps the grouping of the values simple.

Two levels of nesting

With two levels of nesting the input file has a number of `<country>` elements. Each one has the sub-elements `<continent_name>` and `<country_name>`. These elements are mapped to a number of `<continent>` elements where each one has a sub-element `<continent_name>` and several sub-elements `<country>`. The `<country>` element contains all the `<country_name>` elements that have the same continent name.

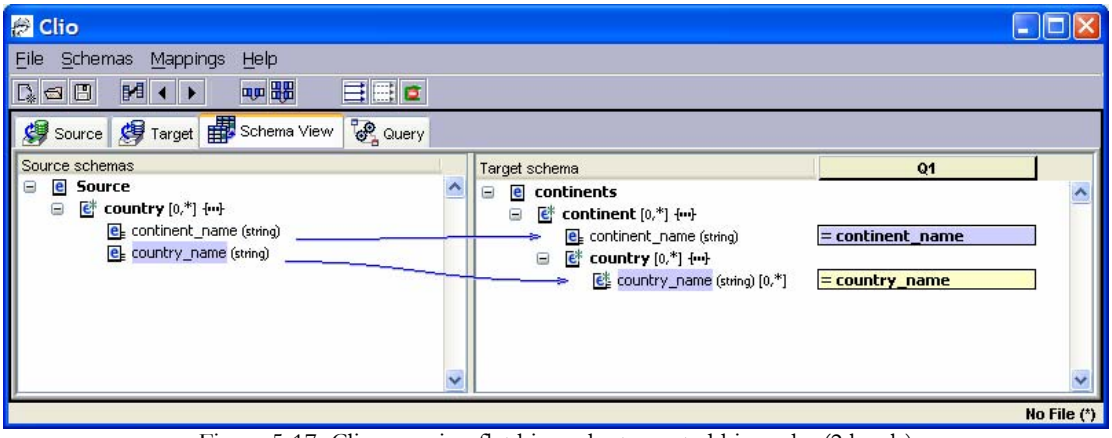

Figure 5-17 Clio mapping flat hierarchy to nested hierarchy (2 levels)

So this transformation realizes a grouping of all countries according to the continent that they belong to. Multiple `<country>` elements that have the same `<continent_name>` and `<country_name>` appear only once in the result.

The input document has 100 to 1600 `<country>` elements. The file size varies from 8 KB to 130 KB.

The results of the ordinary transformation are depicted in Figure 5-18 and Figure 5-19.

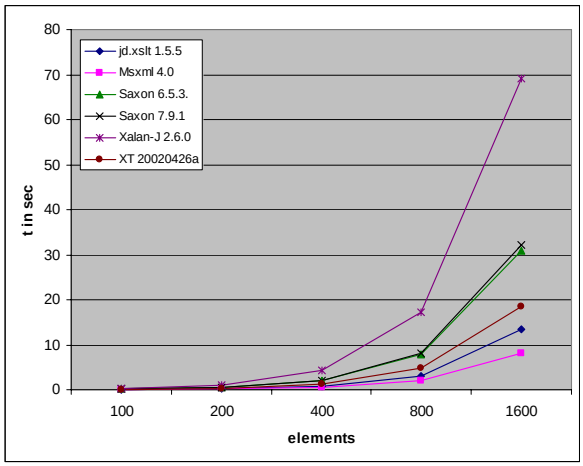

Figure 5-18 Results f2n2l1.xsl

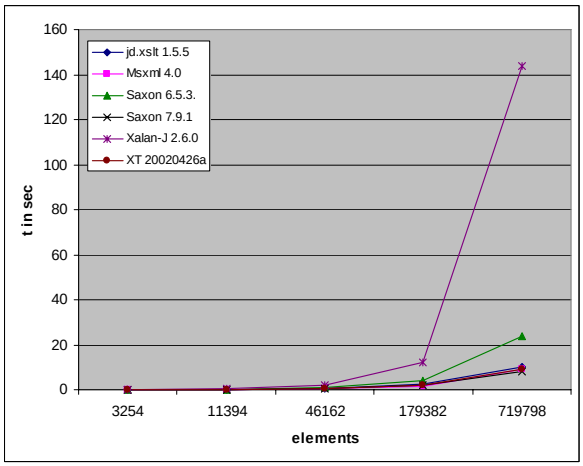

Figure 5-19 Results f2n2l2.xsl

The first script creates huge files as intermediate results. For an input file with 1600 `<country>` elements (130 KB) the intermediate file has still 1600 `<country>` elements. However, each one includes not only one `<continent_name>` but also all `<country_name>` elements that belong to this `<continent_name>` element. This

increases the intermediate file size to 22.1 MB. The elements in the chart are `<country_name>` elements.

Due to this huge file size the XSLT processor requires up to 250 MB of memory during the transformation[4]. Having in mind the size of the input document (130 KB) this is a huge amount. Obviously the execution time grows faster than linearly.

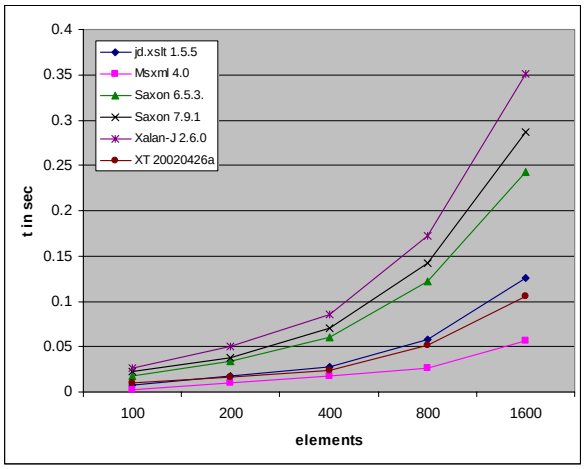

Figure 5-20 Results f2n2lmu1.xsl

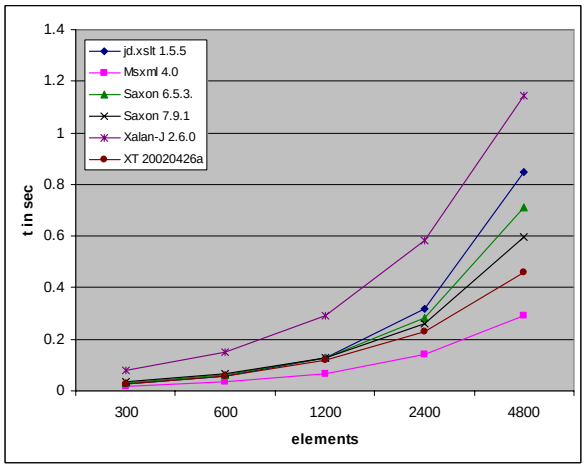

Figure 5-21 Results f2n2lmu2.xsl

With the minimal union option the transformation is executed much faster and consumes less memory during the transformation process. The intermediate result file that is generated is much smaller (750 KB instead of 22.1 MB).

The execution time grows linearly – a big advantage compared to the ordinary scripts.

Three levels of nesting

Increasing the nesting level by one adds the `<state_name>` to the input schema. The source document contains a list of states where each one has its `<state_name>`, a `<country_name>` and a `<continent_name>`.

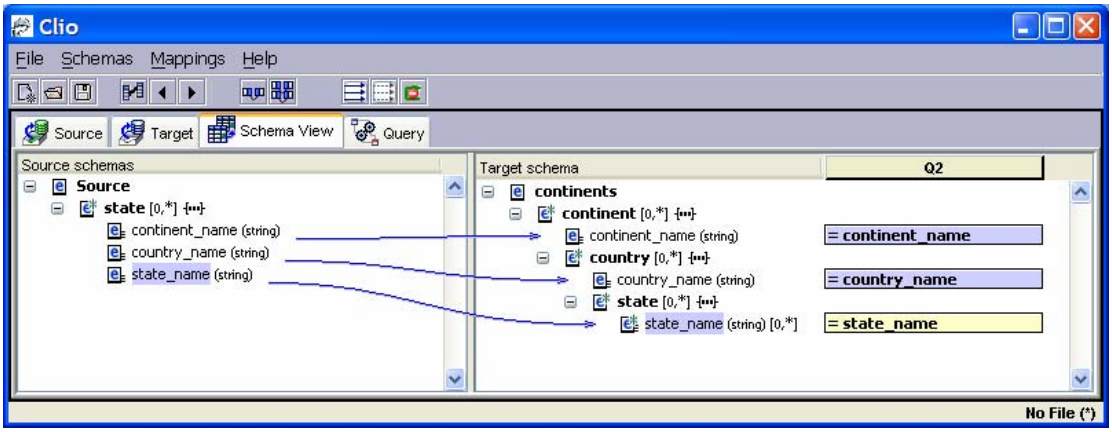

Figure 5-22 Clio mapping flat hierarchy to nested hierarchy (3 levels)

[4] This value is measured with the Windows Task Manager. The value is not expected to be very accurate, but it gives a good idea of the consumed memory.

The mapping in Figure 5-22 shows that the target schema does two grouping operations now. The countries that belong to a continent are assigned to a group and each of the countries contains all the states that belong to them as a sub-element. Just like the preceding transformation with two levels of nesting multiple combinations of `<continent_name>`, `<country_name>` and `<state_name>` elements appear only once in the result. The increased complexity of the transformation is reflected in the test results in Figure 5-23 and Figure 5-24. The elements in the chart are `<state>` elements.

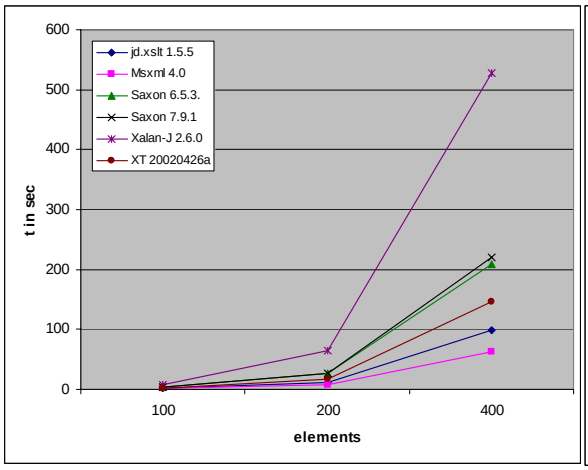

Figure 5-23 Results f2n3l1.xsl

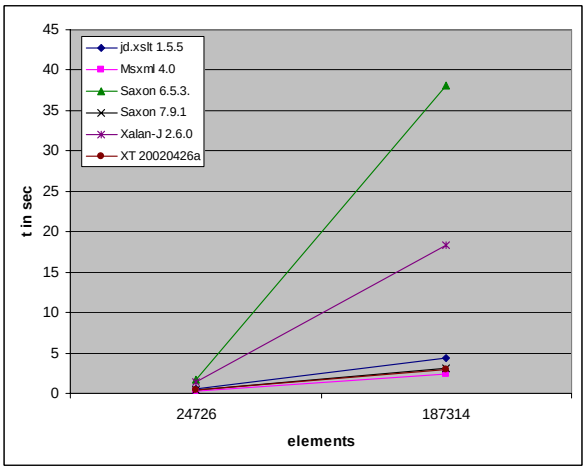

Figure 5-24 Results f2n3l2.xsl

The first script of the ordinary transformation last between 61 seconds (Msxml) and 528 seconds (Xalan-J). Due to this long time the test is limited to only three different input documents. The intermediate result that is generated for the input file with 100 `<state>` elements (47 KB) has a size of 65 MB. The memory consumption during the transformation process is up to 300 MB.

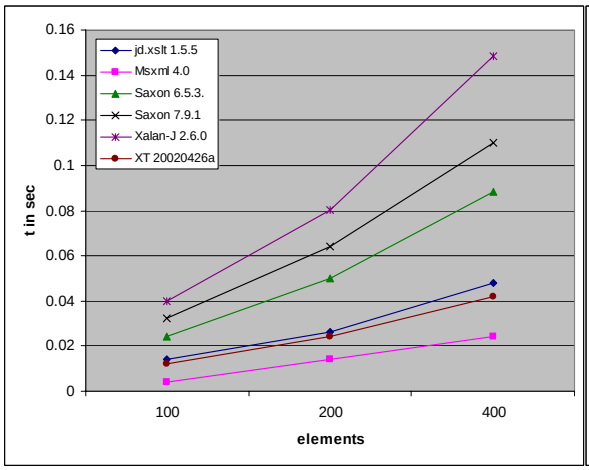

Figure 5-25 Results f2n3lmu1.xsl

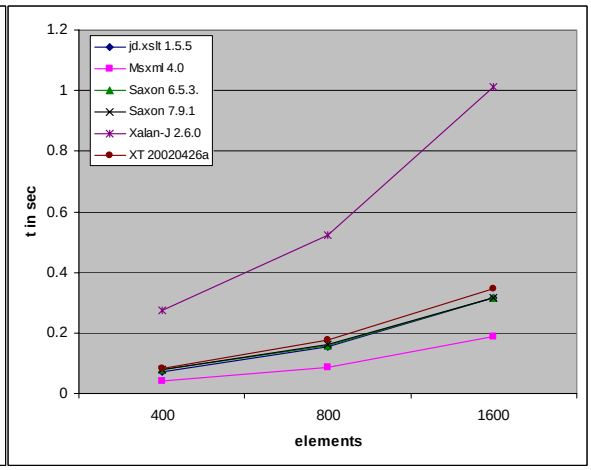

Figure 5-26 Results f2n3lmu2.xsl

With the minimal union option the transformation finishes much faster and consumes much less memory. The elements for the second script are `<state_name>` and `<ClioSet>` elements. The transformation time is reduced from about 550 s to approximately 1 s for the XSLT engine Xalan-J, which is enormous.

Four levels of nesting

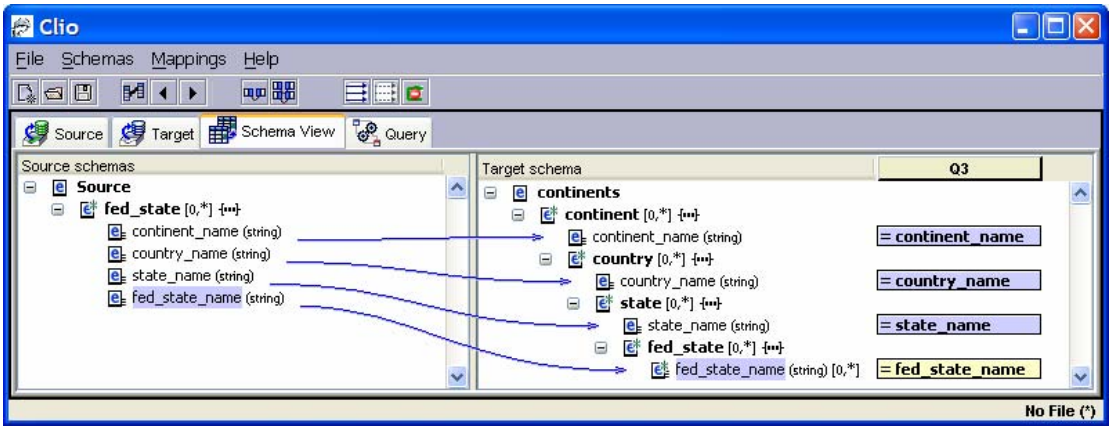

Figure 5-27 Clio mapping flat hierarchy to nested hierarchy (4 levels)

This test is not feasible. With a sample XML file that contains 100 `<fed_state>` elements (57 KB filesize) the first script consumes more than 300 MB of memory. The transformation is too slow to finish in a reasonable time.

Unlike the transformations in 5.3.1 and 5.3.2 this test shows that XSL transformations can consume a lot of memory and time to process. An increased complexity of the transformation causes an increase of the transformation time. In addition the time to process grows non-linearly with an increasing file size.

Another result that became obvious during this test is that the realization of an XSL transformation is very important for the performance. The ordinary script and the minimal union scripts created the same result. However, the execution time of the minimal union scripts was much better. Hence it is important to know about theses differences. This knowledge enables XSLT programmers to optimize their manually written code. It is also helpful to take these differences into consideration for the automatic generation of XSLT code.

5.3.5 Nested hierarchy to flat hierarchy

The opposite operation of the preceding transformation is the mapping of a nested hierarchy to a flat hierarchy. Due to the exclusion of multiple combinations of `<continent_name>`, `<country_name>` and `<state_name>` it is impossible to run the inverse transformation with the output file from 5.3.4 and generate the input file that was used for the transformation in 5.3.4. That is why a new set of input files is generated with ToXgene for this test. The number of `<state_name>` elements varies from 100 to 1600.

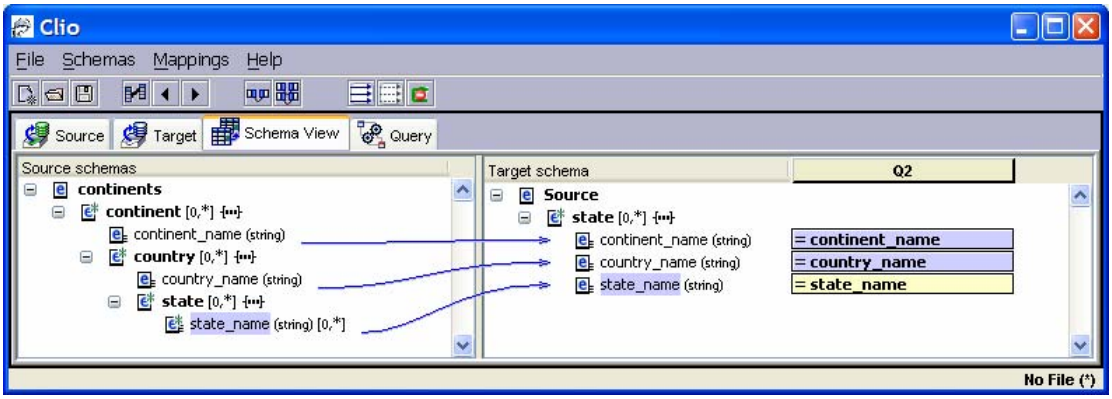

Figure 5-28 Clio mapping nested hierarchy to flat hierarchy (3 levels)

The transformation is executed in less than one second for all XSLT processors.

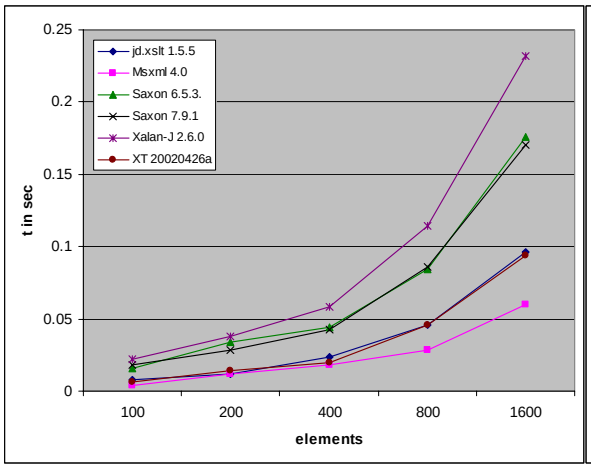

Figure 5-29 Results n2f1.xsl

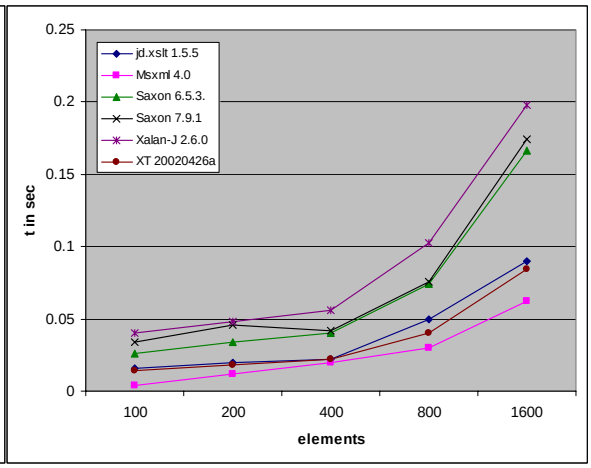

Figure 5-30 Results n2f2.xsl

The minimal union transformation finishes in almost the same time.

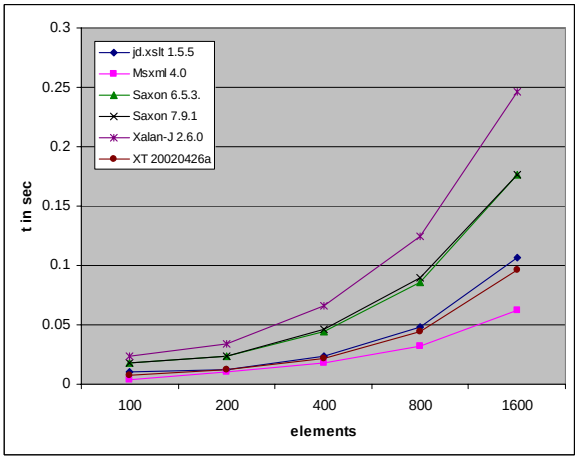

Figure 5-31 Results n2fmu1.xsl

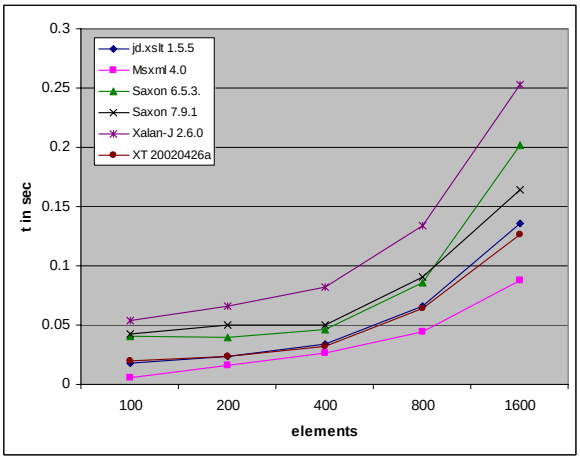

Figure 5-32 Results n2fmu2.xsl

This test showed that for certain XSLT transformations different approaches can produce the same result in the same time.

5.3.6 Nested hierarchy to nested hierarchy

The mapping from a nested hierarchy to a nested hierarchy looks like a simple copy operation. However for this test case there is more to it. During the transformation multiple occurrences of the same continent or country are put together into one hierarchy level.

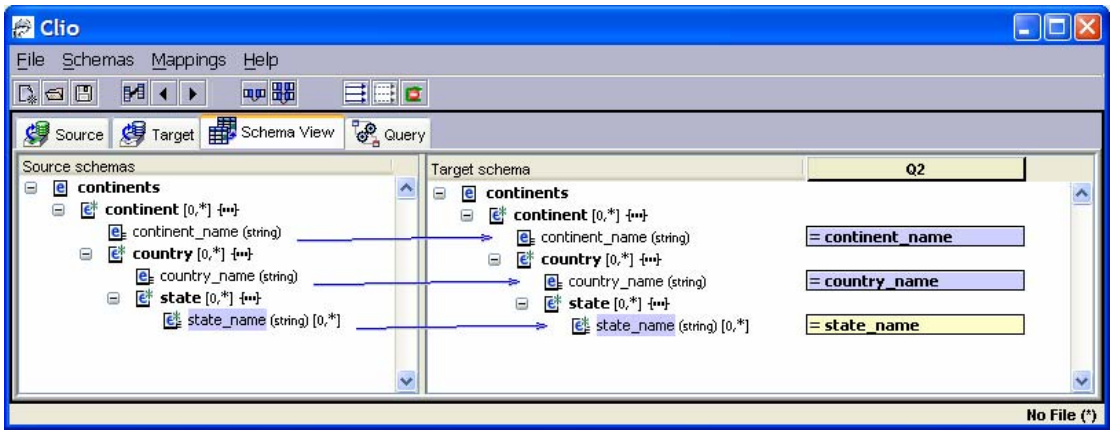

Figure 5-33 Clio mapping nested hierarchy to nested hierarchy (3 levels)

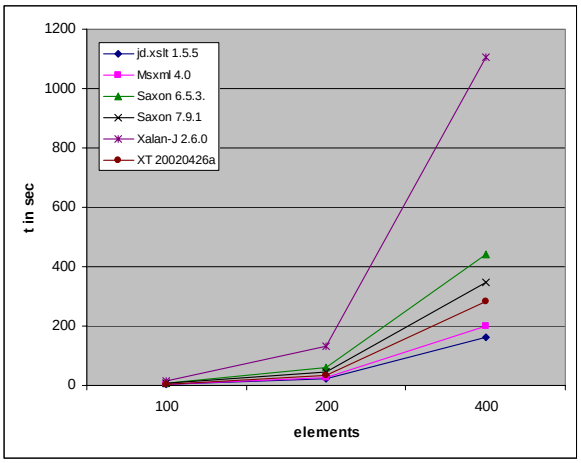

Figure 5-34 Results n2n1.xsl

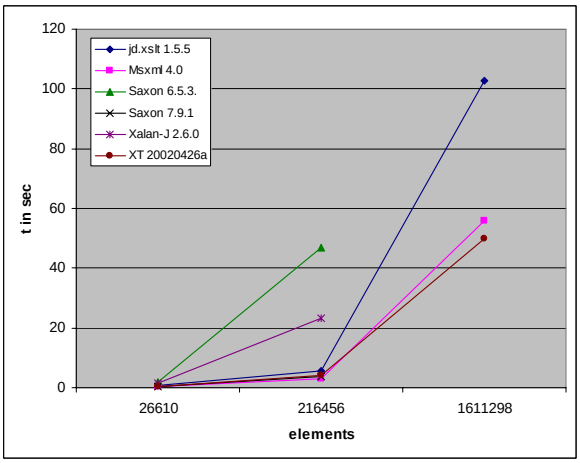

Figure 5-35 Results n2n2.xsl

Figure 6-34 and Figure 6-35 show that this transformation is very slow. For the ordinary scripts the intermediate result file is very big (66 MB for an input file of 58 KB).
The first script consumes up to 340 MB of memory during the transformation. The second script is even worse and takes up to 500 MB of memory during the transformation. The elements in the chart are `<state_name>` elements. The second script is not executed for all the input files with Saxon 6.5.3 and Xalan-J because the tests did not complete in a reasonable time.

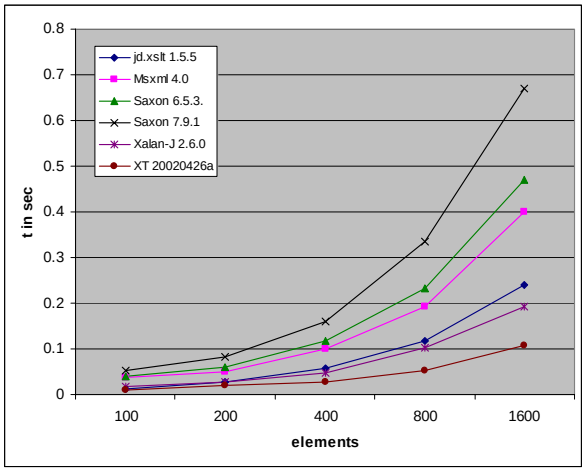

Figure 5-36 Results n2nmu1.xsl

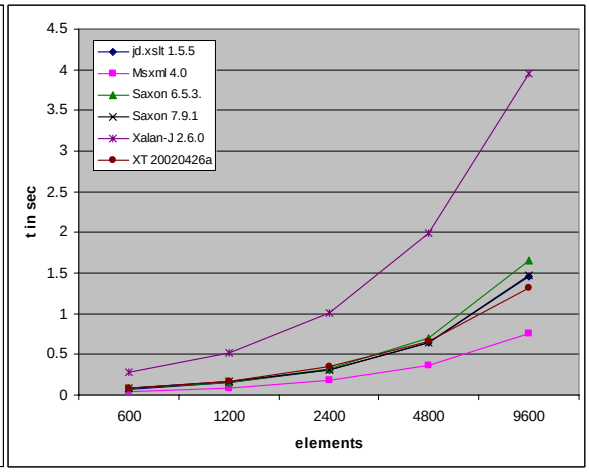

Figure 5-37 Results n2nmu2.xsl

The tests with the minimal union option are executed much faster – however they are producing a different result. This is due to the additional merge operation that is done in the minimal union stylesheets. The elements in the second script are `<state_name>` and `<ClioSet>` elements.

This last test shows big performance differences between the ordinary stylesheets and the minimal union stylesheets. Different results are produced. However, one idea becomes obvious: For some transformations modifications of the data structure that enable the usage of a better performing XSLT stylesheet and still produce the same result are worth to be taken into consideration.

5.3.7 Summary for Clio transformations

The tests of Clio generated XSLT stylesheets showed performance differences between the ordinary scripts and the minimal union scripts. For some transformations these differences were very small, for others they were enormous, e.g. the transformation from a flat into a nested hierarchy which performed about 500 times better with the minimal union scripts. However, the minimal union stylesheets did not have the best performance all the time. For example the transformation of attributes to elements was executed faster with the ordinary stylesheets.

Apart from the effect of stylesheets the XSLT processors had a major impact on the transformation time. The sequence of best performing processors is the same as in the XSLT Benchmark tests. However, the differences are more obvious. Due to an increased number of elements the disadvantages of XSLT engines like Xalan-J turn out more clearly. In general Msxml has the best performance.

Some transformations are not feasible to be used in highly demanded data transformation areas. Their execution is too slow. Especially with an increased number of elements some transformations could last for hours. The dependency on the number of elements is linearly for many transformations. The most time consumptive operations are modifications of the document structure. For the Clio tests this is for example the transformation from a flat hierarchy into a nested hierarchy. Due to multiple nested for-each-loops that are used in the ordinary scripts to perform the transformation the processing time grows non-linearly. The

minimal union stylesheets work with less for-each-loops and show a much better performance.

The Clio-supported generation of XSLT saves a lot of time compared to manually coding XSLT. The transformations can be performed quickly and results are presented instantaneously. The mapping component of Clio helps to find a correct mapping of two data representations. Nevertheless there is one downside to the automatic generation of XSLT. The code is harder to read for humans. In order to manually adapt this code the developer needs profound XSLT knowledge.

6 Improving XSLT performance

The Clio tests in Chapter 5 show that XSL transformations speed can be very slow. So it is important to know how the XML data, the XSLT code and the XSLT processor can affect the performance of the XSL transformation.

Chapter 6 covers several aspects and approaches of how the performance of XSL transformation can be improved. There has not been a lot of research in this area. The following sections introduce some ideas how to avoid some pitfalls and give a number of improvements that can be applied by XSLT developers to improve their XSLT code.

Some references to existing ideas and approaches of how to improve XSLT performance are given in Section 6.1.

When thinking of ways to improve XSLT performance there are three parameters that can be modified in order to achieve that goal. At first the input documents can be changed. In Section 6.2 the question of how the document structure affects the transformation speed is discussed.

The second possibility is to modify the XSLT code. As seen in Chapter 5 there are often different ways to realize an XSL transformation. The difference of the performance of these solutions can be enormous. In Section 6.3 some ideas are tested and different approaches are compared to each other.

Another way to improve the XSL transformation speed is optimizing the XSLT processor. Section 6.4 talks about some ideas of how XSLT processors can be optimized.

Another approach is to implement XSLT functionalities in a hardware device. DataPower delivers its hardware XSLT processor as a solution for web development. More details can be found in Section 6.5.

6.1 Existing approaches

Since 1999 the popularity of XSLT increased continuously. Nevertheless, it is still a new language and especially its performance was neglected by users and XSLT processor implementers for a while.

Currently available XSLT literature covers the question of performance as a minor point. There is one section in Michael Kay's *XSLT Programmers Reference* [Kay00] that presents some ideas of how XSLT performance can be improved. Sal Mangano gives some hints in his

XSLT Cookbook [Man03] as well. In *Professional XSL* [CCD01] the authors also introduce certain ideas to positively affect XSLT performance.

Additional advice can be found in newsgroups like the XSL-list [mulb]. Michael Kay wrote a list of performance improvements that was posted on the XSL-list [dpaw]. Jenni Tennison has a small section about XSLT performance improvements on her website [jten], too. These sources give a starting point for the investigation of XSLT performance. In the following sections some of these ideas are investigated by creating sample scenarios.

6.2 Modifications of input/output documents

The first way to improve XSLT performance is to change the structure of the XML data that is transformed. Of course there is the question whether the effort of changing the XML document is worth of being done just in order to speed up another transformation. Sometimes the efforts of changing the structure of the document are useless because the additional transformation destroys all the benefits.

When developing new XML applications and new XML schemas developers have complete control of the XML data. They can create it according to the needs of fast transformations. They take this XML data to transform it into other formats e.g. for presenting it on different hardware devices like PC, cell phone or PDA. If performance problems occur during these transformations the developers could store their data according to a different XML schema – a schema that allows faster transformations for presenting the data.

Some ideas of how to modify the data format for input documents are discussed in the following sections. For a subset of these ideas test cases are executed in order to measure performance benefits.

6.2.1 Splitting up big input files

One idea, also recommended by Michael Kay, is to split up big input files into smaller units. In order to find out whether this assumption has performance advantages the biggest file of the transformation of elements to attributes (Section 5.3.2, 16000 elements, 12.6 MB) is split into ten files. The XSLT stylesheet is applied to these files.

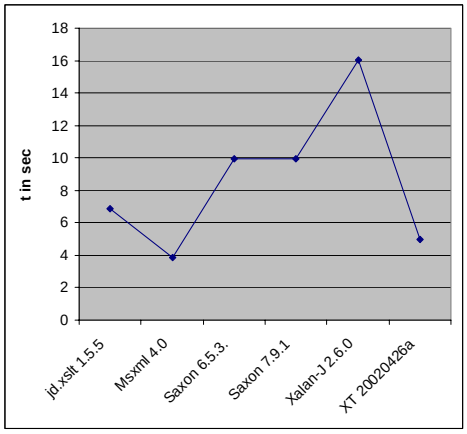

Figure 6-1 Results e2a1.xsl – Complete file

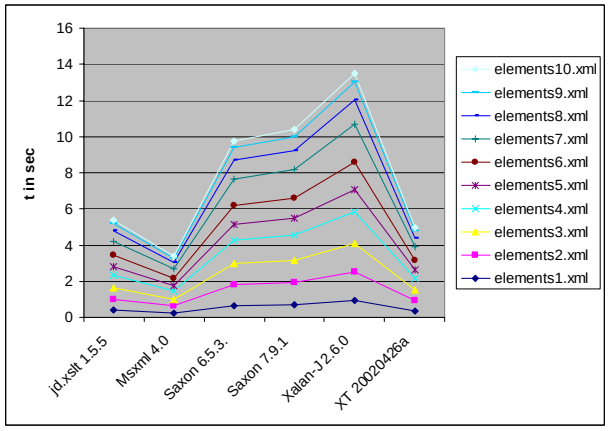

Figure 6-2 Results e2a1.xsl - Sum of single files

When the first script is applied to the complete file, the transformation time varies from 3.8 s (Msxml) to 16 s (Xalan-J). The same transformation executed with the separate files is completed in only 3.8 s (Msxml) to 13.5 s (Xalan-J). The chart in Figure 6-1 shows the transformation time for the complete file. Figure 6-2 depicts the single transformations and their sum which is represented by the curve at the top.

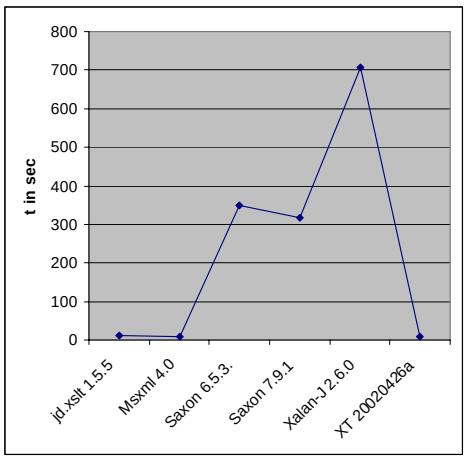

Figure 6-3 Results e2a2.xsl - Complete file

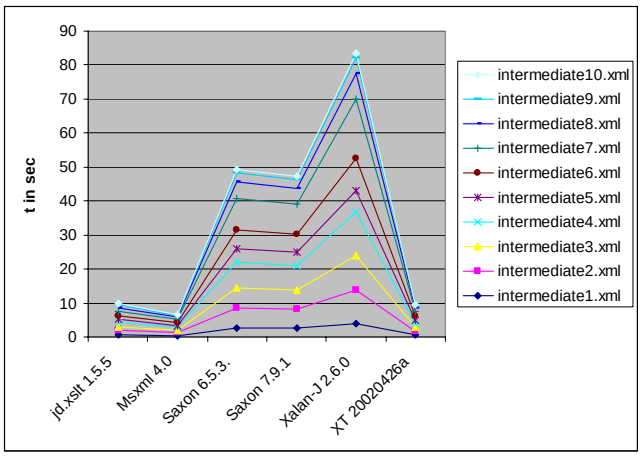

Figure 6-4 Results e2a2.xsl - Sum of single files

The difference between the complete file and the single files is bigger for the second transformation. While the slowest processor Xalan-J needs 700 s to finish the complete file it only takes 83 s for the single files. The memory consumption is lower, too.

For this test case there are performance benefits when splitting up an input file into smaller units. However, one issue that is neglected in this scenario is that the input file had to be split up. For this test it was done manually. Taking this additional time as well as the time for the merge of the output documents into consideration the advantage might vanish for other cases.

In addition the separate processing for single file units is not possible for every scenario. Complex hierarchical data that is represented in one schema can not be torn apart into multiple files.

6.2.2 Using attributes instead of elements

The number of nodes determines the complexity of a document. Thus reducing the nodes by using attributes instead [dpaw] could reduce the complexity of a transformation.

In order to check the consequences of this modification two sets of files are created. Each set contains ten files. The number of elements of the first set varies between 1,000 and 256,000. Each element contains one integer number. The second set contains the same integer numbers. However, they are saved as an attribute of the parent element.

The XSLT that is applied to the input files lists the content of all elements or attributes.

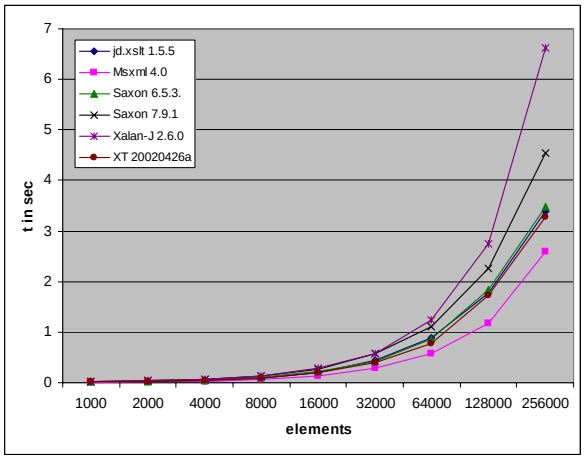

Figure 6-5 Results elementcontent.xsl

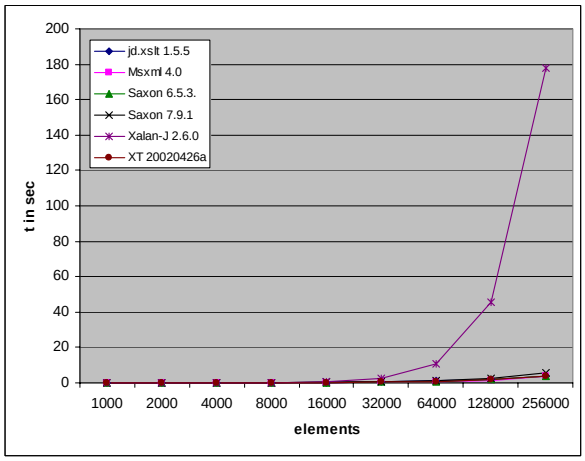

Figure 6-6 Results attributecontent.xsl

Obviously the execution of the transformation is faster when using attributes instead of elements. The differences occur with all processors. They vary between 2.5 per cent (XT) and 36 percent (Msxml). However, they become most obvious with Xalan-J. The *attributecontent.xsl* script is executed about 25 times faster than the *elementcontent.xsl* script.

Obviously the usage of attributes instead of elements can have a positive effect onto the transformation speed.

6.2.3 Keep tag names short

The idea is that using shorter tag names for elements or attributes could reduce the transformation speed of a script. In order to investigate this idea the same transformation is applied to two sets of files where the only difference of these sets is the naming of the elements. In one file the names for the tags are much longer than in the other one. This difference also results in an increased file size.

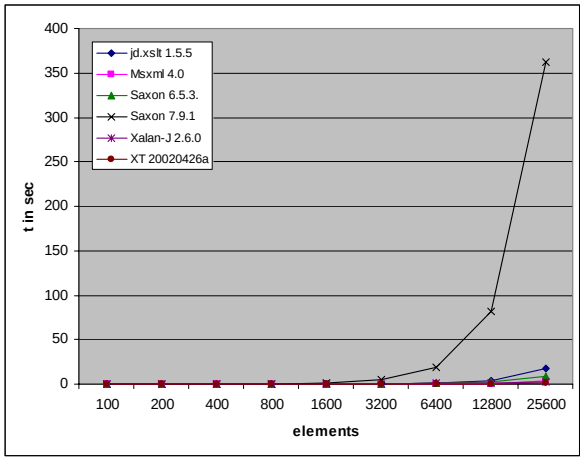

Figure 6-7 Results grouping_muench.xsl

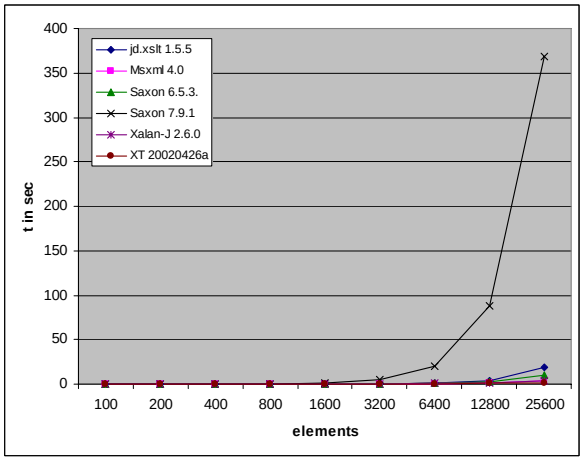

Figure 6-8 Results grouping_muench_long.xsl

The test shows that the length of the tag-names has only a minor effect on the transformation time. All transformations are executed in approximately the same time, even though the files with long tag names are bigger. This issue is also mentioned in Section 3.3.1, where the document size is considered as a worse measure for performance than the number of elements.

6.2.4 Keep the output documents small

This idea is based on the assumption that the less output has to be created during the transformation the faster the transformation will be executed.

One sample scenario is the transformation of XML to HTML. Instead of enriching the HTML output with a stylish markup it is better to use CSS. That way the transformation on the server side is simplified and the client takes up parts of the load, because the CSS is applied by the browser on the client side.

6.3 Modifications of XSLT stylesheets

The second approach to improve the performance of XSL transformations is to modify the XSLT stylesheet. Out of the different ways to realize a transformation the best performing one has to be found.

The following section introduces some alternative ways of coding XSLT. The produced result is always the same. The goal is to find some rules that help XSLT developers to avoid performance pitfalls in their transformations.

6.3.1 Prefer “pattern matching” and “selecting” over “filtering”

Sal Mangano discusses the question for the best way to select nodes in his book [Man03]. The alternatives are “pattern matching”, “selecting” and “filtering” nodes. Mangano’s recommendation is to prefer pattern matching and selecting over filtering.

The following test executes a transformation in the three different ways. The XSLT stylesheet selects all the `<country>` elements with the `<name>` '11' and outputs their `<capital>` element. The stylesheet *country_filtering.xsl* looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

<xsl:output method="text"/>

<xsl:template match="/">
   <xsl:apply-templates select="Source"/>
</xsl:template>

<xsl:template match="Source">
   <xsl:for-each select="country">
      <xsl:if test="@name='11'">
         <xsl:value-of select="@capital"/>
      </xsl:if>
   </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

An option to this filtering method is using the '`match`' attribute of the `<xsl:template>` tag. The following code snippet shows the difference in the *country_matching.xsl* stylesheet:

```
...
<xsl:template match="/">
   <xsl:apply-templates select="Source/country"/>
</xsl:template>

<xsl:template match="country[@name='11']">
   <xsl:value-of select="@capital"/>
</xsl:template>
...
```

The third option is to use a for-each-loop together with a selection function. The stylesheet *country_selecting.xsl* looks like this:

```
...
<xsl:template match="/">
   <xsl:apply-templates select="Source"/>
</xsl:template>

<xsl:template match="Source">
   <xsl:for-each select="country[@name='11']">
      <xsl:value-of select="@capital"/>
   </xsl:for-each>
</xsl:template>
...
```

The output data that is generated with the three stylesheets is the same. The transformation time is shown in Figure 6-9, 6-10 and 6-11.

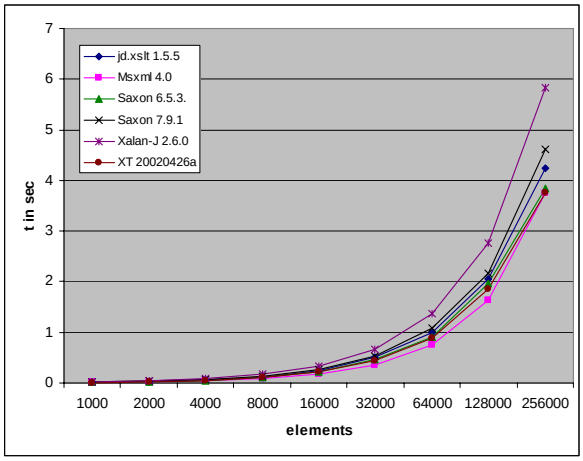

Figure 6-9 Results country_filtering.xsl

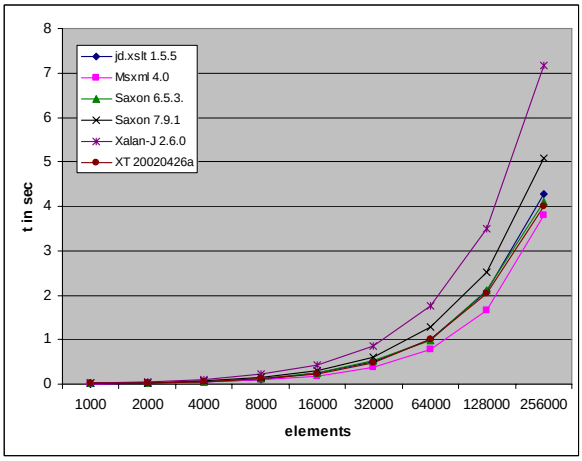

Figure 6-10 Results country_matching.xsl

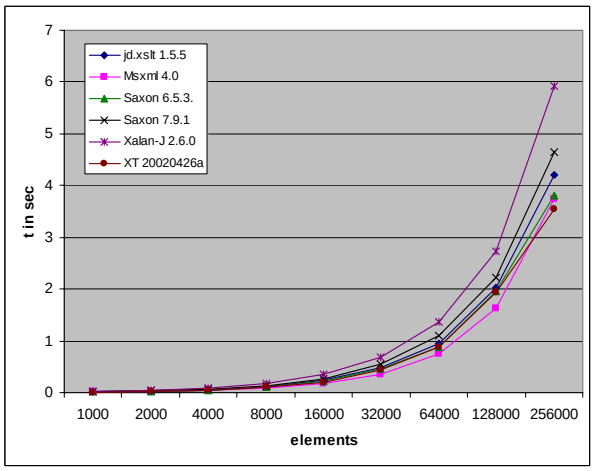

Figure 6-11 Results country_selecting.xsl

Obviously there are hardly differences between the three stylesheets for this test case. Sal Mangano mentioned in his book that the differences between these three stylesheets might vanish over time. Obviously this already happened since he executed his test. Ideally the XSLT developers do not have to worry about those differences in coding style. They write XSLT in their preferred way. The XSLT processor internally executes the intended operation in the fastest possible way.

6.3.2 Use the Muenchian method for grouping

Grouping elements is a transformation that is often needed for structural modifications of XML documents. The Clio transformation in section 5.3.4 is an example for a grouping operation.

There are different ways to solve the grouping problem. The Muenchian method is one of them. It is named after Steve Muench, an XSLT expert who works for Oracle. The idea is to use the `<xsl:key>` tag and the *key()* function to address the elements that have to be grouped. The concept of the Muenchian grouping is explained in detail on Jenni Tennison's homepage [mgrp].

The following stylesheet *grouping_muench.xsl* realizes the Muenchian grouping for the same input file that is used in Section 5.3.4:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes"/>

<xsl:key name="continent_key" match="/Source/country/continent_name"
                                                              use="."/>
<xsl:template match="/">
<continents>
   <xsl:for-each select="/Source/country/continent_name[generate-
                           id(.)=generate-id(key('continent_key',.))]">
   <xsl:variable name="cont_name" select="."/>
   <continent>
      <xsl:copy-of select="$cont_name"/>
      <country>
         <xsl:for-each select="/Source/country[continent_name =
                                              $cont_name]/country_name">
            <xsl:copy-of select="."/>
         </xsl:for-each>
      </country>
   </continent>
   </xsl:for-each>
</continents>
</xsl:template>

</xsl:stylesheet>
```

Instead of using keys the same output can be produced by using the preceding-sibling axis. This is done in the *grouping_normal.xsl* stylesheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">
<continents>
   <xsl:for-each select="/Source/country[not(preceding-
     sibling::country/continent_name=./continent_name)]/continent_name">
   <xsl:variable name="cont_name" select="."/>
   <continent>
      <xsl:copy-of select="$cont_name"/>
      <country>
         <xsl:for-each select="/Source/country[continent_name =
                                              $cont_name]/country_name">
            <xsl:copy-of select="."/>
         </xsl:for-each>
      </country>
   </continent>
   </xsl:for-each>
</continents>
</xsl:template>
```

```
</xsl:stylesheet>
```

The execution time of the *grouping_normal.xsl* stylesheet is depicted in Figure 6-12. It is a very time consumptive operation that runs up to more than half an hour. The *grouping_muench.xsl* transformation (Figure 6-13) is finished much faster.

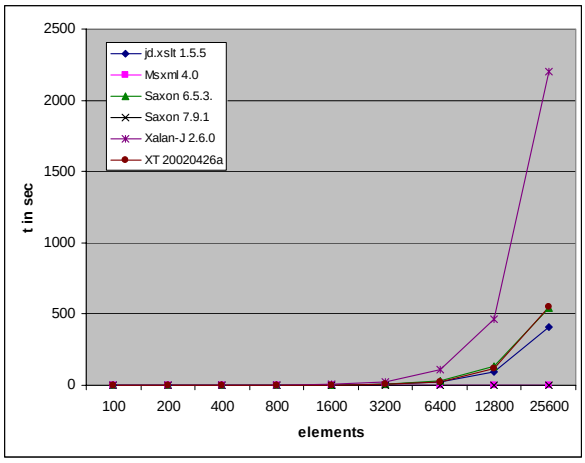

Figure 6-12 Results grouping_normal.xsl

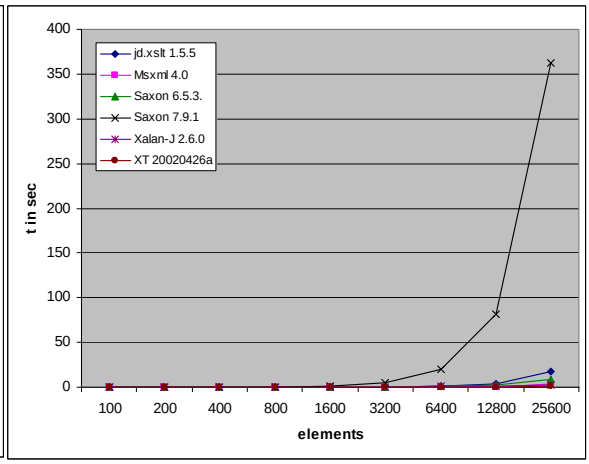

Figure 6-13 Results grouping_muench.xsl

The following table shows the measured differences of the transformation time for the input file with 25,600 <country> elements (2 MB):

	jd.xslt	Msxml	Saxon 6.5.3.	Saxon 7.9.1	Xalan-J	XT
normal	407.626	0.55	544.753	1.522	2197.71	553.906
muenchian	17.675	3.586	9.213	362	2.815	1.122

The behavior of the transformation time is very peculiar. For most of the XSLT processors the normal transformation consumes a lot of time, except from Msxml and Saxon 7.9.1. These two processors finish the same operation within a fraction of the time that the other processors need.

The Muenchian grouping is completed much faster than the ordinary grouping for most of the XSLT processors. The usage of the preceding-sibling axis might be a reason for this. In order to find `<continent_name>` elements that did not occur in the document before all the preceding siblings are checked. This is a very expensive operation.

However, the behavior of Msxml and Saxon 7.9.1 is different. Msxml has a worse performance when executing the Muenchian method. Obviously Microsoft found a fast way to implement the ordinary grouping. Maybe Msxml internally already creates something like the key index. Explicitly creating such an index in the *grouping_muench.xsl* stylesheet consumes extra time.

Interestingly Saxon 7.9.1 needs much more time to complete the Muenchian grouping. The behavior is completely opposite of Saxon 6.5.3. This is a good example for the complexity of XSLT performance. Measuring the transformation time of different XSLT stylesheets also needs to be done with different processors, because the behavior could be completely different.

6.3.3 Usage of keys

Section 6.3.2 showed that the usage of keys can result in performance benefits for the transformation. In order to check whether additional usage of the same key results in further performance improvement, the key is used one more time in the highlighted part of the following stylesheet *grouping_muench2.xsl*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes"/>

<xsl:key name="continent_key" match="/Source/country/continent_name"
                                                                  use="."/>
<xsl:template match="/">
<continents>
   <xsl:for-each select="/Source/country/continent_name[generate-
                           id(.)=generate-id(key('continent_key',.))]">
   <xsl:variable name="cont_name" select="."/>
   <continent>
      <xsl:copy-of select="$cont_name"/>
      <country>
         <xsl:for-each select="key('continent_key',.)">
            <xsl:copy-of select="../country_name"/>
         </xsl:for-each>
      </country>
   </continent>
   </xsl:for-each>
</continents>
</xsl:template>

</xsl:stylesheet>
```

The difference between *grouping_muench.xsl* and *grouping_muench2.xsl* is very small. The additional usage of the key function did not result in further reduction of the transformation time.

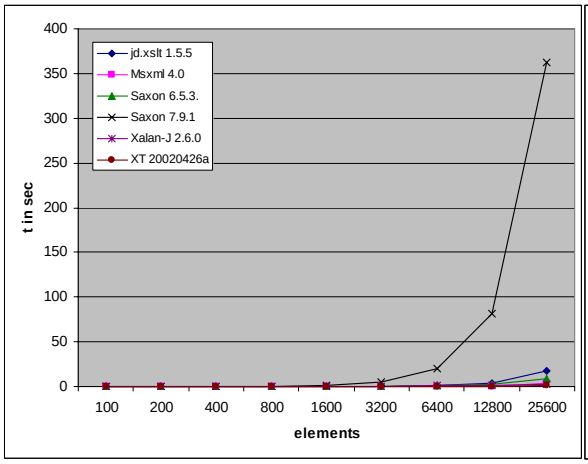

Figure 6-14 Results grouping_muench.xsl

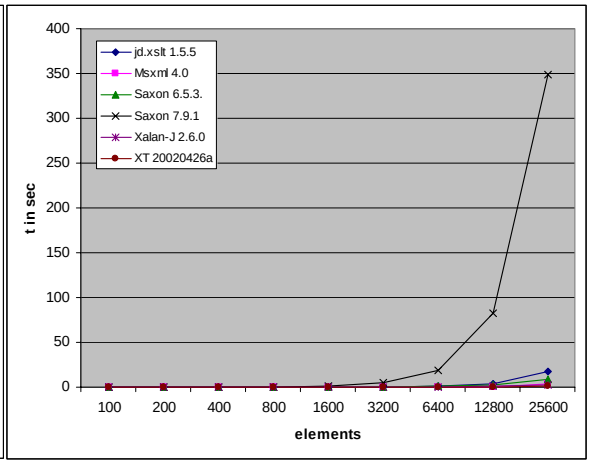

Figure 6-15 Results grouping_muench2.xsl

6.3.4 Prefer the direct addressing of nodes over indirectly addressing them

In order to process the data of an element node it has to be accessed. This is done by addressing it with an XPath expression. XPath offers two ways to do that. One is the directly addressing the node with its complete path, e.g. `/Source/country/continent_name`. The other one is indirectly addressing it e.g. by using `//continent_name`. This XPath expression looks for all the `<continent_name>` elements within an XML file while the first one only looks exactly in the given path.

The stylesheet *grouping_muench3.xsl* replaces the direct addressing in the `<xsl:key>` tag:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes"/>

<xsl:key name="continent_key" match="//continent_name" use="."/>

<xsl:template match="/">
<continents>
   <xsl:for-each select="/Source/country/continent_name[generate-
                          id(.)=generate-id(key('continent_key',.))]">
   <xsl:variable name="cont_name" select="."/>
   <continent>
      <xsl:copy-of select="$cont_name"/>
      <country>
         <xsl:for-each select="key('continent_key',.)">
            <xsl:copy-of select="../country_name"/>
         </xsl:for-each>
      </country>
   </continent>
   </xsl:for-each>
</continents>
</xsl:template>
```

```
</xsl:stylesheet>
```

In addition the stylesheet *grouping_muench4.xsl* replaces the direct addressing in the for-each-loop:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes"/>

<xsl:key name="continent_key" match="//continent_name" use="."/>

<xsl:template match="/">
<continents>
   <xsl:for-each select="//continent_name[generate-id(.)=generate-
                                          id(key('continent_key',.))]">
   <xsl:variable name="cont_name" select="."/>
   <continent>
      <xsl:copy-of select="$cont_name"/>
      <country>
         <xsl:for-each select="key('continent_key',.)">
            <xsl:copy-of select="../country_name"/>
         </xsl:for-each>
      </country>
   </continent>
   </xsl:for-each>
</continents>
</xsl:template>

</xsl:stylesheet>
```

The performance of the two modified scripts is shown in Figure 6-16 and Figure 6-17:

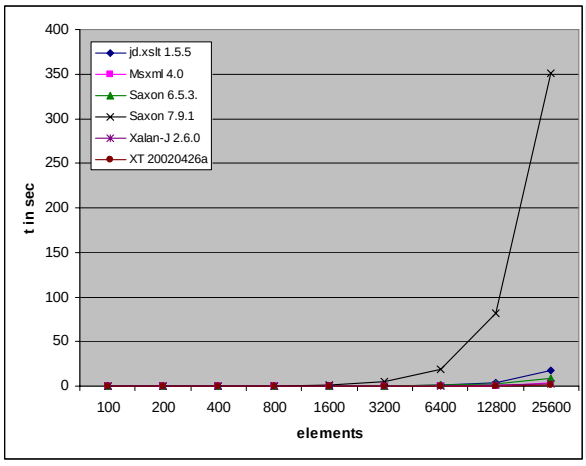

Figure 6-16 Results grouping_muench3.xsl

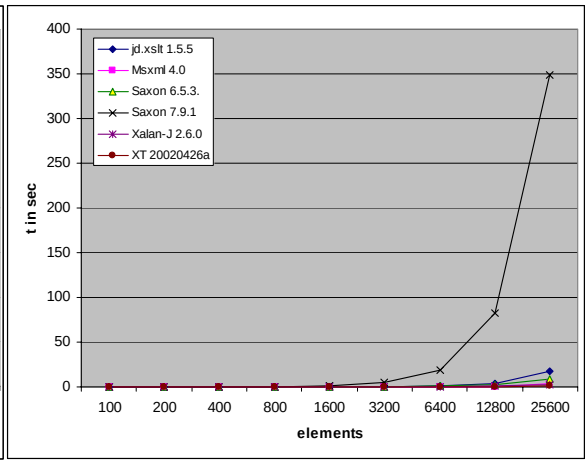

Figure 6-17 Results grouping_muench4.xsl

The impact of both modifications is very little. For this example the usage of directly addressing the nodes has hardly performance benefits. However, with different input data

this could be different. It is important to notice that dependent on the input document indirectly addressing nodes is more flexible and selects more elements. Hence it could produce a different result than the direct addressing.

Sometimes the XSLT developers need the flexibility of indirectly addressing nodes. Its usage also simplifies the stylesheets. If performance is not critical the advantage of higher readability makes up disadvantages of the transformation speed.

6.3.5 Effects of comments

Comments are important for XSLT developers to increase the readability of the code. The question is whether the usage of comments impacts the transformation time.

For the test every second line of the input file from the transformation in Section 6.2.2 is enhanced with comments. This triples the file size while the structure of the document stays the same. The results of the tests are shown in Figure 6-18 and Figure 6-19:

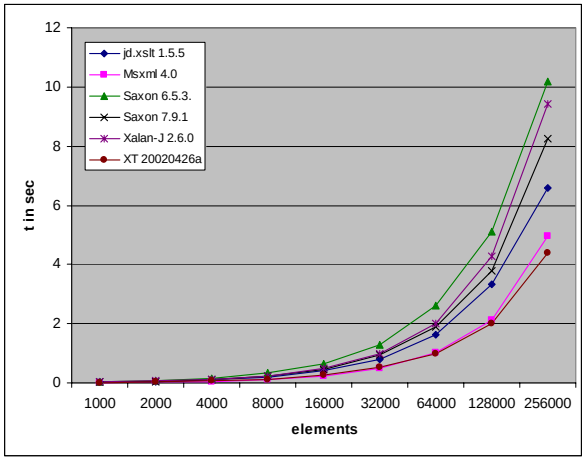

Figure 6-18 Results elementcontent_comment.xsl

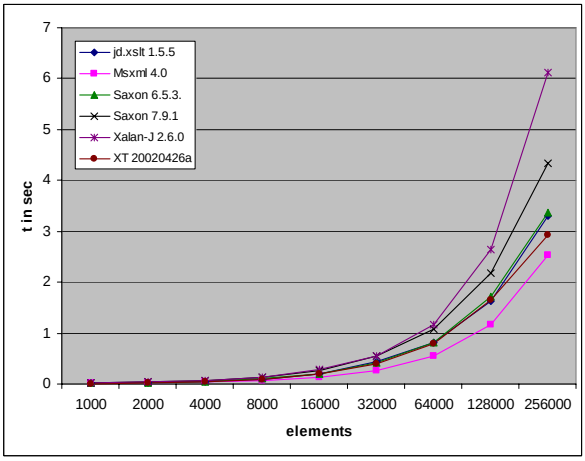

Figure 6-19 Results elementcontent.xsl

Obviously extensive usage of comments has a negative effect on the performance of the XSL transformation. Developers have to find a compromise between good readability and maximized performance.

6.3.6 Split up complex transformations into several stages

In order to reduce the complexity of a transformation the split of this transformation into several stages is an alternative. This idea is also implemented in Clio. The transformation process is divided into two stages because of performance benefits.

One downside of this approach is saving and multiply parsing the generated intermediate results. If possible, the intermediate results should be kept in memory and the transformation should be pipelined – the output of one transformation is the direct input of the next transformation.

6.3.7 Usage of the JAXP API

When transforming many source documents, especially if they use the same stylesheet, it is a good idea to control the process using the JAXP API rather than writing a script that runs each transformation from the command line. The initialisation time for getting the Java VM running and loading all the classes can dominate the actual stylesheet execution cost.

6.4 Modifications of XSLT processor

For the implementers of XSLT processors there are various ways to improve performance. Huge differences of the processing time throughout many tests in this paper made obvious that most of the processors still have a high potential for optimizations. Some ideas[5] for improvements are listed here:

- parse the XSLT stylesheet just once / compile it and then use it multiple times if working with document collections
- when producing multiple output files keep the input file in memory for the next transformation
- keep the XSLT processor and the JVM loaded in memory between runs
- avoid executing the same transformations multiple times – instead store the result
- validate every source document only once
- use a faster XML parser to parse input document and XSLT stylesheet
- consider performing transformation on the client side

The ultimate goal is a behavior similar to database system optimizers. Their task is to find the fastest way to execute a database operation. The XSLT optimizers have to execute XSLT code in the fastest possible way. The XSLT developer would not have to worry about different ways of coding XSLT to realize the intended functionality. The XSLT processor would internally pick the fastest implementation.

6.5 DataPower Hardware XSLT processor

DataPower, the developer of the XSLTMark, takes a different approach to speed up XSLT transformations. They implement an XSLT engine into a hardware device. The *XA 35 XML Accelerator* compiles the operations described in a stylesheet directly into CPU instructions. This machine code can be executed about ten times faster than software approaches [xa35]. Figure 7-18 shows a comparison of the XA35 with software XSLT processors:

[5] partly derived from [dpaw]

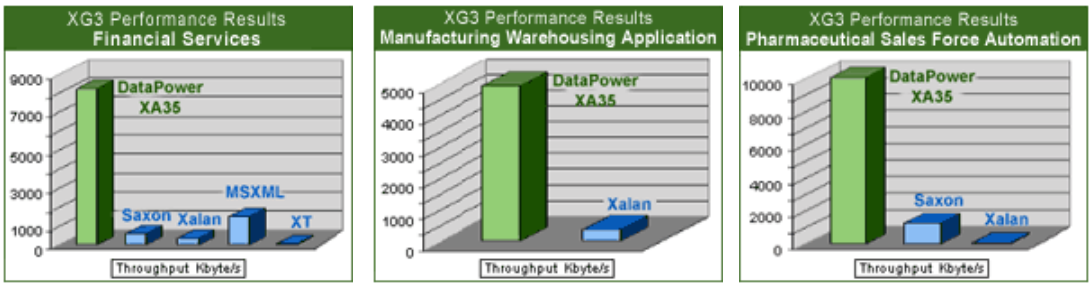

Figure 6-20 DataPower Benchmark Results of XA35 [xa35]

The XA 35 supports the Java API for XML Processing (JAXP). The application server can send requests for processing to the XA 35 using this API. Hence the hardware XSLT engine can easily be integrated into existing applications.

7 Conclusion and Outlook

The evaluation of XSLT performance is based on the results from the various comparison test scenarios ran on the three major factors: XSLT processor, stylesheet, and input data. The results help to interpret the relationship between the three factors and how they interactively affect the transformation speed.

The benchmark tests showed that mostly the sequence of the best performing XSLT processors is the same. However, for some cases, differences between two stylesheets only became obvious for some XSLT processors. When performing XSLT benchmark tests it is important to keep all impacting factors in mind.

With the support of XSLT benchmark software like the Sarvega Benchmark or CatchXSL developers can test their own scripts and see how they can improve the performance of their own XSL transformations. If performance is a crucial factor for a transformation it is a good idea to run it with real data.

The presented ideas provide a reference for developers to improve XSLT performance. The test variations covered throughout the paper are a subset of possible transformations. More simulation models can be developed and executed to get a wider coverage and obtain a more detailed overview of XSLT performance improvements.

Apart from that the improvements of XSLT processors could be investigated in further detail according to the suggestions made in Section 6.4. The tests already showed that there is still a lot of potential to improve the performance of the XSLT processors. The implementers could take suggestions into consideration for their development work.

8 Bibliography

[GR02] John R. Gardner; Zarella L. Rendon: *XSLT & XPATH – A Guide to XML Transformations*. Prentice Hall, Upper Saddle River, NJ, 2002.

[CCD01] Kurt Cagle; Michael Corning; Jason Diamond; Teun Duynstee; Oli G. Gudmundsson; Michael Mason; Jonathan Pinnock; Paul Spencer; Jeff Tang; Andrew Watt; Jirka Jirat; Paul Tchistopolskii, Jeni Tennison: *Professional XSL*. Wrox Press, Birmingham, 2001.

[Man03] Sal Mangano: *XSLT Cookbook*. O'Reilly, Sebastopol, CA, 2003.

[Kay00] Michael Kay: *XSLT Programmer's Reference*. Wrox Press, Birmingham, 2000.

[Ten00] Jenni Tennison: *Beginning XSLT*, Wrox Press, Birmingham, 2000.

[4xsl] 4suite.org

[alto] www.altova.com/resources_xsltengine.html

[ecub] www.ecube.de

[exsl] www.exslt.org/

[dpaw] www.dpawson.co.uk/xsl/sect4/N9883.html#d11386e254

[catx] www.xslprofiler.org/overview.html

[jdxs] www.aztecrider.com/xslt

[jten] www.jenitennison.com/xslt/performance.html

[fast] www.geocities.com/fastxml

[libx] xmlsoft.org/XSLT

[lxsl] www.alphaworks.ibm.com/tech/LotusXSL

[mgrp] www.jenitennison.com/xslt/grouping/muenchian.html

[msxm] msdn.microsoft.com

[mulb] lists.mulberrytech.com/xsl-list

[oxdk] otn.oracle.com/tech/xml

[sabl] www.gingerall.com

[sard] www.sarvega.com/xslt-benchmark.php

[sarp] www.sarvega.com/product-literature.php?xslt=1

[sarv] www.sarvega.com

[saxn] saxon.sourceforge.net

[shap] www.navdeeps.com/shakespeare

[toxg] www.cs.toronto.edu/tox/toxgene

[toxx] www.cs.toronto.edu/tox

[trad] www.uspto.gov/web/offices/ac/ido/oeip/sgml/st32/trademark/TDXFDTDs.html

[xa35] www.datapower.com/products/xa35

[xalc] xml.apache.org/xalan-c

[xalj] xml.apache.org/xalan-j

[xalm] www.geocities.com/jbenhill/xalam.html

[xmar] www.datapower.com/xmldev/xsltmark.html

[xmlb] xmlbench.sourceforge.net

[xmlt] xmlxslt.sourceforge.net

[xt] www.blnz.com/xt

[ANZ01] Aboulnaga, Ashraf; Naughton, Jeffrey F.; Zhang, Chun. Generating synthetic complex-structured XML data. In *Proceedings of the Fourth International Workshop on the Web and Databases*, pages 79–84, Santa Barbara, CA, USA, 2001

[BMK02] Barbosa, Denilson; Mendelzon, Alberto O.; Keenleyside, John; Lyons, Kelly. ToXgene: An extensible template-based data generator for XML, In *SIGMOD Conference,* 2002

[HMH01] Hernández, Mauricio A.; Miller, Renee J.; Haas, Laura M.; Yan, Lingling; Ho, Howard C. T.; Tian, Xuqing. Clio: A Semi-Automatic Tool For Schema Mapping, In *Proceedings of the ACM SIGMOD Conference*, 2001

[MHH00] Miller, Renee J.; Haas, Laura M.; Hernández, Mauricio A. Schema Mapping as Query Discovery, In *VLDB 2000*, Cairo, Egypt 2000

[MHH01] Miller, Renee J.; Hernández, Mauricio A.; Haas, Laura M.; Yan, Lingling; Ho, Howard C. T.; Fagin, Ronald; Popa, Lucian. The Clio Project: Managing Heterogeneity, In *ACM SIGMOD Record*, 2001

[NHT01] Naumann, Felix; Ho, Howard C. T.; Tian, Xuqing; Haas, Laura M.; Megiddo, Nimrod. Attribute Classification Using Feature Analysis, 2001

[PHV02] Popa, Lucian; Hernández, Mauricio A.; Velegrakis, Yannis; Miller, Renee J.; Naumann, Felix; Ho, Howard C. T. Mapping XML and Relational Schemas with Clio, 2002

[SWK01] Schmidt, Albrecht; Waas, Florian; Kersten, Martin; Carey, Michael; Manolescu, Iona; Busse, Ralph. The XML Benchmark project, In *CWI*, 2001

[SWK02] Schmidt, Albrecht; Waas, Florian; Kersten, Martin; Carey, Michael; Manolescu, Iona; Busse, Ralph. XMark – A Benchmark for XML Data Management, 2002

[YMH01] Yan, Lingling; Miller, Renee J.; Haas, Laura M.; Fagin, Ronald. Data-Driven Understanding and Refinement of Schema Mappings, 2001

9 Appendix –Test Results

This section contains detailed test results. The tests were executed with jd.xslt 1.5.5, Msxml 4.0, Saxon 6.5.3, Saxon 7.9.1, Xalan-J 2.6.0 and XT 20020426a.

9.1 Clio Results

9.1.1 Transformation attributes to elements

Stylesheet: a2e1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.018	0.012	0.04	0.042	0.056	0.018
2,000	0.038	0.026	0.082	0.0822	0.1022	0.034
4,000	0.0842	0.048	0.1722	0.1724	0.2182	0.074
8,000	0.1762	0.0982	0.3546	0.3486	0.4926	0.1502
16,000	0.3566	0.2022	0.683	0.711	1.2018	0.2964

Stylesheet: a2e2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.0342	0.028	0.07	0.0722	0.088	0.036
2,000	0.0782	0.06	0.1662	0.1502	0.1862	0.078
4,000	0.1622	0.118	0.4548	0.3424	0.3864	0.1762
8,000	0.3424	0.2402	1.386	0.9172	0.8512	0.3444
16,000	0.677	0.4928	4.6026	2.764	1.9928	0.681

Stylesheet: a2emu1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.02	0.014	0.0462	0.044	0.058	0.022
2,000	0.04	0.028	0.0922	0.0902	0.1162	0.0402
4,000	0.0922	0.056	0.1802	0.1842	0.2384	0.0862
8,000	0.1862	0.106	0.3506	0.3664	0.579	0.1682
16,000	0.3846	0.2184	0.7652	0.7612	1.258	0.3284

Stylesheet: a2emu2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.0642	0.052	0.1102	0.1242	0.1924	0.0682
2,000	0.1462	0.104	0.2404	0.2464	0.3826	0.1542
4,000	0.2904	0.2122	0.5988	0.5528	0.7732	0.2904
8,000	0.5708	0.4446	1.6924	1.3158	1.6202	0.6308
16,000	1.1716	0.9432	5.1474	3.5052	3.527	1.1838

9.1.2 Transformation elements to attributes

Stylesheet: e2a1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.0182	0.014	0.038	0.0382	0.05	0.018
2,000	0.038	0.028	0.0742	0.082	0.1042	0.036
4,000	0.0782	0.0502	0.1522	0.1602	0.2024	0.0782
8,000	0.1636	0.1022	0.3006	0.327	0.4103	0.15
16,000	0.3455	0.2104	0.5905	0.611	0.8315	0.3005

Stylesheet: e2a2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.03	0.022	0.0522	0.062	0.0982	0.034
2,000	0.0682	0.0440	0.114	0.1282	0.2124	0.0682
4,000	0.1262	0.0902	0.2744	0.2864	0.4386	0.1482
8,000	0.2804	0.1824	0.713	0.687	1.0636	0.2884
16,000	0.5573	0.377	2.043	2.043	3.061	0.5543

Stylesheet: e2amu1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.022	0.014	0.044	0.0462	0.056	0.02
2,000	0.0482	0.026	0.08	0.0822	0.1062	0.04
4,000	0.09	0.054	0.1622	0.1702	0.2164	0.086
8,000	0.1844	0.1102	0.3324	0.3324	0.4346	0.1782
16,000	0.3926	0.2242	0.6370	0.707	0.8872	0.3346

Stylesheet: e2amu2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.052	0.042	0.0942	0.102	0.2064	0.0622
2,000	0.1082	0.0862	0.1862	0.2064	0.4646	0.1322
4,000	0.2222	0.1702	0.4748	0.4346	0.8552	0.2502
8,000	0.4406	0.3524	0.9654	0.9834	1.9168	0.4788
16,000	0.9194	0.713	2.4634	2.5476	4.825	0.9654

9.1.3 Flat hierarchy to flat hierarchy

Stylesheet: f2f1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.0082	0.002	0.014	0.014	0.0182	0.008
200	0.0122	0.008	0.0362	0.028	0.0382	0.012
400	0.0162	0.0162	0.0382	0.036	0.048	0.018
800	0.03	0.0202	0.072	0.074	0.0962	0.032
1,600	0.072	0.0422	0.1362	0.1682	0.1862	0.066

Stylesheet: f2f2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.016	0.004	0.0282	0.03	0.04	0.018
200	0.02	0.012	0.034	0.046	0.0502	0.0182
400	0.022	0.0202	0.0462	0.046	0.056	0.022
800	0.048	0.0322	0.072	0.0802	0.0942	0.038
1,600	0.0882	0.066	0.1642	0.1722	0.1902	0.0862

Stylesheet: f2fmu1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.01	0.002	0.016	0.016	0.02	0.0102
2,000	0.01	0.008	0.0262	0.026	0.0342	0.01
4,000	0.018	0.014	0.036	0.0382	0.05	0.018
8,000	0.038	0.024	0.0722	0.0762	0.0942	0.034
16,000	0.0782	0.042	0.1402	0.1582	0.1882	0.074

Stylesheet: f2fmu2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.018	0.006	0.0382	0.04	0.0582	0.018
2,000	0.024	0.016	0.04	0.048	0.0662	0.024
4,000	0.036	0.024	0.0442	0.05	0.0782	0.03
8,000	0.0682	0.046	0.0882	0.088	0.1402	0.066
16,000	0.1362	0.0882	0.2042	0.1642	0.2564	0.1262

9.1.4 Flat hierarchy to nested hierarchy

Two levels of nesting

Stylesheet: f2n2l1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.07	0.03	0.15	0.16	0.301	0.09
200	0.23	0.13	0.521	0.561	1.112	0.33
400	0.782	0.531	2.003	2.093	4.286	1.242
800	3.015	2.043	7.801	7.992	17.185	4.756
1,600	13.339	8.192	30.764	32.206	69.069	18.546

Stylesheet: f2n2l2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
3,254	0.061	0.03	0.071	0.08	0.15	0.04
11,394	0.15	0.09	0.2	0.211	0.42	0.18
46,162	0.58	0.391	0.801	0.61	1.773	0.531
179,382	2.283	1.642	3.966	2.213	12.087	2.144
719,798	10.215	9.063	23.584	8.252	143.636	9.254

Stylesheet: f2n2lmu1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.008	0.0022	0.018	0.0222	0.026	0.01
200	0.018	0.01	0.034	0.038	0.0502	0.0162
400	0.028	0.018	0.06	0.0702	0.086	0.0242
800	0.058	0.026	0.1222	0.1422	0.1722	0.052
1,600	0.1262	0.0562	0.2422	0.2864	0.3504	0.1062

Stylesheet: f2n2lmu2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
300	0.026	0.018	0.032	0.034	0.08	0.028
600	0.058	0.0342	0.06	0.0662	0.1502	0.056
1,200	0.1262	0.0662	0.1302	0.1262	0.2924	0.1202
2,400	0.3164	0.1422	0.2844	0.2624	0.585	0.2284
4,800	0.8492	0.2924	0.709	0.5968	1.1438	0.4586

Three levels of nesting

Stylesheet: f2n3l1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	1.472	1.011	3.315	3.405	8.503	2.314
200	11.587	7.791	25.757	26.548	65.053	17.455
400	98.742	63.281	209.011	219.505	528.11	146.901

Stylesheet: f2n3l2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
24,726	0.541	0.32	1.743	0.46	1.392	0.461
187,314	4.366	2.413	38.135	3.095	18.296	2.944

Stylesheet: f2n3lmu1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.014	0.004	0.024	0.0322	0.04	0.012
200	0.026	0.014	0.05	0.064	0.0802	0.024
400	0.0482	0.024	0.0882	0.1102	0.1482	0.042

Stylesheet: f2n3lmu2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
400	0.0702	0.0402	0.0782	0.0802	0.2744	0.0842
800	0.1542	0.086	0.1582	0.1602	0.5228	0.1764
1,600	0.3164	0.1882	0.3166	0.3164	1.0134	0.3464

9.1.5 Nested hierarchy to flat hierarchy

Stylesheet: n2f1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.008	0.004	0.016	0.018	0.022	0.006
200	0.012	0.012	0.034	0.028	0.038	0.014
400	0.024	0.018	0.0442	0.0422	0.058	0.02
800	0.046	0.028	0.0842	0.0862	0.1142	0.046
1,600	0.0962	0.06	0.1762	0.1702	0.2322	0.0942

Stylesheet: n2f2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.016	0.004	0.026	0.0342	0.04	0.0142
200	0.02	0.012	0.034	0.046	0.048	0.018
400	0.022	0.02	0.0402	0.042	0.056	0.022
800	0.05	0.03	0.074	0.076	0.1022	0.04
1,600	0.0902	0.062	0.1662	0.1744	0.1982	0.0842

Stylesheet: n2fmu1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.01	0.004	0.018	0.018	0.024	0.008
200	0.012	0.01	0.024	0.024	0.0342	0.012
400	0.024	0.018	0.044	0.0462	0.0662	0.022
800	0.048	0.032	0.086	0.09	0.1242	0.044
1,600	0.1062	0.062	0.1764	0.1762	0.2464	0.0962

Stylesheet: n2fmu2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.0182	0.006	0.0402	0.042	0.054	0.02
200	0.024	0.016	0.04	0.0502	0.066	0.024
400	0.034	0.0262	0.0462	0.0502	0.0822	0.032
800	0.0662	0.0442	0.0862	0.0902	0.1342	0.064
1,600	0.1362	0.088	0.2022	0.1642	0.2524	0.1262

9.1.6 Nested hierarchy to nested hierarchy

Stylesheet: n2n1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	3.045	3.174	6.97	5.758	16.323	4.326
200	21.842	28.04	58.754	46.517	133.692	35.801
400	163.725	201.78	440.634	346.549	1105.61	282.776

Stylesheet: n2n2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
26,610	0.631	0.331	1.803	0.481	1.452	0.511
216,456	5.558	2.854	47.038	3.655	23.1133	4.166
1,611,298	102.798	56.021	n/a	n/a	n/a	49.752

Stylesheet: n2nmu1.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.0133	0.0367	0.04	0.0533	0.0167	0.01
200	0.0267	0.05033	0.06	0.0833	0.0267	0.02
400	0.0567	0.1	0.1167	0.1603	0.0467	0.027
800	0.117	0.1937	0.2337	0.334	0.1033	0.0537
1,600	0.2403	0.4007	0.4707	0.671	0.1937	0.1067

Stylesheet: n2nmu2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
600	0.0702	0.0402	0.0842	0.084	0.2764	0.084
1,200	0.1522	0.086	0.1562	0.1624	0.5168	0.1742
2,400	0.3084	0.1842	0.3226	0.3124	1.0074	0.3484
4,800	0.6488	0.3706	0.697	0.6488	1.985	0.6608
9,600	1.464	0.7592	1.6524	1.4782	3.9478	1.3238

9.2 Performance Improvement Results

9.2.1 Splitting up big input files

Stylesheet: e2a1.xsl

file	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
elements1.xml	0.383	0.234	0.659	0.683	0.913	0.334
elements2.xml	0.621	0.391	1.130	1.248	1.606	0.587
elements3.xml	0.627	0.395	1.202	1.244	1.584	0.589
elements4.xml	0.691	0.451	1.268	1.404	1.756	0.649
elements5.xml	0.505	0.308	0.867	0.915	1.210	0.441
elements6.xml	0.603	0.377	1.082	1.134	1.534	0.561
elements7.xml	0.799	0.521	1.436	1.550	2.069	0.753
elements8.xml	0.557	0.339	1.036	1.029	1.366	0.489
elements9.xml	0.388	0.250	0.713	0.779	0.995	0.358
elements10.xml	0.186	0.118	0.342	0.379	0.465	0.182
Sum of files	5.359	3.383	9.734	10.365	13.500	4.943
complete.xml	6.8658	3.8354	9.9344	9.9264	16.043	4.9692

Stylesheet: e2a2.xsl

file	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
intermediate1.xml	0.708	0.437	2.534	2.480	3.872	0.638
intermediate 2.xml	1.135	0.751	5.889	5.872	9.891	1.115
intermediate 3.xml	1.122	0.758	6.029	5.561	10.368	1.115
intermediate 4.xml	1.299	0.875	7.411	7.110	12.568	1.222
intermediate 5.xml	0.868	0.584	3.929	3.886	6.389	0.911
intermediate 6.xml	1.072	0.728	5.638	5.338	9.584	1.111
intermediate 7.xml	1.455	1.041	9.427	8.853	17.315	1.425
intermediate 8.xml	1.031	0.651	4.737	4.513	7.591	0.975
intermediate 9.xml	0.754	0.471	2.731	2.691	4.456	0.691
intermediate 10.xml	0.351	0.217	0.915	0.875	1.369	0.347
Sum of files	9.794	6.512	49.238	47.178	83.403	9.551
complete.xml	12.077	8.062	349.783	317.547	706.886	10.175

9.2.2 Using attributes instead of elements

Stylesheet: elementcontent.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.012	0.008	0.016	0.020	0.020	0.012
2,000	0.028	0.020	0.028	0.036	0.038	0.022
4,000	0.048	0.032	0.052	0.066	0.072	0.048
8,000	0.094	0.070	0.102	0.136	0.140	0.098
16,000	0.208	0.134	0.216	0.270	0.286	0.196
32,000	0.445	0.278	0.413	0.569	0.581	0.405
64,000	0.885	0.584	0.858	1.112	1.238	0.785
128,000	1.772	1.178	1.829	2.270	2.741	1.729
256,000	3.385	2.584	3.480	4.552	6.625	3.275

Stylesheet: attributecontent.xml

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.014	0.014	0.014	0.018	0.024	0.01
2,000	0.022	0.028	0.0282	0.0362	0.0422	0.022
4,000	0.05	0.0402	0.052	0.074	0.0922	0.046
8,000	0.0982	0.0802	0.1022	0.1564	0.2264	0.0942
16,000	0.2262	0.1602	0.2162	0.3206	0.671	0.1942
32,000	0.4846	0.3324	0.4486	0.635	2.6056	0.4106
64,000	0.9552	0.701	0.8892	1.346	10.7174	0.8172
128,000	2.0168	1.5362	1.8206	2.6798	45.4054	1.6926
256,000	4.0938	3.531	3.6894	5.4218	178.082	3.5792

9.2.3 Keep names for elements short

Stylesheet: grouping_muench.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.01	0.0022	0.02	0.0262	0.034	0.008
200	0.016	0.008	0.024	0.042	0.04	0.018
400	0.016	0.014	0.024	0.0722	0.044	0.016
800	0.034	0.02	0.048	0.2644	0.086	0.032
1,600	0.0782	0.04	0.1082	1.0776	0.1744	0.0702

3,200	0.2142	0.0862	0.2824	4.5286	0.3346	0.1522
6,400	0.7992	0.1982	0.8132	19.4378	0.689	0.2924
12,800	4.051	0.5105	2.609	82.058	1.392	0.551
25,600	17.675	3.586	9.213	362	2.815	1.122

Stylesheet: grouping_muench_long.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.01	0.0022	0.02	0.0262	0.034	0.008
200	0.016	0.008	0.024	0.042	0.04	0.018
400	0.016	0.014	0.024	0.0722	0.044	0.016
800	0.034	0.02	0.048	0.2644	0.086	0.032
1,600	0.0782	0.04	0.1082	1.0776	0.1744	0.0702
3,200	0.2142	0.0862	0.2824	4.5286	0.3346	0.1522
6,400	0.7992	0.1982	0.8132	19.4378	0.689	0.2924
12,800	4.051	0.5105	2.609	82.058	1.392	0.551
25,600	17.675	3.586	9.213	362	2.815	1.122

9.2.4 Prefer "pattern matching" and "selecting" over "filtering"

Stylesheet: country_filtering.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.0122	0.014	0.014	0.016	0.026	0.01
2,000	0.028	0.024	0.028	0.0342	0.044	0.0262
4,000	0.056	0.042	0.052	0.066	0.084	0.05
8,000	0.11	0.0842	0.1022	0.1302	0.1662	0.1002
16,000	0.2364	0.1742	0.2202	0.2564	0.3364	0.2122
32,000	0.5086	0.3584	0.4626	0.5348	0.661	0.4426
64,000	0.9894	0.751	0.9072	1.0896	1.362	0.8892
128,000	2.0448	1.6264	1.9548	2.1652	2.758	1.8586
256,000	4.238	3.7634	3.8396	4.6106	5.8264	3.7534

Stylesheet: country_matching.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.012	0.012	0.018	0.018	0.03	0.014
2,000	0.028	0.0242	0.03	0.0362	0.0522	0.0262
4,000	0.0522	0.046	0.058	0.072	0.1082	0.052
8,000	0.1082	0.0902	0.1162	0.1462	0.2164	0.126
16,000	0.2464	0.1782	0.2464	0.2944	0.4246	0.2324
32,000	0.5106	0.3726	0.5168	0.6128	0.8532	0.4868
64,000	0.9974	0.7712	0.9936	1.2798	1.7586	1.0036
128,000	2.077	1.6704	2.103	2.5216	3.499	2.031
256,000	4.2662	3.8054	4.104	5.0752	7.1702	4.0058

Stylesheet: country_selecting.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.012	0.012	0.016	0.02	0.026	0.0102
2,000	0.026	0.026	0.028	0.034	0.046	0.024
4,000	0.048	0.0422	0.052	0.064	0.084	0.0462
8,000	0.1042	0.086	0.1022	0.1302	0.1702	0.1002
16,000	0.2364	0.1744	0.2242	0.2604	0.3424	0.2084
32,000	0.4888	0.3606	0.4586	0.5408	0.675	0.4366
64,000	0.9494	0.753	0.8912	1.0914	1.37	0.8834
128,000	2.025	1.6342	1.959	2.2232	2.73	1.9448
256,000	4.194	3.7352	3.8034	4.6366	5.9284	3.545

9.2.5 Use the Muenchian method for grouping

Stylesheet: grouping_normal.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.012	0.002	0.022	0.018	0.048	0.014
200	0.028	0.004	0.054	0.0262	0.1182	0.032
400	0.078	0.014	0.1464	0.024	0.4306	0.084
800	0.2924	0.018	0.5348	0.05	1.5342	0.2984
1,600	1.0736	0.032	2.141	0.0962	6.2008	1.0996
3,200	5.0972	0.064	8.5404	0.1924	25.8432	5.506

6,400	22.5484	0.1842	33.9268	0.3846	108.47	26.8968
12,800	95.5925	0.2905	134.8385	0.741	462.41	120.308
25,600	407.626	0.55	544.753	1.522	2197.71	553.906

9.2.6 Usage of keys

Stylesheet: grouping_muench2.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.01	0.012	0.018	0.026	0.034	0.0102
200	0.016	0.0062	0.024	0.042	0.0402	0.018
400	0.012	0.012	0.0202	0.068	0.0362	0.012
800	0.0342	0.016	0.0402	0.2544	0.06	0.022
1,600	0.066	0.0342	0.0922	1.0676	0.1202	0.044
3,200	0.1964	0.074	0.2544	4.4404	0.2384	0.1
6,400	0.763	0.1802	0.7652	19.2178	0.4846	0.2084
12,800	3.8905	0.4805	2.4735	82.108	0.9965	0.3905
25,600	17.986	3.214	8.973	348.861	2.053	0.781

9.2.7 Prefer direct addressing nodes over indirect addressing

Stylesheet: grouping_muench3.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.01	0.002	0.018	0.028	0.034	0.01
200	0.016	0.006	0.022	0.04	0.04	0.014
400	0.012	0.012	0.0202	0.0742	0.034	0.0122
800	0.0282	0.018	0.0422	0.2602	0.062	0.0202
1,600	0.0702	0.034	0.0962	1.0816	0.1202	0.046
3,200	0.1942	0.0742	0.2544	4.4182	0.2364	0.096
6,400	0.7752	0.1784	0.7772	19.2418	0.4808	0.2064
12,800	3.9055	0.481	2.5385	81.998	0.9815	0.3905
25,600	18.226	3.205	8.953	350.494	2.013	0.791

Stylesheet: grouping_muench4.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
100	0.012	0.002	0.018	0.024	0.038	0.012

200	0.014	0.006	0.024	0.042	0.0462	0.014
400	0.014	0.012	0.022	0.0682	0.034	0.012
800	0.0322	0.022	0.048	0.2582	0.0622	0.026
1,600	0.0742	0.036	0.1022	1.0554	0.1162	0.0522
3,200	0.2124	0.0762	0.2644	4.4224	0.2324	0.12
6,400	0.8152	0.1822	0.783	19.0916	0.4686	0.2382
12,800	3.991	0.496	2.559	81.9025	0.9665	0.4605
25,600	18.116	3.195	9.003	349.172	1.983	0.931

9.2.8 Effects of comments

Stylesheet: elementcontent_comment.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.024	0.014	0.04	0.0302	0.0322	0.014
2,000	0.048	0.028	0.0762	0.056	0.062	0.0302
4,000	0.1002	0.056	0.1642	0.116	0.1202	0.058
8,000	0.2002	0.116	0.3304	0.2324	0.2384	0.1202
16,000	0.4286	0.2444	0.6388	0.4608	0.4788	0.2544
32,000	0.809	0.4928	1.296	0.9534	0.9874	0.5146
64,000	1.6344	1.0154	2.6258	1.9048	2.015	0.9954
128,000	3.3388	2.1352	5.0994	3.7794	4.268	2.0188
256,000	6.5834	4.9772	10.1866	8.2438	9.4134	4.3742

Stylesheet: elementcontent.xsl

elements	jd.xslt	Msxml	Saxon 6	Saxon 7	Xalan-J	XT
1,000	0.0122	0.0122	0.016	0.018	0.018	0.01
2,000	0.024	0.0222	0.026	0.036	0.038	0.0222
4,000	0.044	0.036	0.05	0.066	0.0682	0.0442
8,000	0.0902	0.0682	0.1002	0.1322	0.1362	0.0962
16,000	0.1984	0.1302	0.2082	0.2624	0.2764	0.1964
32,000	0.4306	0.2684	0.4146	0.5408	0.5608	0.3966
64,000	0.8112	0.5568	0.823	1.0716	1.1658	0.7952
128,000	1.6324	1.1576	1.7264	2.1732	2.6418	1.6422
256,000	3.3028	2.5218	3.367	4.3342	6.1248	2.9182