Optimization of XML Transformations Using Template Specialization

Ce Dong James Bailey

Department of Computer Science and Software Engineering
The University of Melbourne
{cdong, jbailey}@cs.mu.oz.au

Abstract. XSLT is the primary language for transforming and presenting XML. Effective optimization techniques for XSLT are particularly important for applications which involve high volumes of data, such as online server-side processing. This paper presents a new approach for optimizing XSLT transformations, based on the notion of template specialization. We describe a number of template specialization techniques, suitable for varying XSLT design styles and show how such specializations can be used at run time, according to user input queries. An experimental evaluation of our method is undertaken and it is shown to be particularly effective for cases with very large XML input.

1 Introduction

XML is rapidly becoming the de facto standard for information storage, representation and exchange on the World Wide Web. The eXtensible Stylesheet Language Transformations (XSLT) standard [3, 16] is a primary language for transforming, reorganizing, querying and formatting XML data. In particular, use of server side XSLT [15] is an extremely popular technology for processing and presenting results to user queries issued to a server-side XML database.

Execution of XSLT transformations can be very costly, however, particularly when large volumes of XML are involved and techniques for optimization of such transformations are therefore an important area of research. In this paper, we propose a new method for optimization of XSLT programs, based on the technique of program specialization. The underlying idea is that server-side XSLT programs are often written to be generic and may contain a lot of logic that is not needed for execution of the transformation with reference to given user query inputs. Such inputs are passed as parameters to the XSLT program at run-time, often using forms in a Web browser. For example, a user reading a book represented in XML might pass a parameter to an XSLT program referring to the number of the chapter they wish to see presented (i.e. transformed from XML to HTML by the XSLT). The XSLT program may obviously contain logic which is designed for presenting the contents of other chapters, but it will not be needed for this user query. Given knowledge of the user input space, it is instead possible to automatically (statically) create different specialized versions of the original XSLT program, that can be invoked in preference to the larger, more generic version at run-time. In our book example, specialized XLST programs could be created for each chapter. Since the specialized versions can be much smaller than the original program, important savings in execution time and consequently user response time are possible.

In the paper, we describe methods for i) automatically creating a set of specialized XSLT programs (small XSLTs) based on an original generic (big) XSLT program ii) selecting an appropriate small XSLT at run-time according to a cost analysis of the user input. Our contributions in this paper are:

- An optimization approach for XSLT Transformations based on template specialization.
- Four novel principles that can be used for XSLT template specialization: 1) Branch_Principle, 2)Position_Principle, 3)Kin_Principle, 4)Calling_Principle.

Presentation of experimental results demonstrates the effectiveness of specialization for XSLT. We are not aware of any other work which uses the concept of specialization to improve the performance of XSLT programs.

2 Background

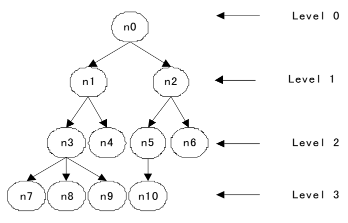
We begin by briefly reviewing some concepts regarding DTDs, (server-side) XSLT and XPath, assuming the reader already has basic knowledge in these areas.

2.1 DTDs and *DTD-Graph*

An XML DTD [2] provides a structural specification for a class of XML documents and is used for validating the correctness of XML data (An example is shown in Fig.1 (a)).

```
<!ELEMENT n0 (n1, n2+)>
<!ELEMENT n1 (n3|n4*)>
<!ELEMENT n2 (n5, n6)>
<!ELEMENT n3 (n7, n8?)>
<!ELEMENT n5 (n10*)>
<!ATTLIST n3 n9 CDATA #REQUIRED>
<!ELEMENT n4 (#PCDATA)>
<!ELEMENT n6 (#PCDATA)>
<!ELEMENT n7 (#PCDATA)>
<!ELEMENT n8 (#PCDATA)>
<!ELEMENT n10 (#PCDATA)>
```

(a) DTD



(b) DTD-Graph

Fig. 1. The DTD and its corresponding *DTD-Graph*

Based on the DTD, we can create a data structure to summarize the hierarchical information within a DTD, called the *DTD-Graph*. It is a rooted, node-labeled graph, where each node represents either an element or an attribute from the DTD and the edges indicate element nesting. We assume that the DTD contains no IDs and IDREFs, and is acyclic. The *DTD-Graph* is similar to the *Dataguide* structure described by Goldman and Widom in 1997[8]. It will be used to explain the *XSLT Template Specialization Principles* in section 3. From the example of the *DTD-graph* in Fig.1.(b), n_0 to n_{10} (except n_9) denote the names of elements which may exist in the XML document and n_9 denotes the name of an attribute of element n_3.

2.2 Templates in XSLT

An XML document can be modeled as a tree. In XSLT, one defines templates (specified using the command *<xsl:template>*) that match a node or a set of nodes in the XML-tree[16], using a *selection pattern* specified by the *match* attribute of the *<xsl:template>* element. We require a matched node to exist in the *DTD-Graph*. The content of the template specifies how that node or set of nodes should be transformed. The body of a template can be considered to contain two kinds of constructs: i) constant strings and ii) *<xsl:apply-templates>*(or *<xsl:for-each>*). We ignore the branch commands *<xsl:if>* and *<xsl:choose>*, since they cannot directly trigger the application of another template. Constant strings can be inline text or generated XSLT statements (e.g. using *<xsl:value-of>*). The XSLT instruction *<xsl:apply-templates>* has critical importance: without any attributes, it "selects all the children of current node in the source tree, and for each one, finds the matching template rule in the stylesheet, and instantiates it"[12]. A *construction pattern* can optionally be specified using the *select* attribute in *<xsl:apply-templates>*, to select the nodes for which the template needs to match. Our specialization methods support XSLT programs that make use of the elements *<xsl:template>, <xsl:apply-templates>, <xsl:for-each>, <xsl:if>, <xsl:choose>, <xsl:value-of>, <xsl:copy-of>, <xsl:param>*. This represents a reasonably powerful and commonly used fragment of the language. For the *<xsl:param>* element, we assume that the parameters corresponding to the user inputs are declared at the top level in the XSLT program.

2.3 Server-Side XSLT

Server-side XSLT is a popular solution for data exchange and querying on the Web. It is often deployed in e-commerce, e-publishing and information services applications. A typical server-side processing model is sketched in Fig.2 below. Transforming the content on the server has advantages such as providing convenience for business logic design and code reuse, cheaper data access and security and smaller client downloads [13]. A problem that can occur in the use of server-side XSLT, is when a generic XSLT program is designed so that it can handle many possible inputs from the user. At run-time, given specific user input values, much of the logic of the generic XSLT program may not be required. Execution of an XSLT program which is larger than required may result in increased run-time. The main theme of this paper is to propose methods for automatically creating smaller XSLT programs, which are specialised to handle specific combinations of user input values. The *user query* is modelled as a set of parameters, each of which corresponds to an element in the DTD. e.g. The input "Chapter1 Section2'' might indicate the user wishes to see the contents of section 2 in Chapter 1 displayed (converted from XML to HTML).

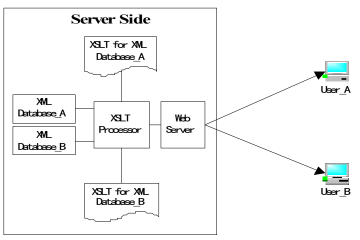


Fig. 2. The Server-Side XSLT model

```
<xsl:stylesheet>             <xsl:stylesheet>
  <t1>                         <t1>
   ...                          ...
   <a>                          <a select>
  </t1>                        </t1>

  <t2/>                        <t2/>
   …                            …
  <tn/>                        <tn/>
</xsl:stylesheet>            </xsl:stylesheet>

(a)Push style XSLT           (b)Pull style XSLT
```

Fig.3. The XSLT design styles

2.4 XSLT design styles

The simplest way to process an XML source tree is to write a template rule for each kind of node that can be encountered, and for that template rule to produce the output required, as well as calling the *<xsl:apply-templates>* instruction to process the children of that node [12]. We call this style of processing *recursive-descent* style or *push* style. A skeleton of a *push* style XSLT is shown in Fig.3.(a). Here, <t> denotes *<xsl:template>* and <a> denotes *<xsl:apply-template>*.

We also can design instructions to explicitly select specific nodes. This makes the instruction more precise about which node to process. This is *pull* style design. The XSLT template instruction *<xsl:apply-templates>* with the value of *select* attribute as the *construction* pattern, is the normal form of a *pull* style processing and *<xsl:for-each>* is also commonly used. A skeleton of *pull* style XSLT is shown in Fig.3.(b). The most general style of XSLT design is the *hybrid* style. It is a combination of *push* and *pull* styles together.

Our specialization methods can be used for any style of XSLT, though our experimental results will refer to these design classifications. An example of a *hybrid* style XSLT program, *XSLBench.xsl*, is given in Fig.4. Looking at Fig.4, the template match for "*FM*" and "*SPEECH*" uses *<xsl:apply-templates>* (*push* style) as the implicit instruction to retrieve child nodes in the source tree and the template match for "*PLAY*" gives an explicit instruction *<xsl:apply-template select="FM|PERSONAE|ACT">* (*pull* style) to retrieve node sets of *FM*, *PERSONAE* and *ACT* in the source tree. An XSLT (program input) parameter is declared by the *<xsl:param>* element with the *name* attribute indicating the name of parameter. This example is from Kevin Jones' XSLBench test suite and it was also used by XSLTMark[4]as a test case for testing the processing of *match* and *select* statements (we have added the parameter declarations to model user input). The corresponding XML source input to *XSLBench.xsl* is a well-known XML document, *shakespeare.xml*[4], consisting of the 37 plays of Shakespeare. These will be used in examples throughout the remainder of the paper.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
  <xsl:output encoding="utf-8"/>
  <xsl:param name="p0"/>
  <xsl:param name="p1"/>
  <xsl:param name="p2"/>
  <xsl:param name="p3"/>
  <xsl:param name="p4"/>
  <xsl:param name="p5"/>
  <xsl:param name="p6"/>
  <xsl:param name="p7"/>
  <xsl:param name="p8"/>
  <xsl:param name="p9"/>
  <xsl:param name="p10"/>
  <xsl:template match="PLAY">
    <html>
      <body>
        <xsl:apply-templates
select="TITLE"/>
        <xsl:apply-templates
select="FM|PERSONAE|ACT"/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="TITLE">
    <xsl:if test="$p0">
      <font style="ITALIC" color="RED">
        <xsl:value-of select="TITLE"/>
      </font>
    </xsl:if>
  </xsl:template>
  <xsl:template match="FM">
    <xsl:if test="$p1">
      <i>
        <xsl:apply-templates/>
      </i>
    </xsl:if>
  </xsl:template>
  <xsl:template match="PERSONAE">
  <xsl:if test="$p2">
    <h2>
      Parts - <xsl:value-of select="TITLE"/>
    </h2>
    <xsl:apply-templates select=".//
PERSONA"/>
  </xsl:if>
</xsl:template>
<xsl:template match="PERSONA">
    <xsl:if test="$p3">
      <p><b><i>
        <xsl:value-of select="."/>
      </i></b></p>
    </xsl:if>
  </xsl:template>
  <xsl:template match="ACT">
    <xsl:if test="$p4">
      <h3>
        <xsl:value-of select="TITLE"/>
      </h3>
      <xsl:apply-templates select="SCENE"/>
    </xsl:if>
  </xsl:template>
  <xsl:template match="SCENE">
    <xsl:if test="$p5">
      <h3>
        <xsl:value-of select="TITLE"/>
      </h3>
      <xsl:apply-templates select="SPEECH"/>
    </xsl:if>
  </xsl:template>
  <xsl:template match="SPEECH">
    <xsl:if test="$p6">
      <xsl:apply-templates/>
    </xsl:if>
  </xsl:template>

  <xsl:template match="SPEAKER">
    <xsl:if test="$p7">
      <p><b>
        <xsl:value-of select="."/>
      </b></p>
    </xsl:if>
  </xsl:template>
  <xsl:template match="LINE">
    <xsl:if test="$p8">
      <xsl:value-of select="."/>
      <br/>
    </xsl:if>
  </xsl:template>
  <xsl:template match="STAGEDIR">
    <xsl:if test="$p9">
      <xsl:value-of select="."/>
      <br/>
    </xsl:if>
  </xsl:template>
  <xsl:template match="SUBHEAD">
    <xsl:if test="$p10">
      <xsl:value-of select="."/>
      <br/>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Fig. 4. The *Hybrid* style XSLT Stylesheet *XSLTBench.xsl*

2.5 XPath Expressions

XPath is a language for locating nodes in an XML document using path expressions. In XSLT, *selection patterns* [15, 17] are specified using a *subset* of XPath and can be used in the *match* attribute of *<xsl:template>* elements. *Construction patterns* are specified using the *full* XPath language and can be used in the *select* attribute of the elements *<xsl:apply-templates>*, *<xsl:for-each>* and *<xsl:value-of>*. XPath also defines a library of standard *functions* for working with *strings*, *numbers* and *boolean* expressions. The expressions enclosed in '[' and ']' in an XPath expression are called *qualifiers*. In this paper, we disallow the use of functions and qualifiers inside construction patterns.

3 XSLT Template Specialization Principles

Specialization is a well-known technique for program optimization [10] and can reduce both program size and running time. In our context, a specialization S, of an XSLT program P, is a new XSLT program containing a subset of the templates contained in P. The templates chosen be included in S may also need to undergo some modification, to eliminate dangling references and to make them more specific. Section 3.3 will later describe the modification process. Execution of S will yield the same result as execution of P, for certain combinations of user inputs.
We now describe two broad kinds of XSLT specialization schemes: one is *Spatial Specialization,* which creates a number of specializations based on consideration of the spatial structure of the XML document and the likely patterns of user queries. The

other is *Temporal Specialization*, which creates specialized program versions based on the expected (temporal) calling patterns between program templates. The overall aim of specialization is to construct groupings of templates likely to be related to one another. i.e. A specialization corresponds to a possible sub-program of the original, generic one.

3.1 Spatial Specialization

We describe three different principles for spatial specialization. Spatial specialization groups templates together according to the "nearness" of nodes they can match within the DTD. It is intended to reflect the likely spatial locality of user queries. i.e. Users are likely to issue input queries containing terms (elements) close to one another in the *DTD-Graph*.

Branch_Principle(*BP*): Templates are grouped together according to sub-tree properties in the *DTD-Graph*. Each node N in the *DTD-Graph* can induce a specialization as follows: Suppose N is at level k, it forms a set Q_N consisting of N + all descendants of N + all ancestors of N along the shortest path to the root node. Q_N is now associated with a set of templates S. For each node in Q_N, find all templates in the XSLT program which contain a *select pattern* that matches the node and place these in S. We say S is now a specialization at branch level *k*.
Example: In Fig 1 (b), for each different *branch level*, nodes in the *DTD-Graph* can be grouped as below. Each of these node sets Q_i would then be associated with a set S_i of templates which can match at least one of these nodes.

- Level 0: All nodes
- Level 1: 2 sets: Q_1={n0, n1, n3, n4, n7, n8, n9}, Q_2={n0, n2, n5, n6, n10}.
- Level 2: 4 sets: Q_1={n0, n1, n3, n7, n8, n9}, Q_2={n0, n1, n4}, 3), Q_3={n0, n2, n5, n10}, Q_4={n0, n2, n6}
- Level 3: 4 sets: Q_1={n0, n1, n3, n7}, Q_2={ n0, n1, n3, n8}, Q_3={ n0, n1, n3, n9}, Q_4={ n0, n2, n5, n10}.

Kin_Principle(*KP*): Templates are grouped together based on the ancestor-descendant relationships of the corresponding elements in the *DTD-Graph.* Given a kin generation number *k*, each node N in the *DTD-Graph* can induce a specialization as follows: Construct a set Q_N consisting of N + all descendants of N of shortest distance at most k-1 edges from N. Q_N is now associated with a set of templates S in the same way as for the branch principle, above.
Example: In Fig 1, suppose the kin generation number is 3, we get three node sets, namely, Q_1={n1, n3, n4, n7, n8, n9}, Q_2={n2, n5, n6, n10} and Q_3={n0, n1, n2, n3 n4, n5, n6}.

Position_Principle(*PP*): Templates are grouped together based on the minimum distance (in any direction) between the corresponding elements in *DTD-Graph.* This differs from the kin-principle in that groupings are no longer ancestor/descendant dependent. Given a distance parameter *k*, each node N in the *DTD-Graph* can induce a specialization as follows: Construct a set Q_N consisting of N + all elements of shortest

distance at most *k* edges from N. Q_N is then associated with a set of templates S in the same way as for the branch principle above.
Example: Looking at Fig. 1, suppose the distance parameter is 1. Some of the node sets are Q_0={n0,n1,n2}, Q_1={n1,n0,n3,n4}, Q_2={n2,n0,n5,n6}, Q_5={n5,n2,n10}. Any sets which are subsets of other sets are removed.

3.2 Temporal Specialization

We now describe a single principle for temporal specialization. A temporal specialization is intended to reflect the expected execution sequence of templates at run-time, based on static analysis of calling relationships within the XSLT program. This reflects, in part, the program designer's view of which collections of templates are related.

Calling_Principle(*CP*): Templates in an XSLT are grouped together by the different calling relationship paths between templates. Work in [7] described the use of a structure known as the *XSLT Template and Association Graph(TAG)*. This is a graph where the templates are nodes and there is an arc from node x to node y if template x may call template y. Based on this structure, we form specializations which model the maximal calling sequences between templates.
Example: Suppose there are five templates t1, t2, t3, t4 and t5 in an XSLT program and the possible calling paths in the *TAG* are: t1->t2, t1->t3 and t1->t4->t5. This gives the specializations: S_1={t1, t2}, S_2={t1, t3} and S_3={t1, t4, t5}

3.3 Summary of specialization processing

The process of *XSLT Specialization* is described by Fig 5. Due to space constraints we sketch, rather than present in detail the various components.

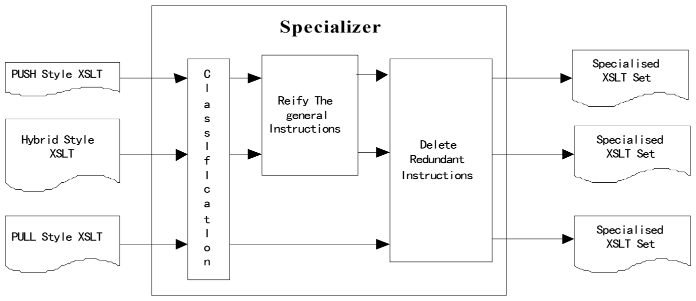


Fig. 5. Specialization process

In Fig 5, the templates of different style XSLTs are classified (grouped) using the specialization schemes(e.g *BP*, *KP*, *PP* and *CP*). Reification is then applied to any *push* style templates. In this case, any templates that contain a *<xsl:apply-templates>* instruction without a *select* attribute, have a *select* attribute added so that any possible children in the specialization are called explicitly. For *pull* style XSLT templates, *<xsl:apply-templates>* instructions are deleted if there do not exist any corresponding templates that could match the s*elect* attribute (removal of dangling references). For *hybrid* templates, we need to apply both *reify* and *delete* steps. The XSLT shown in Fig 6 is one of the possible XSLTs specialized from the original XSLT in Fig 4, using the *Branch Principle* (level 1). We call this program *SpeXSLBench.xsl*. It can be used to do the transformation if the user query is about the title of the play.

```
xsl:stylesheet version="1.0" xmlns:xsl="http://
www.w3.org/1999/XSL/Transform">
  <xsl:output encoding="utf-8"/>
  <xsl:param name="p0"/>
  <xsl:template match="PLAY">
    <html>
      <body>
        <xsl:apply-templates select="TITLE"/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="TITLE">
    <xsl:if test="$p0">
      <font style="ITALIC" color="RED">
        <xsl:value-of select="TITLE"/>
      </font>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Fig. 6. The specialized XSLT

4 Overview of XSLT specialization process

We give an overview of the *XSLT Specialization Process* in Fig 7. Fig 7(a) describes the steps performed statically and Fig 7(b) describes the steps performed at run time. Looking at Fig.7.(a), we parse the DTD into the *DTD-Graph*, parse the XSLT into an *XSLT-Tree* and *TAG* (containing information about the calling relationships [7]) and parse the XML into an *XML-Tree*. Observe that we assume existence of the XML DTD and XML data source for static analysis. This is a realistic assumption for server side XML. Next, we use the template specialization principles to specialize the original XSLT into a set of specialized XSLTs. After that, we apply an *XSLT Transformation Cost Model(XTCM)* to evaluate each specialization and label it with an expected cost. This cost determination considers four parameters: *Scan_Size* (the expected number of nodes examined during program execution), *Access_Nodes* (the expected size of the output), *Num_of_Templates* (the number of templates in XSLT) and *XSLT_File_Size*. We omit the details of the cost model due to space constraints, but note that any cost model which produces an estimate of execution time could be used.

At runtime (Fig 7 (b)), based on the user query, an XSLT template index (a simple B+ Tree data structure) is used to select the best (lowest cost) individual specialization for each term in the user query. If all terms have the same best specialization, then this specialization is used instead of the original XSLT program to answer the query. Otherwise, if different terms have different best specializations, then the original XSLT will be selected to run the transformation instead (the default case).

Recall that specializations were created under the assumption that the user query is likely to contain related terms. However, if the user query includes two or more unrelated topics, then it is unlikely they will be covered by the same best specialization. In this case, a combination of two or more specializations might be more effective than using the original XSLT program. We leave this as a topic for future research.

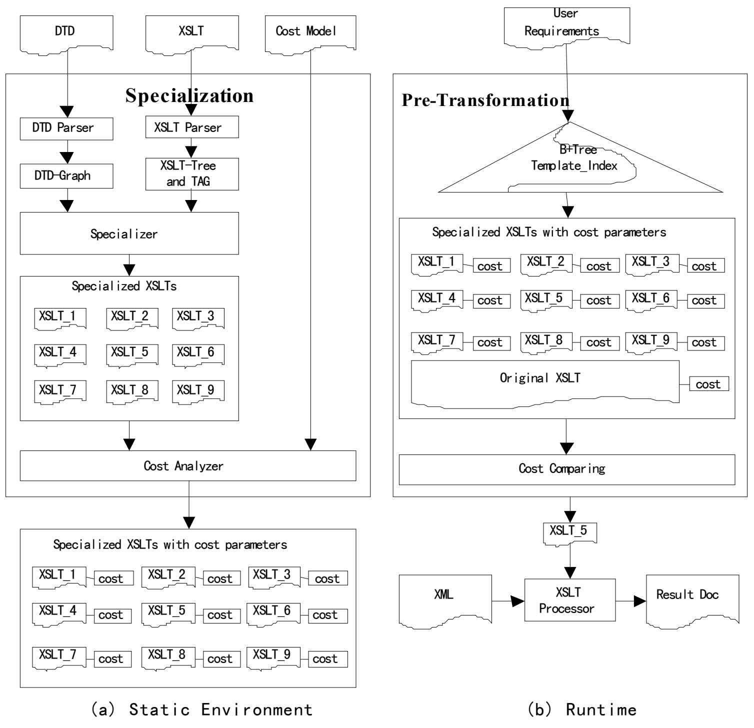


Fig. 7. The overview of XSLT specialization process

5 Experimental Results

We now experimentally investigate the effectiveness of the XSLT specializations. There are a variety of parameters which need to be considered: 1)Different XSLT styles such as *push*, *pull* and *hybrid*, 2)Different XSLT sizes: big size(consists of more than 30 templates), medium size(consists of 10 to 30 templates) and small size(consists of 10 or less templates), 3)Different *XSLT Template Specialization Principles:* including *BP*, *KP*, *PP* and *CP*. Fig 8(a) describes the space of possibilities. The test environment includes three situations 1) Different sizes of XML data: big size(14MB), medium size(7MB) and small size(4MB), 2)Different XSLT processors: Xalan-j v2.5.1 and Saxon v6.5.3, 3)Different systems(hardware and OS): Dell PowerEdge2500 (two P3 1GHz CPU and 2G RAM running Solaris 8(X86)) and an IBMeServer pSeries 650 (eight 1.45GHz CPU, 16GB RAM, running AIX5L 5.2). All of these considerations are shown in Fig.8(b).

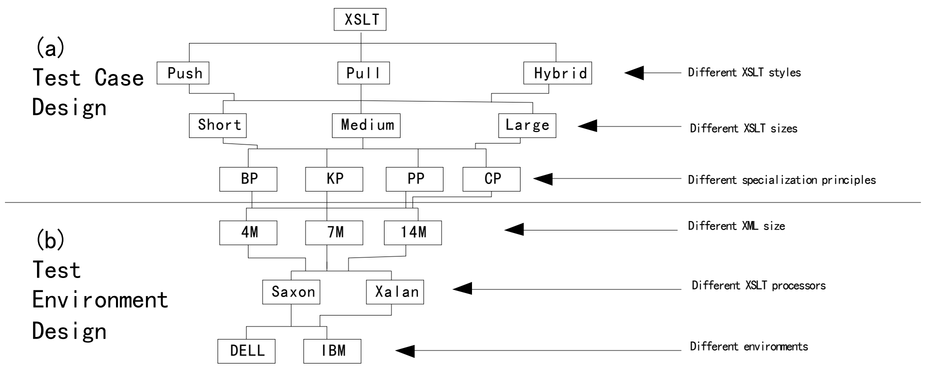
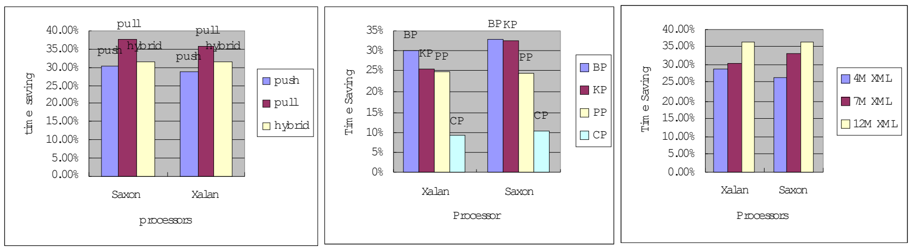


Fig. 8. Test case and environment design

So, any given test requires at least 6 different parameters. The *Shakespeare.xml* is used as the basis for the XML source data and the program *XSLBench.xsl* (shown earlier) + 2 other synthetic programs (one *push* style XSLT and one *pull* style XSLT) are used for the XSLT source. The value 3 was used as a parameter in *BP* and *KP* respectively and the value 1 was used for the parameter of distance in *PP*. Additionally, we test using all possible single query inputs and all possible combined query parameter inputs containing at most 3 terms. We choose the average transformation time over all such queries. For example, if there are 3 possible parameter values, p1, p2 and p3 (representing 3 different query terms), we test the specialization by giving the parameter p1, p2, p3, p1-p2, p1-p3, p2-p3 and p1-p2-p3 respectively and then calculate the average transformation time over all queries.

We show charts below summarising the improvements. We concentrate on contrasting 1)different XSLT styles, 2)different template specialization principles, 3) different size XML data sources. Average time saving improvement compared to the original program is given. e.g. If we want to determine the effect of specialization for different size XML inputs, we classified all test results by the test parameter of XML size(there are three in this paper: 4MB, 7MB and 14MB). And for each group of test results we generate the average value as the final time saving improvement. Then we build a bar chart to compare the effect of different groups.(some sub-classifications are setup to display more information of contrast. e.g. under each group of different XML sizes, we make the sub-classification of different XSLT processors.)

In Fig.9.(a) we see that template specialization is effective across the three different XSLT design styles, with the *pull* style XSLT giving the greatest savings. *Pull* style obtains the most improvement, since pull style templates require more expensive access to the XML data, due to the need for explicit XPath expressions in *construction patterns*. From Fig.9.(b), we see all specialization principles give improvements in the average transformation time. The *Branch_Principle* is the most effective, since it generates the most specializations and hence one is more likely to cover the query. Conversely, the *Calling Principle* is the least effective, since it generates the fewest specializations (due to not few calling paths existing in the program). From Fig.9.(c) we see that the technique of template specialization is effective for varying XML sizes and is most effective for bigger size XML data size, because evaluating XPath expressions takes more time for large XML input and the specializations contain fewer XPath expressions than the original XSLT program.



(a) Different XSLT styles (b) Different Specialization Principles (c) Different Size XML Files

Fig. 9. The summarized experimental results

Considering the XSLT transformation time [15], it can be broken into a number of parts shown in Fig. 10.

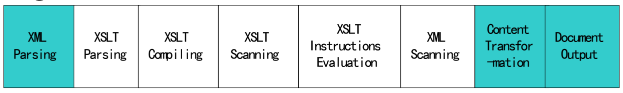


Fig. 10. The different parts of time spend on XSLT transformation

Among all these components, the times for XML parsing, content transformation (assuming the same document output) and document output serialization are unalterable, given a specific XSLT processor and XML source. Oppositely, the times for XSLT parsing, XSLT compiling, XSLT scanning and XML scanning are all alterable. Each specialization contains fewer templates than the original XSLT program. Consequently XSLT parsing, XSLT compiling and XSLT scanning is expected to take less time. Furthermore, the specializations are likely to require less time in XML scanning, since the scanning time is related to the XPath expressions of select attributes for template instructions in *<xsl:apply-templates>*. These can be changed during the specialization process. Since we treat the XSLT processor as a black box it isn't possible to more accurately analyse the savings for each component.

6 Related Work

To the authors' knowledge, there is no other work which considers specialization for XSLT programs. There is of course a rich body of work on specialization for various kinds of programs [10, 16]. The difference for the XSLT scenario, compared to, say, functional or logic programs, is that XSLT is data intensive and a DTD is provided. XSLT based optimization has been considered by Z. Guo, M. Li et al in 2004[9]. They use a streaming processing model *(SPM)* to evaluate a subset of XSLT. By *SPM*, an XSLT processor can transform an XML document to other formats without using extra buffer space. However, some strong restrictions on the XSLT syntax and design are made, limiting the applicability. XPath based XML query optimization has been considered in a large number of papers, e.g. S. Abiteboul and V. Vianu in 1997, A. Deutsch and M. Fernandez et al in 1999, Li and Moon in 2000.[1, 5, 6, 14]

7 Conclusion and Future Work

In this paper, we have proposed a new approach for optimization of server-side XSLT transformations using template specialization. We described several specialization schemes, based on notions of spatial and temporal locality. Experimental evaluation found that use of such specializations can result in savings of 30% in execution time. We believe this research represents a valuable optimization technique for server-side XSLT design and deployment. As part of future work, we would like to investigate extending our methods and algorithms to handle further XSLT syntax, such as the wider use of built-in templates, and functions within *construction patterns*. We also plan consider the possible cost models in more detail and analyse methods for combining specializations.

References

[1]S. Abiteboul and V. Vianu. Regular path queries with constraints. In *the 16th ACM SIGACT-SIGMOD-SIGSTART Symposium on Principles of Database Systems*,, AZ, 1997.

[2]T. Bray, J. Paoli, and C. M. Sperberg-McQueen, and E. Maler (2000): W3C Recommendation. Extensible Markup Language (XML) 1.0

[3]J. Clark. (1999): W3C recommendation. XSL Transformations (XSLT) version 1.0

[4]http://www.datapower.com/xmldev/xsltmark.html

[5]A. Deutsch and V. Tannen. Containment and integrity constraints for XPath. In *Proc. KRDB 2001,* CEUR Workshop Proceedings 45, 2003.

[6]A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu. A query language for XML. In *Proc.of 8th Int'l. World Wide Web Conf,* 1999.

[7]C. Dong and J. Bailey. The static analysis of XSLT programs. *Proc.of The 15th Australasian Database Conference*, Vol.27, Pages 151-160, Dunedin, New Zealand, 2004.

[8]R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semi-structured database. *Proc. Int'l Conf on VLDB*, Athens, Greece, 1997.

[9]Z. Guo, M. Li, X. Wang, and A. Zhou, Scalable XSLT Evaluation, *Proc. of APWEB 2004*, HangZhou, China, 2004.

[10] N. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 1996.

[11]M. Kay. (2000): Saxon XSLT Processor. http://saxon.sourceforge.net/.

[12]M. Kay. Anatomy of an XSLT Processor, 2001.

[13]C. Laird. XSLT powers a new wave of web, 2002.

[14]Q. Li, B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. Int'l Conf on VLDB*, Roma, Italy, 2001.

[15]S. Maneth and F. Neven Structured document transformations based on XSL. Proceedings of *DBPL'99*, Kinloch Rannoch, Scottland, 2000.

[16]W3C. XSL transformations(XSLT) version 2.0. http://www.w3.org/TR/xslt20/.

[17]World Wide Web Consortium. XML Path Language(XPath) Recommendation. http://www.w3.org/TR/xpath.