Optimization of XSLT by Compact Specialization and Combination

Ce Dong James Bailey

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

Abstract. In recent times, there has been an increased utilization of server-side XSLT systems as part of e-commerce and e-publishing applications. For the high volumes of data in these applications, effective optimization techniques for XSLT are particularly important. In this paper, we propose two new optimiza-tion approaches, Specialization Combination and Specialization Set Compac-tion, to help improve performance. We describe rules for combining specialized XSLT stylesheets and provide methods for generating a more compact speciali-zation set. An experimental evaluation of our methods is undertaken, where we show our methods to be particularly effective for cases with very large XML in-put and different varieties of user queries.

1 Introduction

The standardized, simple and self describing nature of XML makes it a good choice for arbitrary data sources representation, exchange and storage on World Wide Web [15]. The eXtensible Stylesheet Language Transformation (XSLT) standard [5] is a primary language for transforming, reorganizing, querying and formatting XML data. In particular, use of server-side XSLT is an extremely popular technology [27] for processing and presenting results to user queries issued to a server-side XML database (e.g. google map, google search, Amazon web services, and the ebay developer program).

Although faster hardware can of course be used to improve transformation speed, server-side XSLT transformations can still be very costly when large volumes of XML and large size XSLT stylesheets are involved (e.g. an XSLT transformation based on a 1000MB XML document and a relatively simple XSLT stylesheet can take up to 128 minutes [23]). Techniques for the optimization of such transformations are therefore an important area of research [12].

In previous work [9], we proposed an optimization method for XSLT programs based on the well known technique of program specialization [1, 18, 20], called *XSLT Template Specialization* (XTS) [9]. 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 web browser). For example, a customer of an online XML based department store might pass a query with several query terms to an XSLT program, referring to a 'Computer' with particular requirement of 'CPU'. The department store server-side XSLT program may contain logic which is designed to present other different kinds of merchandise (e.g. 'Desk'), but they are not 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 runtime. Important savings in execution time and consequently response time improvement were clearly shown by the experimental results [9].

The effectiveness of the XTS optimization technique is restricted to generating specializations only for *strong interconnection queries* [6], which are queries whose terms refer to XML tags that are 'close' to each other in the XML-tree (we further describe the concept of *interconnection* in section 2.1). In this paper, we present a new method, that is able to handle *weak interconnection queries*, whose query terms refer to XML tags that can be 'far apart' from each other in the XML-tree. Such *weak interconnection queries* are common in the real world [6]. Our method is called *Specialization Combination* (SC) and intuitively, it constructs specializations that cover *disconnected* sub-graphs of the DTD.

An additional challenge faced by specialization based optimization, is the potentially large number of specializations that may need to be created to cover the anticipated user queries (e.g. based a 100 template *Original XSLT*, hundreds, or even thousands of specializations might be generated). Such a large specialization set can increase the cost of searching for and finding an appropriate specialization, when the server-side system responds to the user query at run-time. In this paper, we also present an approach called *Specialization Set Compaction* (SSC), that balances the *user query coverage*, *search cost* and *transformation cost* for sets of specializations, and generates a more *Compact Specialization Set* (CSS), that is more suitable for use at run-time.

Our contributions in this paper are two optimization methods suitable for use with specialized XSLT programs:

- *Specialization Combination* (SC), which combines XSLT specializations together in order to handle *Weak Interconnection Queries*.
- *Specialization Set Compaction* (SSC), which produces a *Compact Specialization Set*, that reduces the specialization search space and allows quicker selection of a specialization run-time.

Experimental results demonstrate the ability of these techniques to yield speedups of up to 40%. We are not aware of any other work, which uses similar concepts to *Specialization Combination* (SC) and *Specialization Set Compaction* (SSC), for improving the performance of XSLT programs.

The remainder of this paper is as follows. We first review some basic concepts in section 2. Then, in section 3 we describe the process of *Specialization Combination*

(SC). Next, in section 4, we propose the approach of *Specialization Set Compaction* (SSC) and in section 5 overview the optimization processes of SC and SSC. In section 6, we present and discuss our experimental evaluation of the techniques. Related work is surveyed in section 7 and in section 8, we conclude and discuss future work.

2 Background

We begin by briefly reviewing some useful concepts. We assume basic knowledge of XML and XSLT [5, 21, 22].

2.1 DTDs, DTD-Graph and Interconnection of Query Terms

A DTD provides a structural specification for a class of XML documents and is used for validating the correctness of XML data. A DTD-Graph is a data structure that summarizes the hierarchical information within a DTD. It is a rooted, node-labeled graph, where each node represents either an element or an attribute of the DTD, and the edges indicate element nesting [9, 14]. The nodes in a DTD-Graph can be thought of as corresponding to either XML tags, or XSLT template *selection patterns*, or user query terms. An example of a DTD and its corresponding DTD-Graph is shown in Fig.1.

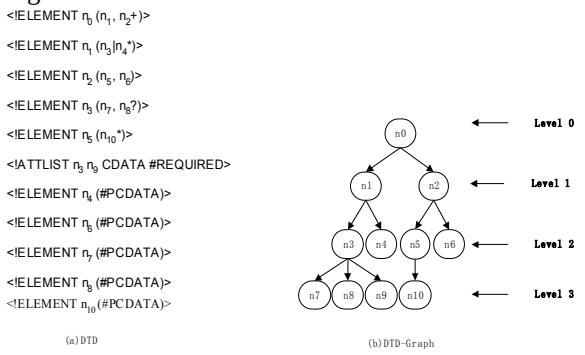

Fig. 1. DTD and corresponding DTD-Graph

Based on the kinds of different DTD-Graph structures, XML documents can be classified into four types [17]: i) *Broad-Deep* XML, ii)*Narrow-Deep* XML iii)*Broad-Shallow* XML and iv)*Narrow-Shallow* XML. Our new techniques (SC and SSC) will be evaluated on XML datasets having these different kinds of structures (discussed in section 6).

A user query for a server-side XSLT system consists of one or more query terms (modeled as nodes in the DTD-Graph). It can be expressed as q = {$term_1$, $term_2$, … $term_n$} (n>=1). A query, which consists of terms (nodes) such that every pair of

terms is separated by at most 2 edges in the DTD-Graph, is defined as a *strong interconnection query*. Otherwise, it is a *weak interconnection query*. In our online department store example, query q_1 (expressed as q_1={Computer, CPU}), would be a *strong interconnection query* if 'CPU' was a child node of 'Computer' in the DTD-Graph. Whereas, query q_2 (expressed as q_2={Computer, CPU, Desk}), would be a *weak interconnection query* if, additionally, 'Computer' and 'Desk' were sibling nodes under a 'Merchandise' node (the 'Desk' node is 3 edges far from 'CPU' node) in the DTD-Graph.

Weak interconnection queries are reasonably prevalent when the user asks for data from different sub-structures of the XML tree, and they cannot be ignored in the real-world [6]. According to our online department store example, a customer may quite possibly buy a desktop 'Computer' (i.e. with Pentium IV 'CPU') and a matched 'Desk' together in one purchase. Many other similar examples exist.

2.2 XSLT Templates, Server-Side XSLT

An XSLT program consists of a set of templates. Execution of the program is by recursive application of individual templates to the source XML document [22]. XSLT stylesheets can be designed using three principal styles [22]: i)*Push* style, ii)*Pull* style and iii) *Hybrid* style. We will use these different styles of XSLT when constructing test cases in our experimental evaluation (described in section 7).

Server-side XSLT is a popular solution for data exchange and querying on the Web. It is often deployed for e-commerce, e-publishing and information service applications. A problem that can occur in the use of server-side XSLT is that, when a generic XSLT program is designed, it is able to handle a broad range of possible user inputs. At run-time, given specific user query inputs, much of the logic of this generic XSLT program may not be required. This results in increased run-time, since extra logic may be required for XPath evaluation and template matching [22].

2.3 XSLT Template Specialization Technique (XTS)

Our previous work, using the *XSLT Template Specialization* (XTS) technique, automatically creates a set of specializations offline and selects an appropriate minimum cost XSLT from the specialization set at run-time, to respond to the user query input.

The XTS technique is not effective at generating specializations for *weak interconnection queries.* This is because the specialization principles for the XTS technique, generate an XSLT specialization by grouping together templates (represented as nodes in DTD-Graph), that correspond to a collection of nodes in a *connected* sub-graph of the DTD-Graph. This *connected* sub-graph can obviously be big, if two existing nodes (query terms) that need to be covered are 'far apart' (weakly connected) to each other. Specializations generated based on this sub-graph can consequently be 'overweight', since they can contain templates that need not be involved in answering the *weak interconnection query*. Consequently, the transformation

time is potentially slower. Therefore, instead of forming *specializations* that correspond to a (potentially large) *connected* sub-graph of the DTD-Graph, we would like to form specializations that correspond to a smaller, *disconnected* sub-graph of the DTD-Graph. This is the motivation behind our method of *Specialization Combination* (SC). Small (*connected*) specializations are combined together to form a new (*disconnected*) specialization, instead of using 'overweight' specializations or even the *Original XSLT*, to execute the transformation for the *weak interconnection query*.

Another problem which can arise, is that the number of specializations that are generated as a result of specialization combination may be very large. This enlarged specialization set takes more time to search, when a given specialization needs to be identified at run- time in response to a user query. To resolve this problem, we have developed the method of *Specialization Set Compaction* (SSC), to reduce the size of the specialization set.

3 Specialization Combination

Assume we have a log file mechanism associated with the server-side XSLT system, which, at run-time, can record information about user queries and the corresponding specializations that can be used to answer them. If using the XTS technique, it is likely that the log file will indicate that some queries (*weak interconnection queries*) need to be handled by the *Original XSLT* or 'overweight' specializations. In other words, the optimization effectiveness of the XTS technique is degraded when *weak interconnection queries* are passed to the system. In this section, we propose a method called *Specialization Combination* (SC), to combine pertinent specializations together, instead of using the *Original XSLT* or 'overweight' specializations, to deal with the *weak interconnection queries*. This saves execution time. Suppose we already have the specialization set that was generated based on the XTS technique. Call this the *Primary Specialization Set* (PSS). The process of combining the specializations for a specific *weak interconnection query*, can now be described as follows (we demonstrate using an example across all steps):

- Step_1: For each query term of a *weak interconnection query*, we list all specializations in the *Primary Specialization Set* (PSS), which can handle this query term. Suppose q is a *weak interconnection query* which consists of query terms t_1 and t_2, (expressed as $q=\{t_1,t_2\}$) and term t_1 can be handled by specializations s_1 or s_2 and term t_2 can be handled by specializations s_3 or s_4. We generate the specialization set list as: $\{s_1 \text{ or } s_2\}\bullet t_1$, $\{s_3 \text{ or } s_4\}\bullet t_2$.
- Step_2: Generate all possible specialization combinations which can 'cover' all query terms for a specific *weak interconnection query*. Based on the example above, we generate the combinations $\{s_1s_3\}$, $\{s_1s_4\}$, $\{s_2s_3\}$ and $\{s_2s_4\}$. At a high level, the details of the generation process are: 1) place templates from different specializations into one <xsl:stylesheet> element, 2) place the contents from all repeated templates into one template and delete the redundant templates; 3)delete the redundant content in each template.

- Step_3: Re-calculate the cost for each combined specialization according to some cost model (any cost model is permitted here. We use a simple one, details not included due to space restrictions)

We add all the generated combined specializations to the *Primary Specialization Set* (PSS) to form the new *Refined Specialization Set*, (hereafter referred to as RSS).

4 Specialization Set Compaction

A RSS may turn out to be very large, if the XTS technique and SC techniques are applied to an *Original XSLT* having many templates. The server-side XSLT system may therefore take a lengthy time to search and select an appropriate *specialization* in answer to a given user query. Even though indexing methods can be applied to speed up the *specialization* searching, many *specializations* that have never been useful for answering any user query, might still have to be scanned at run-time. Deleting the non-invoked *specializations* from the RSS is not necessarily a good idea for forming a more compact specialization set. This is because non-invoked *specializations* may still be good candidates for inclusion, since even though they may not have the smallest cost, they can still cover a relatively large number of query terms. Accordingly, we propose a novel approach, *Specialization Set Compaction* (SSC), to produce a *compact specialization set* which has the minimum total cost, given an expected query set with some estimated distribution.

4.1 Query Set and Refined Specialization Set

Assuming the existing query log has the records of queries issued to the server, then, we can generate summary data listing the distinct queries and their corresponding probabilities. This related data is shown in Fig.2.(a), where QS denotes the *query set*, each q_i denotes a *distinct* query, and each p_i denotes its corresponding *probability* of being issued. The *Refined Specialization Set* (RSS) generated by the *Specialization Combination* (SC) technique is shown in Fig.2.(b), where s is used to denote an individual specialization and s_{orig} denotes the *Original XSLT*, which is selected by the specialized server-side XSLT system to process any queries that cannot be handled by any single *specialization*.

$$QS = \overset{\uparrow\ \uparrow\ \uparrow\ \uparrow\ \uparrow}{\{q_1, q_2, q_3 \ldots q_i \ldots q_n\}}^{p_1, p_2, p_3 \ldots p_i \ldots p_n} \qquad RSS = \{s_{orig}, s_1, s_2, s_3 \ldots s_{m-1}\}$$

(a) Query Set (b) Refineded Specialization Set

Fig. 2. Query Set and Refined Specialization Set

4.2 Time Cost Analysis

There are two aspects that determine the runtime cost of answering a user query: i)searching and selecting the appropriate *specialization* from RSS (this time cost is denoted as T_s), ii) executing the XSLT transformation to generate the answer (this time cost is denoted as T_e). Hence, the total processing time T is equal to $T_s + T_e$.

The average T_s depends on the size of the RSS. It can be described as $T_s=d|RSS|$ (|RSS| denotes the cardinality of *Refined Specialization Set* (RSS) and *d* represents the relationship between |RSS| and T_s, which varies according to the specialization search strategy).

T_e is defined as T_e=Time_Exe(q,s) (s∈ RSS and the function Time_Exe() is used to measure the transformation time for query q using specialization s). If s is a single specialization, which can cover all query terms in q, then the value for Time_Exe(q,s) is expected to be finite . However, if s can not cover all query terms in q, the value of Time_Exe(q,s) is considered to be ∞. Additionally, s_{orig} is the *Original XSLT*, which can cover all possible legal queries and the value of Time_Exe(q, s_{orig}) is always finite.

4.3 The Compact Specialization Set

The *Compact Specialization Set* (CSS) is a subset of RSS and is required to be small, so it does not take too much time to search through. Also, each element in the CSS should have relatively low cost and be applicable to a lot of situations (query terms). We wish to choose a CSS that minimizes the value of the following formula:

Formula_1:

Total_Time(CSS) =Total_Searching_Time(CSS)+Total_Execution_Time(CSS)

$$= \sum_{i=1}^{n} p_i * [d|CSS| + Min_{s\in CSS} (Time\_Exe(q_i,s))]$$

- d|CSS|:: T_s, the time spent on finding a minimum cost specialization from CSS to handle q_i
- $Min_{s\in CSS}$ (Time_Exe(q_i,s)):: T_e, the minimum time spent on executing s (s∈ CSS) to answer q_i.
- d|CSS| + $Min_{s\in CSS}$ (Time_Exe(q_i,s)):: T, the total time for processing q_i based on CSS
- p_i * [d|CSS| + $Min_{s\in CSS}$ (Time_Exe(q_i,s))] :: The time spent for processing the distinct query q_i, which has the probability p_i, based on CSS.
- $\sum_{i=1}^{n} p_i * [d|CSS| + Min_{s\in CSS} (Time\_Exe(q_i,s))]$ :: the total time for processing all distinct queries (q_1,q_2,…q_n) in QS based on the CSS

4.4 Generate the Approximation of Compact Specialization Set

In generating the CSS, if the RSS is not too large, we can generate all subsets of it and choose the one which has the minimum Total_Time according to Formula_1. However, this method is impossible if the RSS is a large set. For example, a RSS with 100 *specializations* has 2^{100} subsets.

It is well known that the query distribution of Web based searching applications is asymmetrical [15]. A small number of distinct queries account for most searches (i.e. have high summed probability (sum(p_i))). Using this knowledge, we can dramatically reduce the search space. Our method is related to the well known problem of computing the transversals of a hypergraph or the vertex cover problem [11]. It consists of the following steps:

- Step_1: Sort the distinct queries in QS according to descending probabilities and create a list of specialization sets (LSS), where each specialization set in LSS consists of all *specializations* which can cover the corresponding query q_i in QS. Also, for each specialization set of LSS, we order the elements in ascending cost from left to right. Step_1 is described in Fig.3.(a)

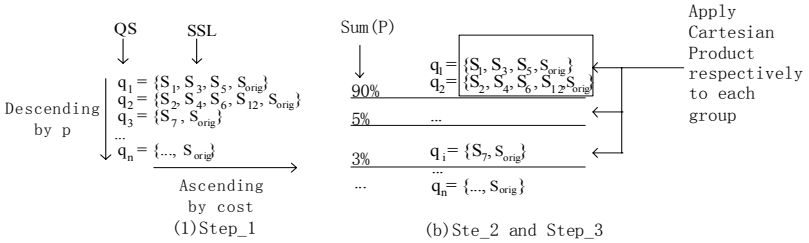

Fig. 3. Steps of Specialization Set Compaction

- Step_2: Slice the LSS horizontally into different groups, according to a predefined threshold list of descending sum (p_i). Specifically, we slice the LSS into 5 groups and define the required values of sum (p_i) for each group respectively as 90%, 5%, 3%, 1.5% and 0.5%. This predefined threshold list of sum(p_i) is defined based on analysing the *user query set* and must obey the following two policies : i) distinct queries with high probability must be grouped in the first slice and ii) sum(p_i) of the first slice should be big enough to cover most of user queries (e.g. 90%). For the groups with small sum (p_i), we only keep several (e.g. 1 or 2) of the leftmost *specializations* of each specialization set and omit (delete) other the other relatively bigger cost *specializations*, since they have low probabilities and only have a small impact on the final result.
- Step_3: Apply a *Cartesian-Product* operation, to each sliced group of specialization sets and generate new specialization sets such that every set can cover all the distinct queries in that group. Step_2 and step_3 are illustrated in Fig.3.(b).
- Step_4: For each group, select the set which has the minimum value of Total_Time, by evaluating all candidates using Formula_1.

- Step_5: Combine all of the specialization sets selected by step_4 into one set and delete the redundant *specializations*. Thus, we obtain the final approximation of *Compact Specialization Set* (CSS).

If we want to obtain a more accurate result, we can also make a second round *Cartesian-Product* on the specialization sets generated by step_3. Then test the result sets one by one based on Formula_1 to choose the minimum set as final CSS.

5 The Overview of Compact Specialization and Combination

After applying the optimization approaches of *Specialization Combination* (SC) and *Specialization Set Compaction* (SSC), we obtain a *compact specialization set* (we will see later, in the experimental results, there exist about 70-80% fewer *specializations* in the RSS after compaction) as output to handle the user queries at the runtime. We overview our techniques in Fig.4.

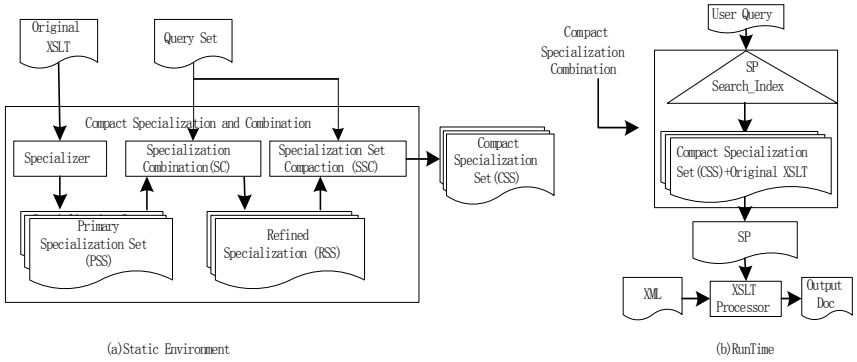

Fig. 4. Overview of Compact Specialization and Combination

From Fig.4, we see that in the static environment, the system generates the *Primary Specialization Set* (PSS) based on the *Original XSLT* using the specialization principles of the XTS technique and then applies the approach of *Specialization Combination* (SC) to generate the RSS. Next, the *Specialization Set Compaction* (SSC) technique produces the *Compact Specialization Set* (CSS) as the final output. At runtime, our server-side XSLT system uses an XSLT specialization index to select the best (lowest cost) individual *specialization* in the *Compact Specialization Set* (CSS) for responding to the user query (The *Original XSLT* is retained as a member of the *Compact Specialization Set* (CSS) for handling any new queries which can not be handled by any *specialization*). We can repeat the process of *Compact Specialization and Combination* (SC plus SSC), based on the latest data of the query set (QS) (e.g. every month), to increase the accuracy of the *Compact Specialization Set*.

6 Experimental Results

We choose XSLT and XML test cases by considering the different XSLT design styles [22] and different XML DTD-Graph structures [17] (mentioned in section 2.1 and 2.2). Specifically, *shakespeare.xml* (*Broad-Deep* structure), is used as the XML input data source for *XSLBench1.xsl* (*Push* style XSLT), *XSLBench2.xsl* (*Pull* style XSLT) and *XSLBench3.xsl* (*Hybrid* style XSLT). The *brutal.xml* (*Narrow-Deep* structure) is the XML input data source of *brutal.xsl*, and *db8000.xml* (*Broad-Shallow* structure) is the XML input of *db.xsl*.

Experimental results were generated for two different XSLT processors: i)Xalan-j v2.6, ii) Saxon v8.3, and two different system environments (hardware&OS): i) Dell PowerEdge2500 (two P3 1GHz CPU and 2G RAM running Solaris 8(X86)), ii) IBMeServer pSeries 650 (eight 1.45GHz CPU, 16GB RAM, running AIX5L 5.2).

To simulate the user query inputs, we stipulate that each query should consist of 1-3 terms [15]. Then, based on the possible query term space for each XSLT, we randomly generate three different query sets, QS_1, QS_2, and QS_3, each containing a specific percentage of *weak interconnection queries.*

Also, in order to simulate the asymmetrical query distribution of Web applications [15] (mentioned in section 4.4), we stipulate that 10% of the queries cover 90% of the probability space.

The following is a description of our testing methodology for each XSLT:

- Generate the *Primary Specialization Set* (PSS) based on the *Original XSLT stylesheet*, using different *specialization principles* (described in section 2.3) with possible values of parameter *k* being (1, 2, 3…).
- Test the *Primary Specialization Set* (PSS) using the XTS technique for different query sets (QS_1, QS_2 and QS_3) respectively and record the average processing time for each.
- Generate the *Refined Specialization Set* RSS_i (i=1,2,3), based on the *Primary Specialization Set* (PSS) and the query set QS_i (i=1,2,3), using the technique for *Specialization Combination* (SC)
- Test the *Refined Specialization Set* RSS_i (i=1,2,3), using the query set QS_i (i=1,2,3) and record the average processing time across RSS_1, RSS_2 and RSS_3.
- Generate the *Compact Specialization Set* CSS_i (i=1,2,3), based on the *Refined Specialization Set* RSS_i (i=1,2,3) and corresponding query set QS_i (i=1,2,3), using the technique of *Specialization Set Compaction* (SSC).
- Test the *Compact Specialization Set* CSS_i (i=1,2,3) based on the corresponding query set QS_i (i=1,2,3) and record the average processing time among CSS_1, CSS_2 and CSS_3.

The optimization performance of our new techniques is effective and encouraging. We provide the comparison of processing time and corresponding time saving (compared to *Original XSLT*) between different optimization approaches (XTS, CS and SSC) in Table.1.

The value under column 'Orig' is the average transformation time using the *Original XSLT* stylesheet, the value under column 'T_e' is the XSLT execution time;

the value under column 'T_s' is the time used to search for an appropriate *specialization* from the relevant specialization set (PSS, RSS or CSS); the value under column 'T' is the total processing time, $T=T_e+T_s$; the value under column 'S%' is the percentage of processing time saved for the optimization approach compared with the *Original XSLT*, S%=(((Orig-T)/Orig)*100)%. All processing time in Table.1 is expressed in seconds and computed based on 1000 user queries.

Table 1. The processing time(seconds) and time saving(%)

	Orig	Query Set	XTS (based on PSS)				SC (based on RSS)				SSC (based on CSS)			
			Te	**Ts**	**T**	**S%**	**Te**	**Ts**	**T**	**S%**	**Te**	**Ts**	**T**	**S%**
XSLBench1	**4993**	QS1	3010	372	3382	32%	2312	536	2848	43%	2422	236	2658	47%
		QS2	3248	372	3620	27%	2374	887	3261	35%	2478	193	2671	47%
		QS3	3519	371	3890	22%	2349	1266	3615	28%	2499	229	2728	45%
		Avg				**27%**				**35%**				**46%**
XSLBench2	**5735**	QS1	3831	515	4346	24%	2414	686	3100	46%	2530	195	2724	52%
		QS2	4081	512	4593	20%	2889	1030	3918	32%	2892	186	3078	46%
		QS3	4252	513	4766	17%	2672	1351	4023	30%	2726	250	2976	48%
		Avg				**20%**				**36%**				**49%**
XSLBEnch3	**5467**	QS1	3580	461	4041	26%	2492	651	3143	43%	2604	143	2747	50%
		QS2	3827	460	4287	22%	2623	1030	3652	33%	2717	150	2868	48%
		QS3	4156	463	4619	16%	2550	1373	3923	28%	2598	179	2777	49%
		Avg				**21%**				**35%**				**49%**
brutal	**2678**	QS1	1803	272	2075	23%	1388	408	1796	33%	1468	135	1603	40%
		QS2	1979	278	2257	16%	1403	601	2004	25%	1517	157	1674	37%
		QS3	2164	269	2433	9%	1394	837	2231	17%	1502	164	1666	38%
		Avg				**16%**				**25%**				**38%**
db	**3228**	QS1	2110	300	2410	25%	1608	450	2058	36%	1819	150	1969	39%
		QS2	2287	302	2590	20%	1627	672	2299	29%	1837	143	1980	39%
		QS3	2474	298	2773	14%	1590	894	2484	23%	1794	167	1961	39%
		Avg				**20%**				**29%**				**39%**

These results illustrate that, firstly, that our new approaches, *Compact Specialization Combination* (SC+SSC), can effectively improve the server-side XSLT processing time compared to the *Original XSLT* and the XTS optimization technique (our previous work). The saving of processing time is due to i) the use of combined specializations, instead of the *Original XSLT* or 'overweight' *specializations*, to handle *weak interconnection queries* and consequently save XSLT execution time (T_e) and ii) the use of *Compact Specialization Set* (CSS) to reduce the search space and save the *specialization* searching time (T_s) at runtime.

Secondly, the *Specialization Combination* (SC) technique is more effective for improving the XSLT execution time (T_e) for query sets which have a higher number of *weak interconnection queries*. Moreover, *Specialization Set Compaction* (SSC) inherits the advantage of dealing with *weak interconnection queries* from the SC

technique and is more effective at reducing the specialization search time (T_s) for the larger specialization set.

Thirdly, our new technique, *Compact Specialization and Combination* (SC+SSC), can be applied to XSLT designed in the three different styles (*Push* style, *Pull* style and *Hybrid* style) and XML documents designed in different structures (*Broad-Deep*, *Narrow-deep* and *Broad-Shallow* DTD-Graph structures). It is more effective for XML designed in the *Broad-Deep* DTD-Graph structure, since i) the query terms of a *weak interconnection query* for a *Broad-Deep* XML database might be more 'far apart' than *Narrow* or *Shallow* XML and the *specialization* (generated with our previous XTS technique) used to handle the *weak interconnection query* might be more 'overweight' and so the combined *specialization* can save relatively more execution time; ii)the larger specialization set might be generated based on the *Broad-Deep* XML database compared with *Narrow* or *Shallow* XML, and the technique of *Specialization Set Compaction* (SSC) can prune relatively more search space. *Compact Specialization and Combination* is not applicable for small XSLT stylesheets and the *Narrow-Shallow* XML, since too few specializations are able to be generated.

We conducted all the above tests using the Xalan XSLT processor as well. The experimental results were similar to that for Saxon (Table.1).

7 Related Work

To the best of our knowledge, there is no other previous work, which considers *Specialization Combination* or *Specialization Set Compaction* for optimizing server-side XSLT programs. Extensive study has been done specialization for various kinds of programs [1, 18, 20]. The main difference on specialization between XSLT scenario and functional or logic programs is that XSLT is data intensive and, usually, a data schema (DTD or XML-schema) is provided. XSLT and XQuery based optimization have been considered in [16, 19, 28, 31]. Our optimization method, differs from [16, 19, 28, 31], since it focuses on XSLT stylesheets and uses statistics from query logs. It can be applied regardless of XSLT processor and hardware&OS platform. XPath or XML index based query optimization has been considered in a large number of papers [2, 8, 26]. The DTD-Graph mentioned in this paper is similar to the Dataguide structure described by Goldman and Widom in 1997 [14].

8 Conclusion and Future Work

In this paper, we have proposed two new approaches: *XSLT Specialization Combination* (SC) and *XSLT Specialization Set Compaction* (SSC), for the task of optimizing server-side XSLT transformations. We have shown that *Compact Specialization and Combination* (SC +SSC) significantly outperform the *Original XSLT* transformation and the method of our previous work (XTS). Based on the technique of SC, the sys-

tem can process and optimize not only *strong interconnection queries*, but also another very important class of user queries, *weak interconnection queries*. Moreover, based on the technique of *Specialization Set Compaction* (SSC), we reduce the size of the *Refined Specialization Set* (RSS) and, practically, generate an approximate *Compact Specialization Set* (CSS) to further improve performance.

Our experimental results showed that these new approaches provide more effective optimization (saving about 40-50% in processing time compared with the *Original XSLT*) for server-side XSLT transformation. As part of our future work, we plan 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*.

References

[1]M. Alpuente and M. Hanus.: Specialization of inductively sequential functional logic programs. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming table of contents*. (1999) 273 – 283.

[2]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) 122–133

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

[4]C.Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi.: Efficient Filtering of XML Documents with XPath Expressions, Proceedings of *Intl' Conference on Data Engineering*, San Jose, California (2002) 235-244

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

[6]S. Cohen, Y. Kanza and Y. Sagiv.: Generating Relations from XML Documents. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, Siena (Italy) (2003) 285-299

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

[8]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*. Toronto, Canada (1999) 1155-1169

[9]C. Dong and J. Bailey.: Optimization of XML Transformations Using Template Specialization. In *Proc.of The 5th International Conference on Web Information Systems Engineering (WISE 2004)*, Brisbane, Australia (2004) 352-364

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

[11] T. Eiter and G. Gottlob. Identifying the Minimal Transversals of a Hypergraph and Related Problems. *SIAM Journal of Computing* 24(6) (1995) 1278-1304.

[12]W. Fan, M. Garofalakis, M. Xiong, X. Jia.: Composable XML integration grammars. In *Proceedings of Thirteenth ACM conference on Information and knowledge management*. Washington, D.C., USA (2004) 2-11

[13]M. Gertz, J. Bremer.: Distributed XML Repositories: Top-down Design and Transparent Query Processing. *Technical Report CSE-2003-20*, Department of Computer Science, University of California, Davis, USA (2003)

[14]R. Goldman and J. Widom.: Enabling query formulation and optimization in semi-structured database. *Proc. Int'l Conf on VLDB*, Athens, Greece (1997) 436-445
[15]Google Gulde. http://www.googleguide.com
[16]Z. Guo, M. Li, X. Wang, and A. Zhou.: Scalable XSLT Evaluation, In *Proc. of APWEB 2004*, HangZhou, China (2004) 137-150
[17]http://www.datapower.com/xmldev/xsltmark.html
[18]S. Helsen and P.Thiemann.: Polymorphic specialization for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS) archive*. Volume 26, Issue 4 (July 2004) 652-700
[19]S. Jain and R. Mahajan and D. Suciu (2002): Translating XSLT Programs to Efficient SQL Queries. *Proc. World Wide Web 200*2, Hawaii, USA (2002) 616-626
[20]N. Jones.: An Introduction to Partial Evaluation. *ACM Computing Surveys*. (1996) 28(3) 480-503
[21]M. Kay.: Saxon XSLT Processor. http://saxon.sourceforge.net/
[22]M. Kay.: Anatomy of an XSLT Processor. http://www-106.ibm.com/developerworks/library/x-xslt2/ (2001)
[23]P. Kumar.: XML Processing Measurements using XPB4J (2003)
[24]C. Laird.: XSLT powers a new wave of web. http://www.linuxjournal.com/article.php?sid=5622 (2002)
[25]D. Lee, W. Chu.: Comparative analysis of six XML schema languages. *ACM SIGMOD Record archive Volume 29, Issue 3*. ACM Press, New York, NY, USA (2000) 76–87
[26]Q. Li, B. Moon.: Indexing and querying XML data for regular path expressions. In *Proc. Int'l Conf on VLDB*, Roma, Italy (2001) 361-370
[27]S. Maneth and F. Neven.: Structured document transformations based on XSL. In *Proceedings of DBPL'99*, Kinloch Rannoch, Scottland (2000) 80-98
[28]L. Villard, N. Layaida.: An incremental XSLT transformation processor for XML document manipulation. *Proc. World Wide Web 2002*, Hawaii, USA (2002) 474-485
[29]World Wide Web Consortium. XML Path Language(XPath) Recommendation. http://www.w3.org/TR/xpath
[30]M. Weiser: Programmers use slices when debugging. In *Communications of ACM*, Volume 25, Issue 7, 446 – 452 (1982)
[31] X. Zhang, K. Dimitrova, L. Wang, M. E. Sayed, B. Murphy, B. Pielech, M Mulchandani, L. Ding and E. A. Rundensteiner.: RainbowII: multi-XQuery optimization using materialized XML views. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, San Diego, California, USA (2003) 671-685