JetXSLT: A Resource-Conscious XSLT Processor

William M. Shui[1] **Raymond K. Wong**[1,2]

[1] School of Computer Science & Engineering
University of New South Wales
Sydney, Australia

[2] National ICT Australia, Australia

Abstract

Streaming XSLT transformations is one of the goals of the XML community. However, it has proved difficult or impossible to have a streaming implementation of the full XSLT language. Many partial solutions have been developed to address this problem so far. This paper describes a more complete processor called JetXSLT - a resource-conscious XSLT processor, which uses less memory and CPU processing compared to other approaches. It processes XSLT stylesheets containing multiple template rules with only a single pass of the source XML document/stream. We also propose a novel selection and filtering mechanism catered specifically for XSLT, which can process multiple XPath selection patterns containing both structural and aggregate predicates with only a single pass of the source XML document. We present its overall design, data structures, implementation and experiments and show that it outperforms other popular alternatives.
Keywords: XSLT, XML, XSL, streaming

1 Introduction

Currently, many on-line systems use XSLT (W3C Recommendation 1999*b*) to convert their XML data into other web formats. Subsets of the source XML data are selected and transformed to a different output format. In general, an XSLT stylesheet consists of one or more template rules that contain XPath (W3C Recommendation 1999*a*) selection patterns for selecting nodes from a source XML document tree. For each template rule in an XSLT stylesheet, its selected context node tree from its source document often becomes the input source for one or more template rules. Therefore, during the processing of XSLT stylesheets, the input source document can be recursive with regards to one or more XPath selection patterns. Furthermore, each template rule in a stylesheet may produce result trees that could again, invoke other template rule based on the context of the input source XML node.

An example of the processing model of template rules in XSLT transformation is illustrated in Figures 1, where in Figure 1(c), multiple XML source nodes from Figure 1(a) are selected by different template rules from Figure 1(b) at different stages of the transformation process. In this example, the root template node *T1* (in Figure 1(b)) is always invoked first to test its selection pattern against the root element a_1 of the source document. Once a_1 is selected as the current context source node, its descendant nodes $(b_1, b_2, b_3, b_4, b_5)$ and (x_1, x_4) are then selected as the next list of context nodes (Phase 1 in Figure 1(c)). In Phase 2, the source nodes (b_1, b_2) and x_4 match the selection patterns of templates *A3* and *A4* respectively. Consequently, their descendant source nodes $(c_2, x_2), (c_7, x_4)$ and (c_7, c_8, x_7) are also selected as the context nodes for the next phase. For the other context nodes produced in phase 1 ((b_2, b_3, b_5) and x_1), they do not have any descendant nodes to match other templates in the stylesheet nor are they selected for transformation. Therefore, they are no longer needed for the next phase of processing. After repeated execution of each template rule in the stylesheet, the shaded source nodes (x_1, x_5, x_5, x_6) in Figure 1(c) are finally selected and their textual contents are then transformed according to the instructions of the XSLT stylesheet.

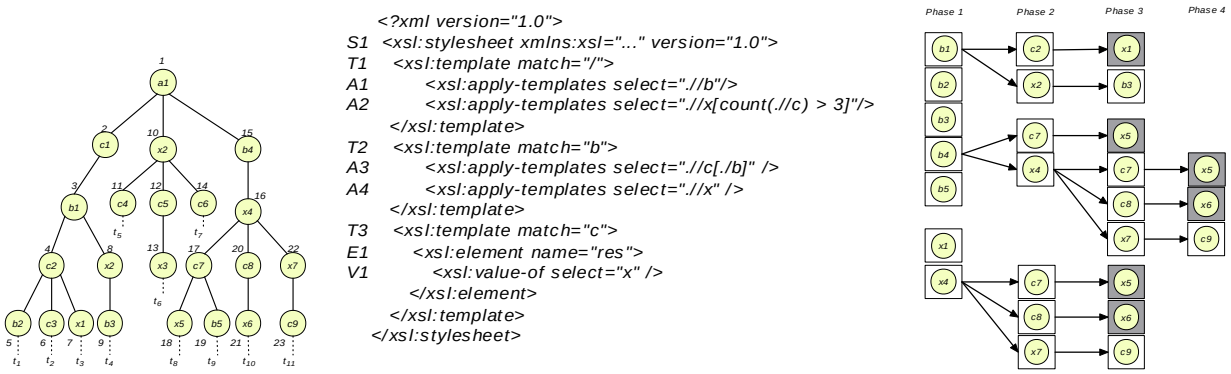
There have been some proposed methods (Moerkotte 2002, Li et al. 2003) for effective processing of XSLT. However, they are based on the assumptions that the XML data is stored in a database management system (DBMS), such that system specific indices and query optimizers can be utilised. These approaches are not suitable in situations where database systems are not accessible or data is not stored in any database. In these situations, a stand-alone XSLT processor is needed to carry out the transformation of XML data. Saxon's approach (Kay 2010) to the problem is based on using a push architecture end-to-end, to eliminate the source tree as an intermediary between push-based XML parsing/validation and pull-based XPath processing.

This paper addresses a few unresolved issues that exist with the current methods of stand-alone processing of XSLT stylesheets.

Firstly, widely used stand-alone XSLT processors: Xalan (Java/C++) and MSXML, both have high memory and processor usage. Both XSLT processors starts by loading a source document into memory and store the data using the document object model (DOM) (W3C Recommendation 2001). Each source node in the DOM tree is subsequently tested against the selection pattern of each XSLT template rule. If a source node is found to match the selection pattern of a template rule, the source node is either written to the output buffer for final transformation or used as the context node tree to test against the selection patterns of other template rules. As shown in our experiments, these processors are both memory and CPU demanding and are not scalable.

Secondly, in Figure 1, we show a snapshot of the processing of an arbitrary XSLT stylesheet (Figure 1(b)) on a sample XML data set (Figure 1(a)). In this snapshot, source nodes that match the XPath selection pattern of a template rule are selected. They are then tested again with the selection patterns of other template rules in the same stylesheet for further processing. As a result, multiple passes of source XML data are needed.

Some have proposed methods to process XPath query patterns on input XML data in streaming fashion (Josi-



(a) A sample XML document tree (b) A sample XSLT stylesheet (c) A sample output

Figure 1: Example run of processing XSLT template rules on a XML document

fovski et al. 2005, Barton et al. 2003, Bar-Yossef et al. 2004, Ludäscher et al. 2002, Ives et al. n.d., Olteanu et al. 2003, Gupta & Suciu 2003, Green et al. 2003, Li & Agrawal 2005, Lee et al. 2002). Due to the recursive nature of the XSLT processing model, where the result of one selection pattern match is used as the input for matching the selection pattern of other template rules. It is still expensive for current stream based approaches, as multiple I/O passes are often required for scanning a large input source XML document and the intermediate context source nodes.

This paper presents a new stand-alone XSLT processor called JetXSLT, which can process XSLT stylesheets with multiple template rules in a single pass of source XML data in a streaming fashion. Further, its run time is linear to the size of the source XML document.

To map the context source nodes from the source XML document to the XSLT stylesheet, a data structure called the *result tree* is proposed as part of the JetXSLT framework. Not only does it serve as an XSLT abstract syntax tree (AST), the *result tree* also serves as the index for the output result. Consequently, the final output can be recreated by a simple traversal of the *result tree*.

We provide extensive experimental results, measuring the performance of different XSLT processors on input XML documents of various sizes and structures. We also measure the performance of XSLT processing using stylesheets with varying types of template rules. Overall, our experiment results have shown the superiority of JetXSLT over existing proposals, in terms of both space and time.

The rest of this paper is organized as follows: Section 2 describes related work, Section 3 presents the overview of the workflow of JetXSLT and its key process data structures. Section 4 describes the evaluation of the AST. In Section 5, the experiment results comparing JetXSLT and other popular XSLT processors are presented and analysed; and finally, Section 6 concludes the paper.

2 Related Work

Recently, work has been done on improving the efficiency of XSLT processing, for cases where source XML datasets are stored in database systems, where auxiliary information such as indexes and selectivity are available for faster query processing. The works of Moerkotte et al (Moerkotte 2002) and Li et al (Li et al. 2003) takes the approach of pushing the processing of XSL template rule instructions into SQL queries. Liu et al (Liu & Novoselsky 2006) optimized the generation of partial results for XSLT by converting some template rules into XQuery, which can then be quickly processed by leveraging existing object indexes in the database system. The work of Boncz et al. (Boncz et al. 2006) takes a similar approach, by storing XML data and processing XQuery using relational algebra. However, the aforementioned work may not be appropriate solutions for XSLT processing on XML datasets that are randomly generated on the fly or for incoming data streams. This is due to the known performance overhead of maintaining one or more indexes for each one of the XML datasets. Furthermore, many of the XML datasets can only be accessed once for the publication of the transformed data and the index created will not be reused again.

To the best of our knowledge, the work of (Guo et al. 2004, Schott & Noga 2003, Dong & Bailey 2004, Villard & Layaïda 2002) are most related to this paper, as they also focus on the XSLT processing of un-indexed XML datasets. Guo et al (Guo et al. 2004) proposed a method of processing an XSLT stylesheet on XML documents containing DTDs. However, their approach may fail if the source document does not contain a DTD, such that a map cannot be created between the stylesheet and the DTD. Furthermore, their approach fails when the input XSLT stylesheet contains complicated XPath expressions with predicates, such as AND/OR logic and aggregate functions such as $fn : count$.

Schott et al (Schott & Noga 2003) use a lazy approach in outputting the result tree, such that the regions of the source XML document are only parsed if the corresponding XSLT result tree is accessed. Although this approach may be useful for cases where a region of a transformed result tree is accessed frequently through an application interface, it may suffer from its own laziness when the entire result tree is accessed. Furthermore, if a large XML document is shallow and broad in its structure, then the lazy approach also loses its advantage when the selection patterns contain predicates.

Other works on XSLT transformation include incremental transformation process (Villard & Layaïda 2002) and static analysis of XSLT programs (Dong & Bailey 2004). In (Villard & Layaïda 2002), Villard et al worked on the efficient maintenance of the output produced by applying an XSLT stylesheet on a source XML document, such that a change in either the source XML document or the stylesheet can be quickly reflected in the target output document. The focus of their work is quite different to ours, as the scope of our work is on optimizing the one-off instance of processing an XSLT stylesheet on an XML document. In (Dong & Bailey 2004), Dong et al focused their work on statically determining the reachability of an XSLT stylesheet on a source XML document. However, this may not apply to XML documents that have no DTD.

For processing XPath queries on streaming XML, Josifovski et al (Josifovski et al. 2005) proposed TurboXPath, which is based on their previous work (Barton et al. 2003, Bar-Yossef et al. 2004). In (Josifovski et al. 2005), they

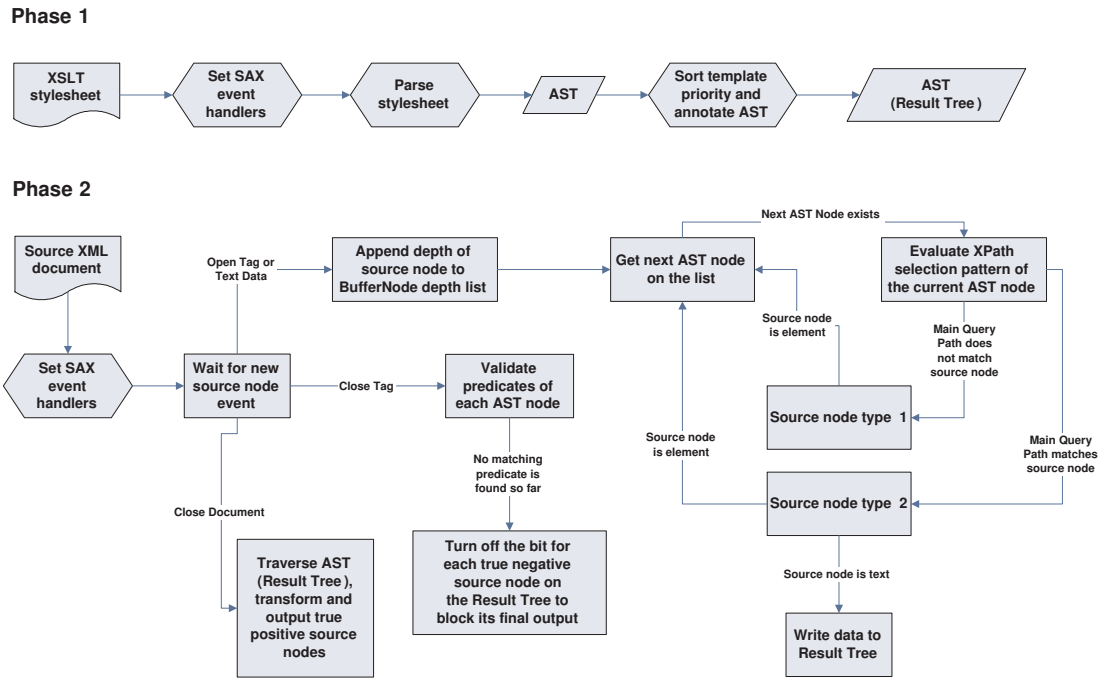


Figure 2: The overview of the work-flow of the entire JetXSLT process.

are able to process XQuery queries (both forward and backward) using only a single pass of the input XML stream, although they store temporary data in database systems before performing a merge join operation, to produce the final XPath output. Their approach also avoids the use of finite state automaton (FSA) and the creation of run-time automata states. As a result, their approach is less expensive computationally compared to the FSA style of approaches such as (Ludäscher et al. 2002, Ives et al. n.d., Olteanu et al. 2003, Gupta & Suciu 2003, Green et al. 2003, Peng & Chawathe 2003). Li et al (Li & Agrawal 2005) proposed a way to determine whether a given set of XQuery queries can be transformed so that they can be executed with a single pass on the source XML data. They also showed various optimization techniques for the transformation of XQuery queries.

Both (Josifovski et al. 2005) and (Li & Agrawal 2005) are unable to process XSL stylesheets without a major redesign of their framework. They do not consider the recursive processing model of template rules in XSLT. Furthermore, (Josifovski et al. 2005) cannot efficiently process XPath queries with aggregate predicates such as the *count* function. For Li's work (Li & Agrawal 2005), a single pass of the source data is only possible if the given queries pass their *single pass analysis*. Therefore, if the given queries fail the single pass analysis, the processor still has to resort to multiple passes of the source XML data or the materialisation of the entire input dataset.

Most recently, Saxon has been extended to support streaming (including streaming of some constructs that are not supposed to be streamable according to the new XSLT 2.1 draft)(Kay 2010). This is achieved by using a push architecture end-to-end, to eliminate the source tree as an intermediary between push-based XML parsing/validation and pull-based XPath processing.

3 Data Structures

To help readers to better understand our work, an illustration of the workflow of our XSLT processor is presented in Figure 2.

3.1 Preliminaries

This paper uses an *abstract syntax tree* (AST) to represent an XSLT stylesheet, where each XSLT rule component corresponds to an AST node. For simplicity, this paper focusses on the efficient processing of key node-selecting XSLT components that significantly affect the performance of XSLT processors, which include: *xsl:stylesheet*, *xsl:template*, *xsl:apply-templates*, *xsl:for-each*, *xsl:value-of*.

Based on the XSLT processing model (W3C Recommendation 1999*b*), we consider a run-time path R - a snapshot of the history of a source node S processed by the AST of an XSLT stylesheet. That is, if S is selected for output, then there must exist a run-time path $R = \{a_0, a_1, \ldots, a_{n-1}\}$, where each AST node $a_i \in R$ is executed consecutively to process S until S reaches the stage for final output. The symbol n denotes the total number of times S has been processed by AST nodes from R.

If the selection patterns of all AST nodes on the run-time path of a source node are concatenated to form a single XPath expression x, and t denotes the tree representation of x, then $b(t)$ denotes the root-to-leaf branch of t, where $b(t)$ contains a sequence of consecutive query nodes. If $p(x)$ denotes a predicate XPath expression in x and $p(t)$ denotes the corresponding tree fragment in t, then we define the *main query path* of t as a $b(t)$, where none of the query nodes in its root-to-leaf path is a member of $p(t)$.

We define a source node s to be a *partial* match to x, if s does not match the entire tree pattern of x and it only matches the structural requirement of the *main query path* of x. We also define s to be a *complete* match to x, if s matches the entire structural pattern of x. For example, source node a_2 Figure 3 is a *partial* match to the XPath expression $//a//b[.//e]//c[.//f]//d$, as the *main query path* is $//a//b//c//d$. However, source node a_1 is a *complete* match because its structural make up satisfies the entire query pattern tree. We also denote a source node as a *true negative* match, if it is only a *partial match* to the selection pattern of an AST node; and a *true positive* match, if it is a *complete match* to the selection pattern of an AST node.

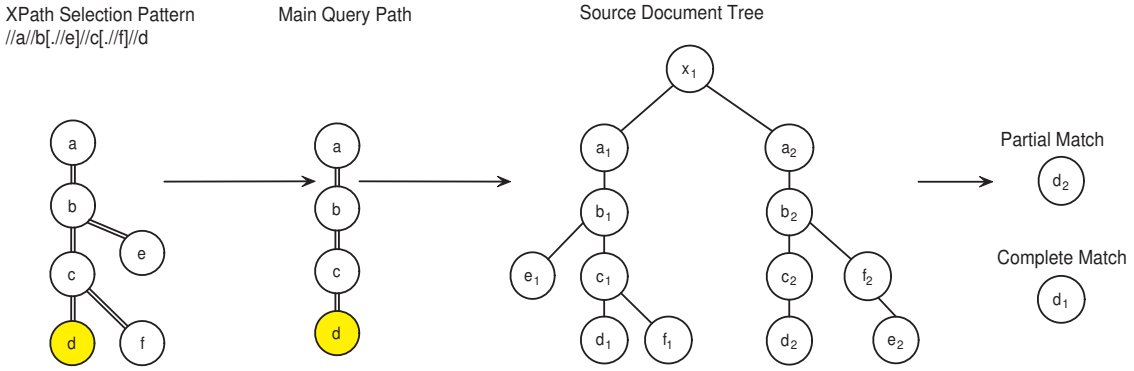


Figure 3: An example of a *partial* match and a *complete* match.

We maintain global data which is used to facilitate the correct execution of the JetXSLT processor. They are stored in a processor state monitor $pState$ and they include the following: i) $pState.depth$, denoting the current depth of the SAX parser on the source document tree; ii) $pState.process\_list$, a list of AST nodes that are stored in the order of current source node run-time path; iii) $pState.data\_pool$, the pointer to the *data pool* of the *result tree*; iv) $pState.text\_node\_stack$, a stack which stores source text nodes that are still within the scope of the SAX parser. That is, the closing tags of ancestor elements of the stored text nodes are yet to be read by the SAX parser; v) $pState.elems$, a hash-table with unique element names as keys and each key is associated to a pointer to the unique named buffer node; and vi) $pState.is\_in\_apply\_templates$, which is a logical flag indicating if the current source node has been matched to a selection pattern in an AST node of type *xsl:apply-templates*.

3.2 The Result Tree

In JetXSLT processor, an AST is created to represent an XSLT stylesheet, where each node in the AST represents an XSLT element in the stylesheet and other basic information such as pointers to its surrounding AST nodes, its XPath selection pattern and its AST node type. We partially store the matching node-sets from the source XML document on disk as part of a *result tree*, such that in terms of disk I/O, only a single pass of the source document is required. This property can be very useful in circumstances where the data source needs to be processed in streaming fashion.

Structurally, a *result tree* is an extended AST of the XSLT stylesheet. The extension includes further annotation of each AST node with extra data structures, including: i) a *data pool*; ii) *data pointer layers*; and iii) *bit arrays*. The extra annotation in the *result tree* allows our proposed XSLT processor to re-trace the run-time path of a source node, before its string-value representation is selected, transformed and written to the final output. Using the example in Figure 1, the run-time path of the final output source node t_8 is $S_1 \rightarrow T_1 \rightarrow A_1 \rightarrow T_2 \rightarrow A_3 \rightarrow T_3 \rightarrow E_1 \rightarrow V_1$, where S_1 is the root AST node and V_1 is the last processing AST node (*xsl:value-of*) before the text source node is written to the final output.

In JetXSLT, every *result tree* has a corresponding *data pool*, which is a data block (in memory or secondary) containing source text nodes identified by one or more XSLT selection patterns. The source text nodes in the *data pool* are always stored in ascending XML document order, such that a sequential scan of the *data pool* can easily project the selected text node-set in original document order. We store only source text nodes in the *data pool* in this implementation, because in XSLT the default string-value representation of XML nodes, such as elements, attributes, comments and processing instructions, are empty strings. Since a *result tree* in JetXSLT is an annotated AST with added auxiliary data structures for each node, we can also consider all nodes in a *result tree* as AST nodes. In our implementation, the annotation data structures added to each *result tree* node are i) *data pointer layer* and ii) *bit array*. The *data pointer layer* allows JetXSLT to record the run-time path of each source node that eventually reaches the *data pool*. The *bit array* indicates whether a specific run-time path is a *true positive* or a *true negative* result for the final output.

In JetXSLT, each *data pointer layer* of an AST node A consists of a sequence of pointers stored in ascending XML document order, where pointer P_i corresponds to a source node $S_i \in S$ and S is a node-set matching the selection pattern of AST node A. Each pointer of the *data pointer layer* has four attributes: i) pos; ii) $size$; iii) $IsEnd$; and iv) $next\_runtime\_ast\_node$. Attribute pos refers to the byte position of the source text node in the *data pool* and $size$ refers to the data size of the source text node. Attribute $IsEnd$ indicates whether the parent AST node of the current data pointer is the last stop of the source node run-time path. If $IsEnd$ is $true$, then we can access the *data pool* using the information provided by attributes $size$ and pos. Finally, attribute $next\_runtime\_ast\_node$ is a pointer to the next AST node on the source node run-time path. In our implementation, if $IsEnd$ is $true$, then $next\_runtime\_ast\_node$ is **null** because the parent AST node of the data pointer is the last node on the source node run-time path. However, if $IsEnd$ is $false$, then pos refers to the logical position of another data pointer, which is located in the AST node referred to by $next\_runtime\_ast\_node$.

Each bit in the *bit array* of an AST node corresponds to a pointer in the the *data pointer layer*, indicating whether or not we should visit the data pointer for the final output. If the bit value is 1, then its corresponding data pointer in the *data pointer layer* is considered readable during the final output. However, if the bit value is 0, then the data pointer is unreadable.

3.3 Result Tree Output

In Figure 2, we show that our XSLT processor only outputs the *true positive* node-sets from the *result tree* after catching the SAX event *end document* (i.e., the SAX parser has reached the end of the source XML document). The output functions are described in Algorithm 1, where the function *OutputASTRoot* initiates the process of retrieving *true positive* matches by traversing the *bit array* and the *data pointer layer* of the root AST node (i.e., *xsl:stylesheet*). Each 1 bit in the *bit array* corresponds

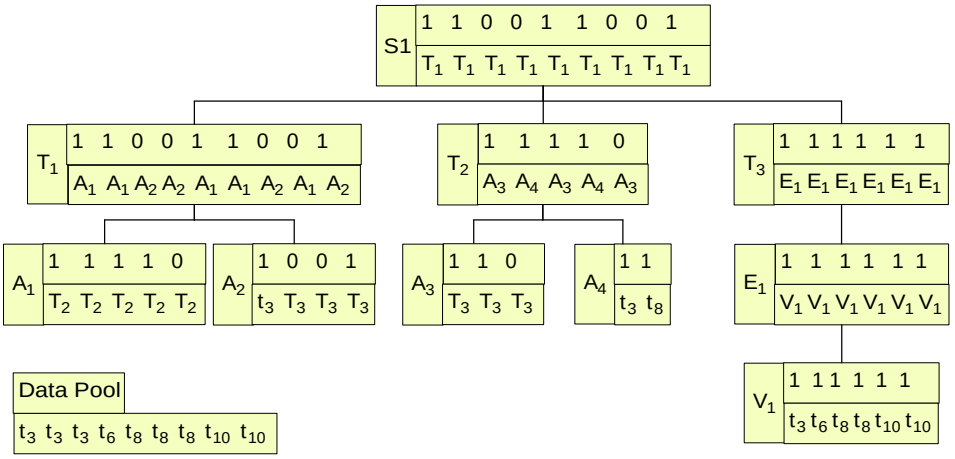


Figure 4: A *Result Tree* for the example instance in Figure 1, where *partial* and *complete* matching source node are stored in its *data pool*.

Algorithm 1 Outputting Selected Source Node Data from the Result Tree

OutputASTRoot$(node)$
1 **for each** bit $b_i \in node.BitArray[0,\ldots,n-1]$
2 **if** $(b_i = 1)$ **then**
3 *OutputDataPointer*$(node.DataPointers[i])$

OutputDataPointer(ptr)
1 **if** $(ptr.IsEnd)$ **then**
2 Output $ptr.size$ bytes of data read from the $datapool$ of the *result tree*, starting at position $ptr.pos$.
3 **else**
4 *OutputASTNode*$(ptr.next\_runtime\_ast\_node, ptr.pos)$

OutputASTNode$(ast\_node, ptrPos, outputStreamHandler)$
1 **if** $(ast\_node.BitArray[ptrPos] = 1)$ **then**
2 *OutputDataPointer*$(ast\_node.DataPointers[ptrPos])$

to a data pointer, which itself also corresponds an AST node in the run-time path. Hence, we can easily reach the *true positive* source node by following the 1 bits and data pointers along a run-time path. Similarly, a 0 bit blocks the traversal process, hence blocking the output of *true negative* matches.

In the function *OutputDataPointer*, if the end of the run-time path has been reached (i.e. $IsEnd$ is $true$), then it reads and output the corresponding *true positive* text data stored in the corresponding location in the *data pool*. However, if the end of the run-time path has not been reached, then function *OutputASTNode* is called to process the next AST node referenced by the current pointer, subsequently *OutputASTNode* will re-invoke *OutputDataPointer* for each 1 bit it encounters.

4 XSLT AST Evaluation

The previous section presented the data structure *result tree*[1], which allows the storage of selected node-sets (both *true positive* and *true negative* matches) from the source XML document. In this section, we focus on the processing of the stylesheet AST for selecting and storing source nodes to the *result tree* before the final output is generated.

4.1 The SAX Event Handlers

In JetXSLT, we initiate the processor instance by first creating an AST from the input XSLT stylesheet, this is followed by sorting all *xsl:template* nodes in descending processing priority (as specified in the recommendation (W3C Recommendation 1999*b*)). The processor then sets its core callback functions as SAX event handlers for the parser (i.e. the *open tag*, *close tag* and *text data* events). Finally, the parser reads the source XML dataset, which commences the node selection and transformation process.

Similar to the work of Josifovski et al. (Josifovski et al. 2005), JetXSLT also creates named buffer nodes to store XML elements with the same names and text data nodes. The stack in each buffer node records the document depths of source nodes that are read by the SAX parser. For SAX events *open tag* and *text data*, we record the encountering of source nodes by pushing the current depth $pState.depth$ to the stacks of the corresponding buffer nodes (named and wild-card), followed by invoking the function *Eval* (Algorithm 2) for further processing of the source node. For the SAX event *close tag*, we remove the depth stored in the stack of the corresponding buffer nodes (named and wild-card), followed by decrementing $pState.depth$ and invoking function *CloseTag* (Algorithm **??**) for restoring some run-time states.

It should be noted that the difference between JetXSLT and other approaches is in the functions *Eval* and *CloseTag*, where the internal processing of source nodes and run-time information begins. Furthermore, throughout the JetXSLT instance, we only instantiate and maintain named buffer nodes that have the same element names that are present in one or more XPath selection patterns in the stylesheet. Thus, for the document tree in Figure 1(a) and based on the XSLT stylesheet in Figure 1(b), we only need to maintain named nodes b, c and x, because neither a nor wild-card $(*)$ is needed to match any source node. This approach reduces the memory required to maintain internal buffer nodes for processing an XSLT stylesheet, as a source document may have a large set of unique element names (e.g., the TreeBank (Marcus et al. 1993) dataset).

4.2 Implementing the XSLT Processing Model

Figure 2 shows the main algorithm for the JetXSLT process. In phase 2 of Figure 2, we can summarize the execution order as: i) parse the source document, select *partial* or *complete* matching source nodes and store them in the *result tree*; ii) filter out source nodes that are *true negative* matches from the *result tree*; and iii) output the matching *true positive* source nodes from the *result tree* for final transformation. For the above phase 2 steps, functions in Algorithm 2 and Algorithm 3 are the first points

[1]The terms AST and the Result Tree will be used interchangeably for the rest of this paper, because the Result Tree is the AST containing temporarily selected source nodes before the final output

of access.

Algorithm 2 Function for processing the stylesheet AST

```
//predicate_counter exists for every instance of AST node
//and it is set to 0 at the time of the instance creation.
Eval(ast_node, isText, pState)
1   if (ast_node is xsl:stylesheet) then
2       Set the flag last_visited for all buffer nodes to 0.
3       pState.process_list.clear()
4       pState.is_in_apply_templates ← false
5       pState.process_list.append(ast_node)
6       EvalStylesheet(isText, pState)
7       return true
8   else if (ast_node is xsl:for-each or xsl:template) then
9       if (EvalXPathExpr(ast_node.xpath, ast_node, isText, pState)) then
10          pState.process_list.append(ast_node)
11          pState.is_in_apply_templates ← false
12          for each (n in ast_node.child_nodes) do
13              if (Eval(n, isText, pState)) then
14                  if (n.xpath.hasPredicateExpr ∧ isText) then
15                      IncrementByOne(ast_node.predicate_counter)
16          return true
17  else if (ast_node is xsl:apply-templates) then
18      if (EvalXPathExpr(ast_node.xpath, ast_node, isText, pState)) then
19          pState.process_list.append(ast_node)
20          pState.is_in_apply_templates ← true
21          EvalStylesheet(AST.root, isText, pState)
22          return true
23  else if (ast_node is xsl:value-of) then
24      if (EvalXPathExpr(ast_node.xpath, ast_node, isText, pState)) then
25          pState.process_list.append(ast_node)
26          if (n.xpath.hasPredicateExpr ∧ isText) then
27              IncrementByOne(ast_node.predicate_counter)
28          Let ptr be a pointer data structure.
29          WriteToDataPool(ast_node, ptr, pState)
30          UpdateProcessList(ptr, ast_node)
31          return true
32  return false
```

In Algorithm 2, when a new source node is read by the SAX parser, function *Eval* traverses the XSLT stylesheet and attempts to find an AST node with a selection pattern that matches the current tree structure of the source node. It returns $true$ if a *partial* or a *complete* match for ast_node is found. The order in which JetXSLT iterates through the XSLT AST can be split into four different blocks, which respectively correspond to AST nodes of types *xsl:stylesheet*, *xsl:template* or *xsl:for-each*, *xsl:apply-templates* and *xsl:value-of*. Each block is separated by lines separated by lines 1 (block 1), 8 (block 2), 17 (block 3), 23 (block 4), and it is only executed in JetXSLT if the parameter ast_node has the same node type as the corresponding node type of the code block.

AST node type *xsl:stylesheet*: in the function *Eval* (Algorithm 2), block 1 is executed to reset global variables in $pState$. Next, ast_node is appended to the run-time path $process\_list$, as the first AST node to have processed the current source node. This is followed by the evaluation of the stylesheet node in function *EvalStylesheet* (in Algorithm 3), which recursively executes *Eval* to process each child *xsl:template* node in the stylesheet (lines 1 - 4). In lines 5 - 8 of the function *EvalStylesheet*, the text data of matching source nodes (*partial* or *complete*) are also written to the *result tree* (*WriteToDataPool*) and their corresponding run-time paths are updated (*UpdateProcessList*).

AST node types *xsl:template* and *xsl:for-each*: in the function *Eval*, block 2 is executed to evaluate AST nodes of these two types. This is because the node selection action of both *xsl:template* and *xsl:for-each* are the same and unlike *xsl:apply-templates*, no recursive process is required. The code block first evaluates the XPath selection pattern of the AST node and if a *partial* or a *complete* match is found, then the AST node is appended to the end of the run-time path $process\_list$. The flag $is\_in\_apply\_templates$ in $pState$ is also set to $false$, indicating the AST node being evaluated is not of type *xsl:apply-templates*. This evaluation process is also propagated to the child nodes of ast_node in a cascading fashion, emulating the XSLT processing model in the XSLT specification.

Algorithm 3 Auxiliary function for processing the stylesheet AST

```
EvalStylesheet(isText, pState)
1   for each (n in AST.root.sorted_child_nodes) do
2       if (Eval(n, isText, pState)) then
3           if (n.xpath.hasPredicateExpr ∧ isText) then
4               IncrementByOne(ast_node.predicate_count)
5   if (pState.is_in_apply_templates ∧ isText∧
    pState.process_list.last is an apply-templates node) then
6       Pointer ptr
7       WriteToDataPool(ast_node, ptr, pState)
8       UpdateProcessList(ptr, pState)
```

AST node type *xsl:apply-templates*: in any instance of processing an XSLT stylesheet, the XSLT processing model specifies that if a source node matches the selection pattern of a *xsl:apply-templates* node, then the source node and its descendants are re-processed by every template in the XSLT stylesheet for a possible match. Therefore, in block 3 (lines 17 - 22) of function *Eval*, if a source node is found to *partially* or *completely* match the selection pattern of ast_node, which is of type *xsl:apply-templates*, then the global variable $pState.is\_in\_apply\_templates$ is set to $true$ and function *EvalStylesheet* is invoked to re-evaluate the source node and its descendants. This links functions *Eval* and *EvalStylesheet* together to form the recursive processing mechanism of an XSLT stylesheet.

AST node type *xsl:value-of*: in the function *Eval*, instruction block 4 is executed if ast_node is of type *xsl:value-of*, which evaluates the selection pattern of ast_node against the current new source node. If *EvalXPathExpr* returns $true$, then the new source node may be a *partial* or *complete* match to the selection pattern of ast_node. Consequently, its data is written to the *data pool* by *WriteToDataPool* and its run-time path is updated by *UpdateProcessList*. It should be noted that, unlike the *xsl:apply-templates* AST node, the specification of *xsl:value-of* AST node does not require re-processing of the same source node. Hence, the function *Eval* does not need to be invoked in this case (lines 28 - 30).

Astute readers may have noticed that we have only focussed on the projection of source text nodes for the final output, ignoring the non-selecting AST nodes such as *xsl:element*, *xsl:attribute* and so on. This is because the XSLT recommendation (W3C Recommendation 1999*b*) only specifies the output instructions for text source nodes selected by AST nodes of type *xsl:value-of* or *xsl:apply-templates*. Run-time non-selecting nodes such as <*xsl:element*> and <*xsl:attribute*> can be easily implemented by generating the output data on the fly without the need to store them in the *result tree*. Therefore, they are not considered as the bottleneck for the overall performance of XSLT processing.

4.3 Building the Result Tree

In order to store the text data of a source node to the *result tree*, our implementation of JetXSLT requires that two conditions in function *EvalStylesheet* are satisfied: i) the tail of the run-time path ($pState.process\_list$) is an AST node of type *xsl:apply-templates*; and ii) the source node is either a *partial* or *complete* match to the selection pattern of the AST node. If a source node is found to be a match, then it is written to the *result tree* through functions *WriteToDataPool* and *UpdateProcessList*.

In Algorithm 2, the function *UpdateProcessList* is a helper function. For each AST node n in the run-time path $pState.process\_list$, the function appends the pointer ptr to the *data pointer layer* of n. Next, the four attributes

of the data pointer $n.data\_pointer\_layer.last$ are updated and a new bit 1 is appended to its *bit layer*. The second helper function *WriteToDataPool* writes the data from the top of the stack of the text buffer node to the *data pool*. Next, the same actions as the function *UpdateProcessList* are executed (i.e. adding new corresponding pointers and updating *data pointer layers* of each node in the run-time path). By using these helper functions, the data written to the *result tree* can later be accessed and traversed by the output functions in Algorithm 1 in only a single pass.

5 Performance Evaluation

In this section, we present the experiment results on the performance of JetXSLT and some widely used XSLT processors. We measure the performance of each XSLT processor based on the size of input source data (in XML), the amount of physical memory and the total percentage of processor used for each document transformation. We also show the performance of JetXSLT when different sized memory buffers are used, which provides insights into possible further optimization of our XSLT processor.

5.1 Experiment Setup

Our experiments compared the performance of our proposed XSLT processor with the popular Xalan-C XSLT processor(*The Apache Xalan Project* n.d.) and the Microsoft XML library 6.0 (*Microsoft XML Core Services* n.d.). All executables were compiled using Visual Studio .Net 2003 and all were implemented in C++. Both JetXSLT and Xalan-C also used the same Xerces-C(*The Apache Xerces Project* n.d.) C++ code base for parsing XML documents.

The test machine was a PC, equipped with an AMD Athlon 3200+ 2.20GHz processor, 1.5 GB of physical DDR RAM, 250GB 7200 RPM SATA HDD and Windows XP. The raw XML datasets used in the experiments are: the Shakespeare(*Shakespeare Dataset* n.d.) dataset and the TreeBank(Marcus et al. 1993) dataset. The tree representation of the two datasets differ significantly, where Shakespeare dataset is a regular structured XML document and TreeBank is a randomly structured, deeply recursive XML document. Nevertheless, these two types of XML documents compliment each other and we believe that they are adequate as the test cases for most types of XML documents used in the real world today. For larger sized input documents, we simply concatenated the originals of Shakespeare and TreeBank documents respectively, until the desired document size was achieved. In our experiments, we did not use the DBLP dataset, as the depth of its document tree is only 3 levels for a majority of its records. We believe this dataset is not suitable for testing the scalability of XSLT processors, where the observation of the performance of each template selecting context node-set from the output of another is the main focus.

The XSLT stylesheets used in our experiments are listed in Table 1 for both the Shakespeare and TreeBank data sets. Each test XSLT stylesheet in Table 1 consists of one or more XSLT components listed in Tables 2. In these tables, all of the T symbols represent *xsl:template* nodes, the A symbols represent *xsl:apply-templates* nodes, the F symbols represent *xsl:for-each* nodes and the V symbols represent *xsl:value-of* nodes. In Table 1, stylesheet Q11 for the Shakespeare dataset can be read as the following XSLT stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet ... >
  <xsl:template match="/">
    <xsl:apply-templates select=".//TITLE" />
    <xsl:apply-templates select=".//PGROUP" />
    <xsl:apply-templates select=".//SPEECH" />
  </xsl:template>
  <xsl:template match="PGROUP">
    <xsl:apply-templates select=".//PERSONA" />
  </xsl:template>
  <xsl:template match="SPEECH">
    <xsl:apply-templates select=".//LINE" />
  </xsl:template>
</xsl:stylesheet>
```

5.2 Run-time Comparison

In Figure 5, the run-time of each participating XSLT processor for processing each stylesheet for both Shakespeare and TreeBank datasets are presented. To eliminate the cold-cache effect, each experiment was performed multiple times and the average is presented in the final result. Overall, JetXSLT and Xalan-C have outperformed MSXML on average run-time for both Shakespeare and TreeBank datasets.

Our experiment results show that our approach significantly outperforms Xalan-C and MSXML when processing stylesheets Q5, Q6, Q9, Q10 and Q11 for the Shakespeare dataset; and stylesheets Q1, Q4, Q6 and Q7 for the TreeBank data set. The common characteristic shared by these two groups of test stylesheets is: they both contain complicated predicate conditions such as aggregate count function calls or text match. Some of the test stylesheets even involve multiple selection patterns containing predicate conditions. The speed improvement of our processor is even more obvious when larger datasets are used.

The increased performance of our approach can be attributed to the reduced number of iterations of the source XML document at run-time. JetXSLT only scans the source document once through the SAX interface, while storing both *true positive* and *true negative* nodes to the *result tree* on the fly. All *true negative* nodes are blocked from output by flipping their corresponding bits in the *bit layers* in the *result tree*. Consequently, only *true positive* source nodes are written to the final output.

5.3 Effectiveness of Varying Memory Buffers

In our implementation of JetXSLT, a pool of buffer pages are evenly allocated to the *bit layer* and the *pointer layer* of each AST node in the *result tree*. By default, the total buffer pool size is set to 8MB. We only allocate 128KB of read buffer for retrieving the actual text data from the *data pool* of the *result tree*. Quite often, a large segment of text data on the *data pool* may belong to a set of *true negative* source nodes. Therefore, we can simply skip the reading of the data during the final output process.

Using this buffer pool configuration, we have conducted further experiments. We found that if the allocated buffer size for JetXSLT is greater than the actual size of the source datasets, there is no significant performance gain. This may be attributed to the following factors:

1. Low selectivity of the *true positive* source nodes, causing an increase in read accesses. This occurs when the total number of *true positive* source nodes returned is high. Thus, increasing the number of disk read accesses to the *data pool* for retrieving the text data. This also increases the frequency of accessing the *bit layer* and the *pointer layer* of each AST node on the run-time path of each *true positive* source node.

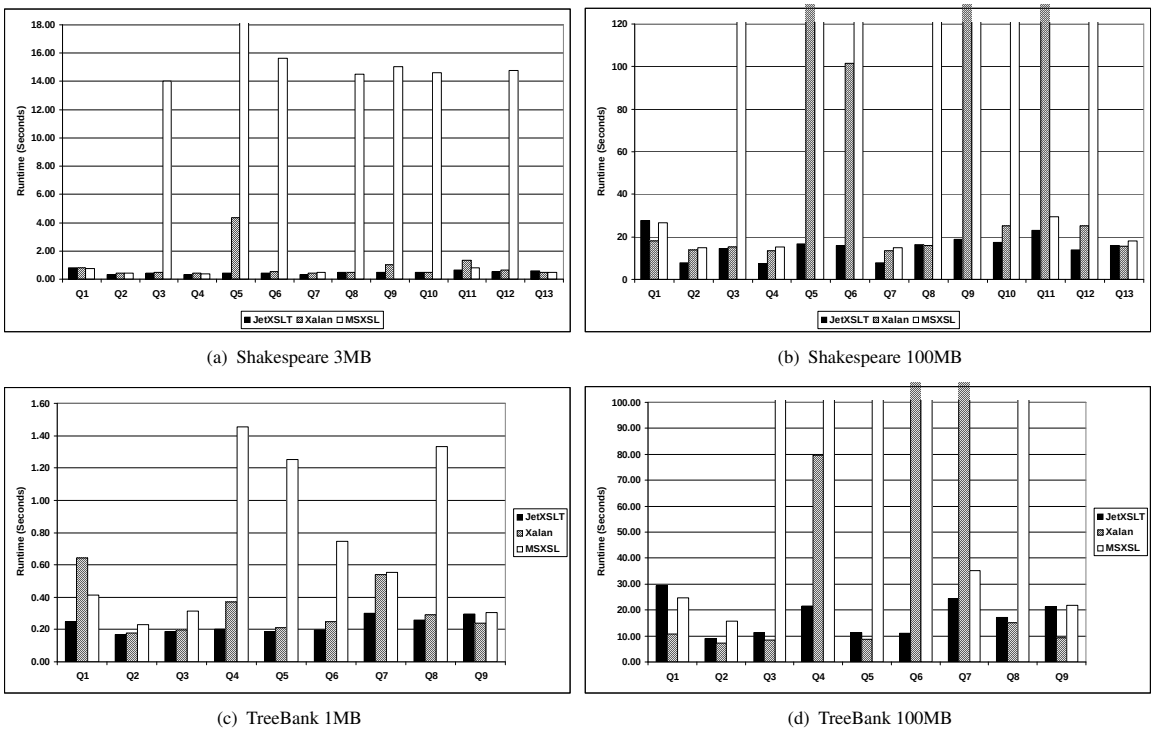
2. Context switching which causes re-flushing of large memory blocks. As more AST nodes are accessed during the output, the buffer for each AST node also needs to read and flush the *bit layer* and the *pointer layer*. Although the total number of *true positive* nodes are the same across all buffer sizes, it should

Shakespeare	XSLT Nodes	Treebank	XSLT Nodes
Q1	<T1><A1/></T1>	Q1	<T1><A1/></T1>
Q2	<T1><A2/></T1>	Q2	<T1><A15/></T1>
Q3	<T1><A3/></T1>	Q3	<T1><A16/></T1>
Q4	<T1><A4/></T1>	Q4	<T1><A17/></T1>
Q5	<T1><A5/></T1>	Q5	<T1><A18/></T1>
Q6	<T1><A6/></T1>	Q6	<T1><F7><A19/></F7></T1>
Q7	<T1><F1><A2/></F1></T1>	Q7	<T1><A20/><A21/><A22/></T1> <T6><A23/></T6> <T7><A19/></T7>
Q8	<T1><F2><V1/></F2></T1>	Q8	<T1><A24/><A25/><A26/></T1>
Q9	<T1><F3><A7/></F3></T1>	Q9	<T1><A1/></T1> <T6><F8><A19/></F8></F6> <T8><F9><A27/></F9></T8>
Q10	<T1><F4><A7/></F4></T1>		
Q11	<T1><A2/><A8/><A9/></T1> <T2><A10/></T2> <T3><A7/></T3>		
Q12	<T1><A11/><A12/><A13/></T1>		
Q13	<T1><A1/><T1/> <T4><F5><A14/></F5></T4> <T5><F6><A14/></F6></T5>		

Table 1: XSLT stylesheets used on Shakespeare (left) and XSLT stylesheets used on Treebank (right)

XSLT NODE TYPE: <xsl:template>			
Node	**XPath Query Pattern**	**Node**	**XPath Query Pattern**
T1	/	T5	PERSONAE
T2	PGROUP	T6	SBAR
T3	SPEECH	T7	S
T4	ACT	T8	NP
XSLT NODE TYPE: <xsl:apply-templates>			
Node	**XPath Query Pattern**	**Node**	**XPath Query Pattern**
A1	empty, default apply-templates	A15	.//NN//NP
A2	.//TITLE	A16	.//VP[count(.//NN) > 100]//JJ
A3	.//SPEECH[count(.//SPEAKER) > 10]//LINE	A17	.//NP[.//NN]
A4	.//PERSONAE//PGROUP/PERSONA	A18	.//NP[.//NN/text() = MuSOrOyQfWAD+wLGsXN6UK==]//DT
A5	.//SPEECH[.//SPEAKER]//LINE	A19	.//NN
A6	.//SPEECH[.//SPEAKER/text() = "COUNTESS"]//LINE	A20	.//NP
A7	.//LINE	A21	.//S
A8	.//PGROUP	A22	.//SBAR
A9	.//SPEECH	A23	.//DT
A10	.//PERSONA	A24	.//NP[count(.//NN) > 2]//DT
A11	.//SPEECH[count(.//LINE) > 10]//SPEAKER	A25	.//S[count(.//VBP) > 1]//NN
A12	.//PERSONAE[count(.//PGROUP) > 2]//PERSONA	A26	.//SBAR[count(.//NP) > 2]//VBG
A13	.//ACT[count(.//SPEECH) > 2]/TITLE	A27	.//VBG
A14	.//SPEAKER		
XSLT NODE TYPE: <xsl:for-each>			
Node	**XPath Query Pattern**	**Node**	**XPath Query Pattern**
F1	.//PERSONAE	F6	.//PGROUP[count(.//PERSONA) = 2]
F2	.//SPEECH[count(.//SPEAKER) > 10]	F7	.//VP//NP
F3	.//SPEECH[.//SPEAKER]	F8	.//NP[count(.//NNP) > 1]
F4	.//SPEECH[.//SPEAKER/text() = "COUNTESS"]	F9	.//NP[.//NN]
F5	.//SPEECH[count(.//LINE) > 10]		
XSLT NODE TYPE: <xsl:value-of>			
Node	**XPath Query Pattern**	**Node**	**XPath Query Pattern**
V1	.//LINE		

Table 2: XSLT nodes used for constructing all of the test-case XSLT stylesheets



(a) Shakespeare 3MB

(b) Shakespeare 100MB

(c) TreeBank 1MB

(d) TreeBank 100MB

Figure 5: Run-time Comparison of JetXSLT vs Xalan and MSXML using Shakespeare and TreeBank Datasets

be noted that the total size of memory being read and flushed per run-time path is different. Ultimately, as the number of run-time paths increases, more re-flushing of buffer is required, which can have extra overhead in allocating and freeing the memory during the paging process.

However, for larger datasets such as for the 100MB files, JetXSLT with a larger buffer size significantly increases its performance, which is comparable to that of Xalan and MSXML, if not outperforming them.

5.4 Memory and CPU Usage

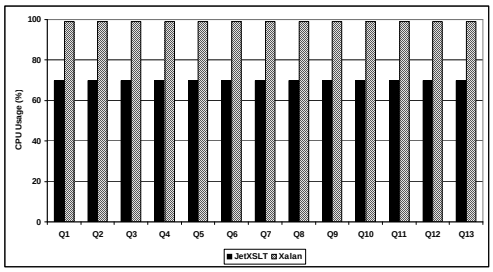


Figure 6: Maximum percentage of CPU consumption during run-time

In our experiments so far, we have shown that Xalan-C has outperformed MSXML in run-time performance. Therefore, we omit the results for MSXML for comparing CPU and memory usage, given that it uses significantly more memory than Xalan-C and it has the same percentage of CPU usage as Xalan-C, because both processors are main memory based processors. In Figure 6, we show that the percentage of CPU usage by Xalan-C is approximately 100%, whereas JetXSLT uses only approximately 70%. This is due to the disk based framework of JetXSLT, which outputs by reading from text data from the *data pool* of the *result tree*. For a busy server, this behaviour can be quite beneficial in terms of scalability, because the spare CPU cycles can be used for other processes.

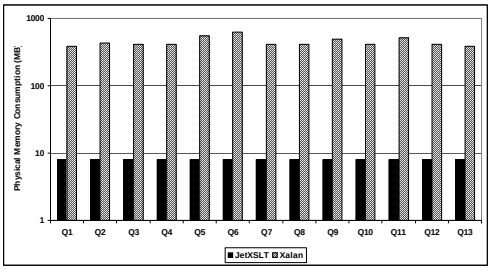


Figure 7: Maximum physical memory consumption during run-time

In Figure 7, we present the memory usage of JetXSLT and Xalan-C. It is clear that our approach uses orders of magnitude less memory than Xalan-C. This is due to the *result tree*, which is crucial for storing temporary *true negative* and *true positive* source nodes. Consequently, this reduces the total memory requirement for JetXSLT in processing XSLT stylesheets.

6 Conclusion

In this paper, we have presented a novel approach in processing XSLT stylesheets, where recursive processing template rules frequently occur. We have demonstrated that our prototype (JetXSLT) effectively outperforms major stand-alone XSLT processors in terms of run-time, CPU usage and memory consumption. Further, our design only requires a single pass of the input source XML document, which makes it ideal for processing streaming XML data that require XSLT transformation. We have demonstrated that our approach in processing complicated XSLT stylesheets addresses the scalability issues that currently exist in both enterprise and open source web servers, browsers or any application that is dependent on XML and XSLT technologies.

References

Bar-Yossef, Z., Fontoura, M. & Josifovski, V. (2004), On the Memory Requirements of XPath Evaluation over XML Streams., *in* 'PODS', pp. 177–188.

Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M. & Josifovski, V. (2003), Streaming XPath Processing with Forward and Backward Axes, *in* 'ICDE', pp. 455–466.

Boncz, P. A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J. & Teubner, J. (2006), MonetDB/XQuery: a fast XQuery processor powered by a relational engine., *in* 'SIGMOD Conference', pp. 479–490.

Dong, C. & Bailey, J. (2004), Static Analysis of XSLT Programs, *in* 'ADC', pp. 151–160.

Green, T. J., Miklau, G., Onizuka, M. & Suciu, D. (2003), Processing XML Streams with Deterministic Automata., *in* 'ICDT', pp. 173–189.

Guo, Z., Li, M., Wang, X. & Zhou, A. (2004), Scalable XSLT Evaluation., *in* 'APWeb', pp. 190–200.

Gupta, A. K. & Suciu, D. (2003), Stream Processing of XPath Queries with Predicates, *in* 'SIGMOD Conference', pp. 419–430.

Ives, Z., Levy, A. & Weld, D. (n.d.), 'Efficient evaluation of regular path expressions on streaming XML data'. Univ. of Washington Tech. Rep. CSE000502.

Josifovski, V., Fontoura, M. & Barta, A. (2005), 'Querying XML streams.', *VLDB J.* **14**(2), 197–210.

Kay, M. (2010), A Streaming XSLT Processor, *in* 'Proceedings of Balisage: The Markup Conference 2010', Vol. 5.

Lee, M.-L., Chua, B. C., Hsu, W. & Tan, K.-L. (2002), Efficient evaluation of multiple queries on streaming XML data., *in* 'CIKM', pp. 118–125.

Li, C., Bohannon, P., Korth, H. F. & Narayan, P. P. S. (2003), Composing XSL Transformations with XML Publishing Views, *in* 'SIGMOD Conference', pp. 515–526.

Li, X. & Agrawal, G. (2005), Efficient Evaluation of XQuery over Streaming Data, *in* 'VLDB', ACM, pp. 265–276.

Liu, Z. H. & Novoselsky, A. (2006), Efficient XSLT processing in relational database system., *in* 'VLDB', VLDB Endowment, pp. 1106–1116.

Ludäscher, B., Mukhopadhyay, P. & Papakonstantinou, Y. (2002), A Transducer-Based XML Query Processor., *in* 'VLDB', pp. 227–238.

Marcus, M. P., Santorini, B. & Marcinkiewicz, M. A. (1993), 'Building a large annotated corpus of English: the Penn Treebank', *Computational Linguistics* **19**.

Microsoft XML Core Services (n.d.). http://msdn2.microsoft.com/en-us/library//ms763742.aspx.

Moerkotte, G. (2002), Incorporating XSL Processing into Database Engines, *in* 'VLDB', pp. 107–118.

Olteanu, D., Kiesling, T. & Bry, F. (2003), An Evaluation of Regular Path Expressions with Qualifiers against XML Streams., *in* 'ICDE', pp. 702–704.

Peng, F. & Chawathe, S. S. (2003), XPath Queries on Streaming Data, *in* 'SIGMOD Conference', pp. 431–442.

Schott, S. & Noga, M. L. (2003), Lazy XSL transformations., *in* 'ACM Symposium on Document Engineering', pp. 9–18.

Shakespeare Dataset (n.d.). http://www.ibiblio.org/xml/examples/shakespeare/.

The Apache Xalan Project (n.d.). http://xalan.apache.org/.

The Apache Xerces Project (n.d.). http://xerces.apache.org/.

Villard, L. & Layaïda, N. (2002), An incremental XSLT transformation processor for XML document manipulation., *in* 'WWW', pp. 474–485.

W3C Recommendation (1999*a*), 'XML Path Language (XPath) Version 1.0', *http://www.w3.org/TR/xpath* .

W3C Recommendation (1999*b*), 'XSL Transformations (XSLT) Version 1.0', *http://www.w3.org/TR/xslt* .

W3C Recommendation (2001), 'Document Object Model (DOM) Level 3 Core Specification.', *http://www.w3.org/TR/2003/WD-DOM-Level-3-Core-20030609/* .