Optimized Reprocessing of Documents Using Stored Processor State

James A. Ollis David F. Brailsford Steven R. Bagley

Document Engineering Laboratory
School of Computer Science
University of Nottingham
Nottingham NG8 1BB, UK
{jao,dfb,srb}@cs.nott.ac.uk

ABSTRACT

Variable Data Printing (VDP) allows customised versions of material such as advertising flyers to be readily produced. However, VDP is often extremely demanding of computing resources because, even when much of the material stays invariant from one document instance to the next, it is often simpler to re-evaluate the page completely rather than identifying just the portions that vary.

In this paper we explore, in an XML/XSLT/SVG workflow and in an editing context, the reduction of the processing burden that can be realised by selectively reprocessing only the variant parts of the document. We introduce a method of partial re-evaluation that relies on re-engineering an existing XSLT parser to handle, at each XML tree node, both the storage and restoration of state for the underlying document processing framework. Quantitative results are presented for the magnitude of the speed-ups that can be achieved.

We also consider how changes made through an appearance-based interactive editing scheme for VDP documents can be automatically reflected in the document view via optimised XSLT re-evaluation of sub-trees that are affected either by the changed script or by altered data.

Categories and Subject Descriptors

E.1 [Data]: Data Structures — *Trees*; I.7.2 [Document and Text Processing]: Document Preparation — *Markup languages*; I.7.4 [Document and Text Processing]: Electronic Publishing.

General Terms

Languages, Documentation

Keywords

XSLT, SVG, VDP, variable data documents, document editing, document authoring, partial re-evaluation.

1. INTRODUCTION

A previous paper [1] has introduced the idea of preparing material for Variable Data Printing (VDP) using an XML-based workflow starting with a document expressed in Document Description Format (DDF) and then transforming it using XSLT. With the help of an XSLT-based layout engine, a final version of each printable instance is generated, as SVG output.

In a further paper [2] there is a description of how an interactive editor interface can be superposed on top of this underlying XSLT-based model. However, in doing this, the changes signalled in the interactive editor are fed back into an unchanged, 'under the hood', transformation chain. The whole page is then recomputed, including all the parts of it that are completely unaffected by the editing process. At this stage the performance penalties of this approach begin to be felt.

Clearly, things could be speeded up by recomputing just those items on a page that are actually affected by a given edit. This in turn leads to a consideration of the suitability of XML and XSLT for isolating exactly what needs to be done and ensuring that any recomputation is optimised, and free from side-effects.

For the purposes of this work, we have concentrated on a simplified, yet representative, document workflow as shown in figure 1.1.

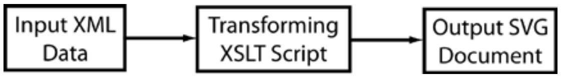

Fig 1.1 – Example document processing workflow

2. DOCUMENT AUTHORING TOOLS

The major problem when authoring a variable data document is that of providing the author with a good representation of a 'typical' document instance, in circumstances where, by the very nature of VDP, a wide range of variants will eventually have to be produced.

Some tools, such as uDirect [3] and CatBase [4] work on the premise of providing a placeholder component that can be included in the design of the document, which is then later replaced with the variable content when the set of result documents is generated.

To provide the author with a complete view of the final documents, without waiting for the full set to be produced, some of the tools provide functionality for generating a limited set of preview documents using sample data. Although this is an improvement over simply displaying a view full of placeholders, the sample document set still takes time to produce and, without generating an unfeasibly large number of instances, it cannot convey the full diversity of what might be produced.

The authoring tool produced by Lumley *et al* [2], tackles the problem of placeholder components by working directly on a result instance. This approach provides the author with a complete

FINAL DRAFT of Short Paper accepted for:
ACM *DocEng'10*, September 21–24, 2010, Manchester, United Kingdom.

view of the document, but there is still little to show the effects of the variable content on the rest of the document and, as already noted, the document instance needs to be fully regenerated after every edit made to the document.

We propose an alternative authoring process where the author is presented with an interactive instance of the document in which each variable component can be separately modified to show different representations based upon alternative variable data. This allows the author to explore the effects of the variable data on the document, while still providing a complete instance on which to work.

We now discuss the ways in which an implementation of partial re-evaluation can speed up this kind of editing model, thereby reducing the waiting time for the newly edited instance to be displayed.

3. REPROCESSING THE DOCUMENT

If a document is fully reprocessed after every edit the amount of processing depends on the total size and complexity of the page. Our aim is to break this coupling and to make the amount of reprocessing depend only on the complexity of the *edit being performed*.

3.1. Partial Re-Evaluation

The idea of implementing lazy processing for XML-based documents, and particularly for laziness in the XSLT processing itself, has been investigated by Noga et al. [5,6]. In their workflows it is the case that only part of the output from a transformation might be required and so processing was triggered in a demand-driven way whenever a given XML node was accessed. The work by Villard and Layaïda [7] was similar in spirit. They analysed a set of changes to be made to a document, as a result of data values being altered, and tried to calculate the necessary recomputation of the XSLT script in order to effect those changes.

In our case we have to go beyond implementing laziness simply for a 'one off' usage, either to generate part of a document or to implement a single set of changes. For us the edit/re-process iteration will be activated many times during editing and since the editor needs a standard document instance to work on it also follows that it has to be initialised with a fully calculated document.

Since the XSLT script in our example workflow produces the resulting SVG document by binding in the content of the variable data file, each component (images, text etc.) in the SVG is generated directly from a given node in the XSLT script. This mapping allows us the possibility of regenerating selected components in the result SVG document by re-executing specific instructions in the XSLT script. However, simply knowing which instructions to re-execute is not sufficient to allow us to regenerate the required component(s). Each instruction in the stylesheet is executed with the processor in a particular state, which is dependent on the execution of previous instructions. It is therefore essential for this state to be set up correctly at every tree node, before attempting any partial re-evaluation of the document.

4. STORING PROCESSOR STATE

In order to restore the state of the underlying XSLT processor at a given point, such that we can repeat the execution of part of the stylesheet, we must first acquire and store the relevant state during the initial execution. The state of the processor comprises the following pieces of information:

- The current mode
- The current context node
- The current position
- The names and values of any variables and parameters that are in scope

To link this information to the related point in the XSLT stylesheet we also need to produce a reference to the associated instruction that was executed.

There are potentially two ways of retaining this state information during the execution of the stylesheet: either modify the stylesheet to generate the state information as extra XML elements/attributes, or modify the XSLT processor so that it produces the information separately to the output.

4.1. Modifying the Stylesheet

The idea of modifying an XSLT stylesheet to handle the storage of associated information within the document tree itself has been investigated in previous work [8]. We researched the possibilities of storing the extra information either as extra elements, or as additional attributes added to existing elements.

In summary, the 'additional attributes' approach worked particularly well for the investigations being undertaken at the time but we could not adopt it in the present researches owing to its fragility in the unlikely, but perfectly possible, event that XSLT state became dependent on the precise number of attributes at some given node.

4.2. Working Within the Processor

Many of the problems of state storage can best be tackled by adapting the XSLT processor itself. The task of retrieving the required execution state information is then made easier by having direct access to the processor's internal data structures. The values of variables, parameters etc. can be accessed and copied at any point during execution, without the need to modify the original stylesheet.

The problem of separating the stored state information from the original output of the stylesheet is also solved in this approach, since we have control over the output streams used by the processor. The original result stream can be left unaffected, whilst the state information produced as the processor executes can be stored internally, or redirected to a separate output stream.

Although existing XSLT processors are clearly not designed with partial re-evaluation in mind, and therefore there is potentially significant refactoring required, we believe that adapting an existing processor is a better solution than developing a new processor from scratch. The main benefit is that of having a fully working XSLT processor from the very beginning which allows us to concentrate fully on the partial re-evaluation functionality.

In selecting an XSLT processor for modification there are many potential candidates such as Saxon [9], Xalan [10], etc. We opted to use Saxon in this work due to its support for XSLT 2.0 and its reputation of being one of the best XSLT processors available.

5. PROCESSOR MODIFICATIONS

The execution process performed by Saxon can be divided into the following stages:

- XSLT script and input XML document parsing
- Stylesheet compilation
- Stylesheet execution.

Each of these stages requires some augmentation to provide the functionality necessary to support partial re-evaluation.

5.1. Input Data Representation

The initial parsing of the XSLT script into an internal tree representation is typical of that performed by many XML-based tools and is left largely unchanged. However, in order to support partial re-evaluation easily, as a result of changes to the input data, a new *switchable* tree representation of the input XML document is used.

Because the result document that is created is entirely dependent upon the variable data instance used, the document author must be able to change this data in order to view alternative versions of the document that might be produced. Simply selecting a complete new input document limits flexibility and also complicates the process of recording the processor's execution state. Therefore, a new document model is utilized in which all variations of nodes and content are amalgamated into a single structure that can be morphed into any single instance by selecting a particular combination of the alternative nodes available. This structure implements the standard DOM interface, and so can be directly processed both by Saxon and also by any other parts of the editing framework that are aware of its underlying nature.

5.2. Stylesheet Compilation

The next stage in the internal processing pipeline is that of compiling the parsed XSLT stylesheet into a series of instructions that will later be executed. This process is augmented by maintaining relationships between the compiled instructions (Java objects) and their corresponding tree nodes. These relations are necessary when selecting and re-executing the specific instructions that relate to parts of the stylesheet that we wish to partially re-evaluate.

A second major modification supports changes to the stylesheet as a result of an editing operation. Any changes made to the parsed XSLT stylesheet tree must be reflected in its compiled form if they are to take effect when the document is re-executed. Therefore, functionality must be added to support the removal of instructions from the compiled executable when the corresponding node has been removed from the stylesheet tree. In most cases, this "un-compilation" is simply a case of removing the relevant instruction object from its parent, but in others a series of changes made to related objects must also be reversed.

5.3. Stylesheet Execution

The final execution stage sees the most changes. Each of the instructions in the compiled form of the stylesheet must record the state of the processor when it is executed. To achieve this, the execution routines built into the various instruction classes have been modified to save the required information to a global 'state tree' held within the processor. To minimise the amount of memory required, the state tree is structured so that each entry stores only new or updated values — any unchanged or previously defined variables etc. are referenced from preceding states.

As well as storing the state during execution of the compiled instructions, support must also be added to allow for this information to be used during partial re-evaluation of the stylesheet. Therefore, the output generated by our revised version of Saxon extends the standard DOM interface by exposing the relations between the output elements and the instructions that generated them. Using these relations, along with a reference to the correct entry in the state tree, we can call upon the processor to re-execute the stylesheet starting at a given instruction. Functionality has been added to support the restoration of the processor state from the supplied values stored within the state tree.

Before performing the re-execution of the selected instruction(s), we change the output stream used by Saxon through which any result nodes are sent. By creating a new stream and diverting the newly produced output through it, we can capture the generated result tree fragment. This can then be used to replace the existing subtree in the original result document.

6. AUTOMATIC RE-EVALUATION

In a typical editing environment, it is essential that the currently displayed view is an accurate representation of the document. Therefore, when an edit is made to the document the view must be redrawn, with the additional possibility of some parts of the result document requiring reprocessing. Consequently, it makes sense for the reprocessing to be performed if, and only if, *the current document view requires it*. This moves the decision of what parts of the result document are in need of re-evaluation to the result document itself and eliminates redundant re-processing of components that are not displayed. The problem remains however, that each node in the result tree has no knowledge of the edit that has been performed.

To solve this problem, references to nodes in the input data tree are stored as part of the processor state every time a node is accessed during processing. A similar process of recording the specific instruction objects used in the stylesheet is also performed. Therefore, when the result document tree is accessed as part of the necessary re-drawing, each node has access to references to all the input data nodes and instruction objects that were used during its creation and can check to see whether these nodes/instructions have been changed or replaced.

7. RESULTS

The analysis now presented uses three contrasting configurations to highlight the relative performance of the proposed techniques. The first of these configurations uses the original version of Saxon without any modifications. The two remaining setups both use the augmented version of Saxon that has been developed with support for the techniques discussed in this paper. The first of these performs a complete re-evaluation of the document, whereas the other performs only a partial re-evaluation in accordance with the techniques discussed earlier in the paper. For simplicity, the three systems will be referred to as A, B and C respectively.

Table 7.1 shows the performance of the test systems when re-processing a document that contains only a number of simple lines of text. An increase in the size/complexity of the document is modelled by adding greater numbers of text components to the page. A localised edit is simulated by reprocessing just one of the lines of text present in the result document.

As expected, the time taken to generate the updated version of the result document by the plain Saxon system (A) increases in direct proportion to the number of components. This outcome is also observed in system B where the entire document in also re-evaluated. The partially evaluating system, C, maintains a near constant re-computational cost irrespective of the overall size of the document.

Components	A (ms)	B (ms)	C (ms)
10	1.137	1.281	0.200

20	1.901	2.125	0.202
40	3.271	3.670	0.202
100	7.692	8.705	0.202
1000	77.326	86.361	0.206

Table 7.1 – Performance results of test systems

Although the values for system B are larger than those for A, the frequency with which this worse case is encountered in our partially re-evaluating processor is relatively low. This contrasts with the workings of system A, where every reprocessing operation results in just such a worst case. Independence between document components, and the system of automatic re-evaluation previously discussed, means that circumstances requiring a full re-processing of the document, such as those modelled in system B, are unlikely to occur in practice. Furthermore, since documents authored from scratch are built one component at a time, the overheads that are the cause of the larger values for system B are absorbed in an incremental fashion, thus maintaining low individual re-processing costs.

8. CONCLUSIONS AND FUTURE WORK

In this paper we have shown how a processing framework can be optimized for repeated editing of a document using a partially re-evaluating processor that records, and restores, its execution state. We are greatly encouraged by the observation, in practice, that the speed increase, of case C compared to case A, truly does result in a much more responsive editing experience. It is envisaged that optimizations, and other improvements, currently being tested with the reworked DOM implementation and its associated data structures, can further reduce the initial processing cost and subsequent overheads.

Given that VDP documents often have a markedly "component based" nature, the possibility of generating the final SVG output in the form of SVG COGs [11] has the potential for limiting, or removing, the dependencies between document components, thereby leading to a greater localization of affected node sets within the input document. Furthermore, the possibility of developing fully componentized *source* documents offers the potential to increase their inherent locality, hence making the performance gains shown in this paper more readily achieved.

A final area of interest is that of extending the work beyond the realm of editing VDP documents into the phase where they are actually printed. Rather than generating a series of documents by fully evaluating with new instance data, it might be possible to treat each customised data instance as a series of edits to the previous instance. For sets of documents with a sufficient degree of commonality, this might well provide a method of efficiently generating instance documents without the need to perform ahead-of-time optimizations such as those described in [12, 13].

9. ACKNOWLEDGMENTS

Thanks are due to Hewlett Packard (UK) and EPSRC for supporting James Ollis's PhD studentship.

10. REFERENCES

[1] John Lumley, Owen Rees and Roger Gimson, "A Framework for Structure, Layout & Function in Documents" in *Proceedings of the ACM Symposium on Document Engineering (DocEng'05)*, pp. 86–89, ACM Press, 2–4 Novemeber 2005.

[2] John Lumley, Roger Gimson and Owen Rees, "Configurable Editing of XML-based Variable-data Documents" in *Proceedings of the ACM Symposium on Document Engineering (DocEng'08)*, pp. 76–85, ACM Press, 16–19 September 2008.

[3] CatBase Software Ltd., CatBase. `http://www.catbase.com`

[4] XMPie, uDirect, `http://www.xmpie.com/`

[5] Markus Noga, Steffen Schott and Welf Löwe, "Lazy XML Processing" in *Proceedings of the ACM Symposium on Document Engineering (DocEng'02)*, pp. 9–18, ACM Press, 20–22 November 2002.

[6] Steffen Schott and Markus Noga, "Lazy XSL Transformation" in *Proceedings of the ACM Symposium on Document Engineering (DocEng'03)*, pp. 88-94, ACM Press, 8–9 November 2003.

[7] Lionel Villard and Nabil Layaïda, "An Incremental XSLT Transformation Processor for XML Document Manipulation" in *Proceedings of the 11th International Conference on the World Wide Web*, pp. 474–485, ACM Press, 7–11 May 2002.

[8] James Ollis, David Brailsford and Steven Bagley, "Tracking Sub-Page Components in Document Workflows" in *Proceedings of the ACM Symposium on Document Engineering (DocEng'08)*, pp. 86–89, ACM Press, 16–19 September 2008.

[9] Kay, Michael. Saxon XSLT Processor. `http://saxon.sourceforge.net`

[10] Apache Software Foundation, The Apache Xalan Project, `http://xalan.apache.org/`

[11] Alexander J. Macdonald, David F. Brailsford and Steven R. Bagley, "Encapsulating and manipulating component object graphics (COGs) using SVG" in *Proceedings of the ACM Symposium on Document Engineering (DocEng'05)*, pp. 61–63, ACM Press, 02–04 November 2005, Bristol, UK.

[12] Alex Macdonald, David Brailsford and John Lumley, "Evaluating Invariances in Document Layout Functions" in *Proceedings of the ACM Symposium on Document Engineering (DocEng'06)*, pp. 25–27, ACM Press, 10–13 October.

[13] Alex Macdonald, David Brailsford, Steven Bagley and John Lumley, "Speculative Document Evaluation" in *Proceedings of the ACM Symposium on Document Engineering (DocEng'07)*, pp. 56–58, ACM Press, 28–31 August 2007.