Pruning DOM Trees for Structured Document Processing*

Yasushi Hayashi† Zhenjiang Hu† Masato Takeichi†
Nobuaki Wake†† Masafumi Hara†† Norio Oshima††

†Graduate School of Information Science and Technology, University of Tokyo
††Justsystem Corporation
{hayashi,hu,takeichi}@mist.i.u-tokyo.ac.jp
{nobuaki_wake,masafumi_hara,norio_ooshima}@justsystem.co.jp

PSD (Programmable Structured Document) is a framework in which structured documents are edited efficiently and safely by evaluating embedded expressions in themselves. The PSD processing system we are currently developing requires an external evaluator to get the DOM data of documents held in the editor. In this work, a method to prune DOM trees is proposed to improve the performance of document manipulations by avoiding unnecessary data communication between the editor and the external evaluator. Based on information about references given by the user, it generates a pruned DOM tree, eliminating unnecessary parts for the evaluation from the original DOM tree. The mechanism of tree pruning is explained and its efficiency is evaluated using examples.

1 Introduction

XML has been widely adopted as a standard for describing structured documents. In the conventional way an XML document is written, the element values are independent from each other. Introducing computations in the XML document for manipulating themselves naturally allows some parts of a document to be dependent on other parts. It will enhance its usability, especially these dependencies are essential factors for the document. In [6], we proposed the concept of *Programmable Structured Documents (PSD)*, that is, a document that (1) contains computation: a document is itself a program; (2) can be manipulated by itself: a document becomes a meta program; (3) is reliable and reusable. For example, in [6], we introduced a PSD application called *TreeCalc*, which aims at a tree version of functional spreadsheet[2], where users can easily create and delete an XML tree, some of its subtrees depending on other subtrees.

In the current framework of our PSD system, we assume that an editor uses DOM [7] to keep XML data, but the language for embedded code need not be restricted by this assumption. It could be a language that cannot directly deal with XML as DOM objects. This demand to be language-independent leads to the structure of a PSD system having an editor to keep XML documents and an external evaluator for evaluating embedded code. For such kind of systems, having an effective interface from which the external evaluator can access and manipulate the documents efficiently and flexibly is critical. For example, in TreeCalc, we embed Haskell in XML, and made use of an XML editor/viewer, developed in Java, which employs a DOM for editing and presentation. The whole XML document is sent from the editor to the external evaluator where a Haskell code is evaluated. The evaluated result is returned to the editor to be plugged back in the original document. However, this process performs unnecessary data communication when a large part of data need not be accessed by the evaluator. Especially, when the system deals with large XML data, this data communication cost could be a large overhead. To remedy this drawback, in this work, we introduce *tree pruning* in which a system constructs a document tree by eliminating unnecessary parts for the evaluation from the whole document tree before any document manipulation starts.

*The project is supported by the *Comprehensive Development of e-Society Foundation Software* of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

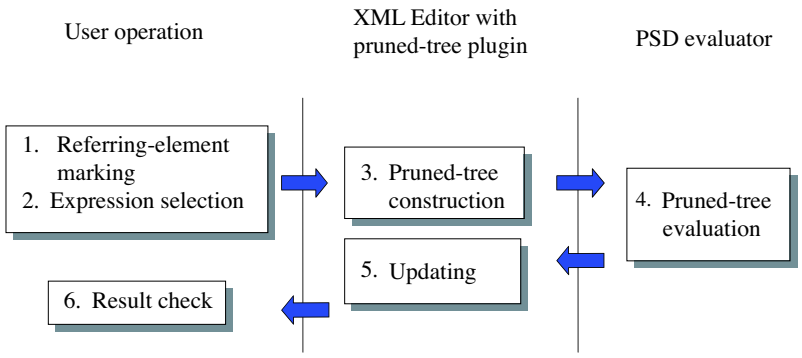

Figure 1: Overview of tree pruning

2 Overview of tree pruning

Figure 1 illustrates the process of XML data manipulation in the proposed system. Firstly, users give information of references by marking referring elements. In the current user interface of a viewer, a user selects an embedded expression and executes a command to evaluate it. When the command is executed, the editor constructs a *pruned tree*, which will be explained in detail in the following sections, and then send the pruned tree to an external evaluator. The evaluator receives the pruned tree including the code, and evaluate it. The evaluated subtree is sent back to the editor and used to update the original DOM tree. The result is displayed to the user by the viewer.

3 Pruning DOM trees

The aim of tree pruning is to make the parts of the document to be sent to the external evaluator as small as possible. Therefore, we try to keep only subtrees whose elements may be referred during the evaluation. From now on, we call them “useful subtrees”. To identify the useful subtrees, the system relies on user’s annotations. If some element in the source document refers to other elements, it is marked by users with an attribute “refer-to” in the form of

```
<src:element
   prt:refer-to="referId(refer-list)">
   Contents
</src:element>
```

where `referId` is an unique ID for the reference and `refer-list` is a sequence of XPath expressions that specify elements to be referred. After selecting a code element and executing a command to evaluate, the closest ancestor element which has a refer-to attribute is marked as the “start” element. Then the elements referred by the start element are marked with information about the reference ID names. If they, in turn, refer to other elements, those referred elements are also marked recursively. Circular dependencies can be detected in the recursive process. After all referred elements are marked, the system copies (in the order they appear in the document) all elements that are either marked or have a marked element as its ancestor. The useful subtrees consists of those copies. A pruned tree has the root element `prt:pruned-tree` and includes the useful subtrees taking the form of:

```
<prt:pruned-tree>
   Contents: (src:element)*
</prt:pruned-tree>
```

That is, the root element `prt:pruned-tree` has a sequence of useful subtrees as its direct children. When making copies of useful subtrees, the system records information about reference ID names and the position in the source document in the root element of the subtrees in the form of:

```
<src:element
  prt:referred-from=referred-list
  prt:original-path=element-expression>
  Contents
</src:element>
```

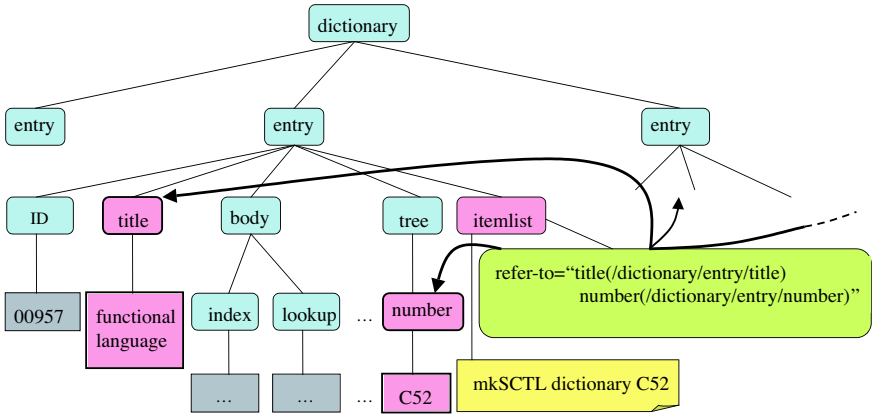

Figure 2: DOM tree of CS dictionary

where `referred-list` is a sequence of reference ID names. Each of ID names takes the form of `ID[index]` where `index` is an integer to express a reference order when multiple elements are referred by one reference. An `element-expression` is the position of the referred element in the source document, which is expressed as a path expression.

When editing the pruned tree, the `referred-from` attribute is particularly important because it allows us to identify the referred elements in the pruned tree. Note that path expressions in the source document cannot be used for this purpose anymore since the tree structure of the source document is different from that of the pruned tree. The `original-path` attribute is not used in the external evaluator but used in the editor for locating the returned tree to the original tree. The referred-from attributes and the original path attribute are eliminated when the pruned tree is pluged back into the source document.

4 A concrete example

To illustrate this process more concretely, we use the XML edition of *Iwanami Encyclopedic Dictionary of Computer Science* (we call it CS dictionary for brevity). The dictionary has a root element `dictionary` having a sequence of `entry` elements as its children, each has children `ID`, `title`, `body`, `tree` that gives information about the ID number, title, and description, and category tree respectively for the entry. A `body` element has a number of children, including `index` and `lookup`, which indicate that some words in the description have references in other parts of the dictionary. Figure 2 illustrates the DOM tree data structure for one entry.

In PSD, users can write expressions in the XML document. For example, `mkSCTL dictionary "C52"` in Figure 2 is a piece of code that automatically generates the list of the titles which belong to the same category number C52 (for “programming methodology”) in this case. The expression has been placed after a `tree` element together with a new `itemlist` element as its parent. When it is evaluated, the resulting list replaces the code. The user gives

```
refer-to="title(/dictionary/entry/title)}
      number(dictionary/entry/tree/number)"
```

as an attribute of `itemlist` element to tell the system that it refers to all `title` elements and all `number` elements, where “title” and “number” before the parenthesis are ID names of the references. After selecting a code and execute a command, the `itemlist` element is marked as the “start” element. Then the referred elements `title` and `number` are marked with information about the reference ID. As a result, the useful subtrees consists of copies of the elements that have a `title` or `number` element as their ancestor in the original document. The pruned tree is the set of useful subtrees as shown in Figure 3. Note that although the relation between the `title` and the `number` under one entry seems to be broken, it can be recovered since

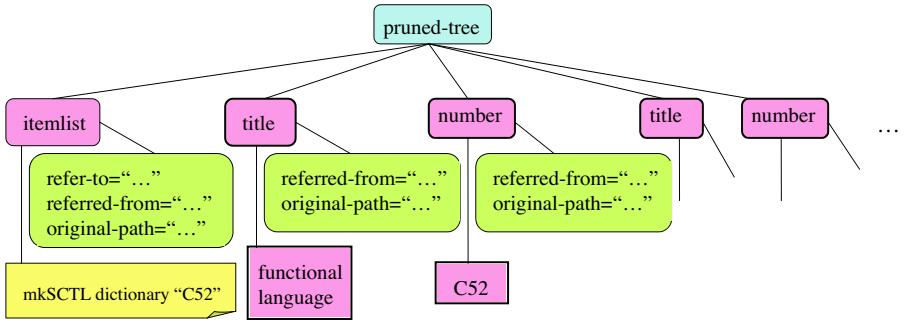

Figure 3: Pruned tree

their relative order in the sequence is the same as the order they appear. When making copies of the useful subtrees, the first `title` elements are assigned attributes `referred-from="title[0]"` and `original-path="*[1]/*[2]/*[4]/*[1]"`, and the first `number` elements are assigned attributes `referred-from="number[0]"` and `original-path="*[1]/*[2]/*[7]/*[1]"`, similarly with the other elements. By evaluating the

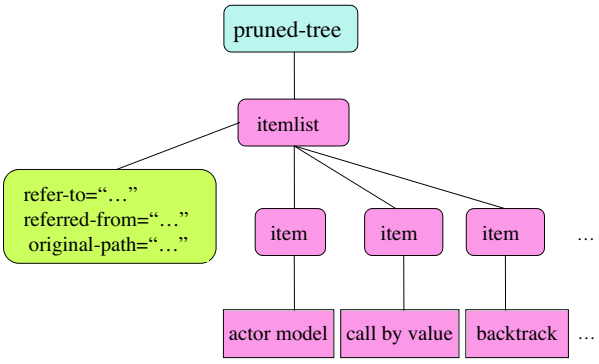

Figure 4: Evaluated pruned tree

expression on the pruned tree, the list of titles whose category number is C52 is generated as shown in Figure 4. When this resulting pruned tree is returned, the source document is updated as shown in Figure 5.

5 Experiments

In our experiments, we use Haskell for writing embedded codes and its interpreter as the external evaluator. A pruned DOM tree in the editor is transformed into XML and sent to an external process where it is further translated into Haskell tree

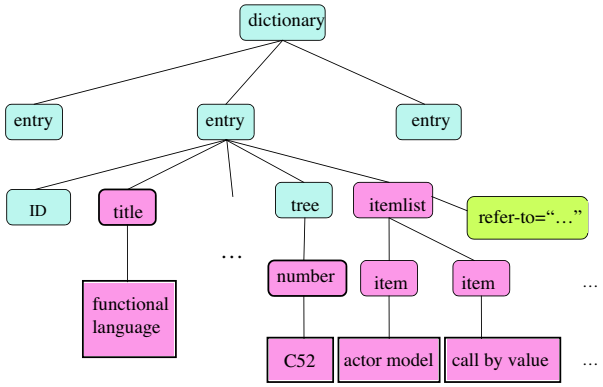

Figure 5: Updated source document

by a macro processor. After the evaluation, the resulting XML pruned tree is sent back to the editor to be plugged into the source document.

To test its performance, we measure the time to build pruned tree and to run the external process in which the pruned tree is sent, edited, and returned. Then, we compare it with the time to run the external process when the whole tree is sent, edited, and returned, so that the result shows how much efficiency has been gained by tree pruning. Figure 6 shows the test result for the evaluation of `mkSCTL dictionary "C52"` in the previous section varying the number of entries up to 280. The plotted line labeled "original" shows the runtime when the whole document tree was sent, while the plotted line labeled "pruned" shows the runtime when a pruned tree was sent. In this example, since the tree pruning avoids sending the text of the body element, which is the most dominating part in the whole document, some performance gain is expected. In

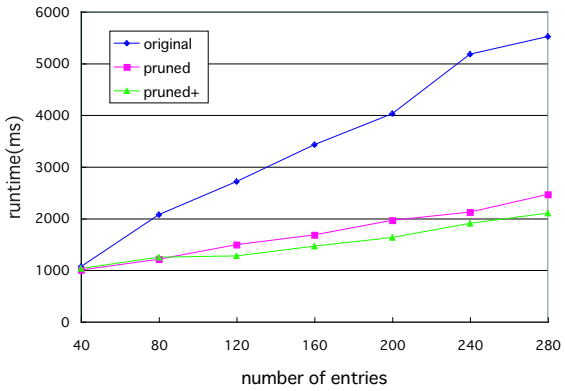

Figure 6: Comparison of runtimes

fact, the test result shows that the costs of using the pruned tree methods are less than half of that of using original method. We got similar results for other examples in which the most of the texts in the body element are not be sent. The plotted line labeled "pruned+" shows the rusult of an additional experiment that will be explained in the next section.

6 Discussions and related work

In addition to the experiment in section 5, we also made an additional experiment using a slightly different version of the pruned tree in which only the text nodes that are not included in the useful subtrees are eliminated, while preserving the whole tree structure (in this sense, *defoliation* may be a better word for it). Its motivation is that this additional information is often useful when writing a code with pattern matching in some languages like Haskell and CDuce[1]. In this case, since the data of all tag names is sent to the external evaluator, the communication cost increases. The significance of the increase depends on the proportion of the amount of data about tag names to that of the whole data. For the example in the previous section, its proportion is only about 20%. The runtimes when using this type of pruned tree for the same example is shown by the plotted line labeled "pruned+" in Figure 6. They are slightly smaller than that when using the other type, but the difference is small. The reason for its better performance is due to its smaller costs for constructing the pruned tree.

Optimization Techniques that process only a necessary part without reading the whole DOM data are often used in DOM manipulations [4][5]. A characteristics of our technique is that it is used for the purpose of minimizing communication cost in a setting in which the manipulations are done by an external evaluator. For a similar purpose, our group is also investigating a different approach called "By-need DOM" [3] in which the DOM server sends the necessary data for an external evaluator only when needed during the evaluation. A comparison of performance characteristics between the two different techniques will be beneficial.

7 Conclusion

We proposed a new method to avoid unnecessary data communication between an XML editor/viewer and an external evaluator to improve performance of XML document manipulations in a PSD processing system. The performance test shows that a considerable performance gain can be observed at least for a certain kind of application.

References

[1] V. Benzaken, G. Castagn, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2003.

[2] W. A. C. A. J. de Hoon, L. M. W. J. Rutten, and M. C. J. D. van Eekelen. Implementing a Functional Spreadsheet in Clean. *Journal of Functional Programming*, 5(3):383–414, 1995.

[3] S. Nishioka, K. Nakano, Z. Hu, and M. Takeichi. By-Need Evaluation of Programmable Structured Documents (in Japanese). In *The 21st JSSST Annual Conference*, 2004.

[4] M. L. Noga, S. Schott, and W. Löwe. Lazy XML Processing. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, pages 88–94. ACM Press, 2002.

[5] ObjectWeb Consortium. Enhydra XMLC. `http://xmlc.objectWeb.org/`.

[6] M. Takeichi, Z. Hu, K. Kakehi, Y. Hayashi, S.-C. Mu, and K. Nakano. TreeCalc: Towards Programmable Structured Documents. In *The 20th JSSST Annual Conference*, 2003.

[7] W3C. Document Object Model (DOM). `http://www.w3.org/DOM`.