XML Pipeline Performance

Nigel Whitaker
DeltaXML Ltd
<nigel.whitaker@deltaxml.com>

Tristan Mitchell
DeltaXML Ltd
<tristan.mitchell@deltaxml.com>

Abstract

This paper considers XML pipeline performance of a case study constructed from real code and data. It extends our original small study into 'Filter Pipeline Performance' on the saxon-help email list. The system used in this case study is a comparison tool for OASIS OpenDocument Text or '.odt' files.

We will not describe the code in detail, but rather concentrate on the mechanisms used to interlink the various pipeline components used. These components include XML parsers, XSLT filter stages, XML comparison, Java filters based on the SAX XMLFilter interface and serialization. In the previous study we compared two pipelining techniques; this paper will extend this to consider three mechanisms which can be used to construct pipelines; these are:

- *The Java API for XML Processing (JAXP) included in Java Standard Edition 1.4 and subsequent releases*
- *The s9api package provided by the Saxon XSLT processor 9.0 and subsequent releases*
- *The Calabash implementation of the XProc XML pipeline language*

Our primary focus is performance and we will look at run times and memory sizes using these technologies. The overall runtime of the complete system will be described, but we will then concentrate on the performance of running a chain of filters (XSLT and Java based) as this is likely to be of more general interest. We will also discuss some optimization techniques we have implemented during the development process and further ideas for future performance improvement work.

1. Introduction

In this section we will introduce the system being studied, briefly describe the data and also some aspects of the experimental approach.

1.1. The case-study system

The system used for this case study is an ODT comparator. ODT or 'Open Document Text' is a document format supported by a number of word processors and office systems including OpenOffice.org and derivatives such as IBM Lotus Symphony, KWord (from KDE/KOffice) and Google Docs. The format is a standard developed by OASIS [4]. The ODT comparator is one of a number of ODT products from DeltaXML and this one is available for free-of-charge use, either online or through an OpenOffice.org plugin.

Our implementation of an ODT comparator consists of 5 processing pipelines roughly associated with the XML components of an ODT file (content.xml, styles.xml, meta.xml). The pipelines use XSLT 2.0 [9] and Java filters to perform various data manipulations (for example list and table rearrangements, style rationalization and reconstruction), comparison and post-processing and serialization back into the components of the ODT 'zip' file. Some filters are relatively simple (33 lines of code), the longest is 585 lines of XSLT 2.0. Java extension functions are also called, for example to compare binaries such as images and perform measurement unit conversions. The longest pipeline, for content.xml, consists of: input filter chains of 7 filters, a comparator, 22 output filter stages together with parsing and serialization.

1.2. Performance Objectives

We are proponents of using pipelined architectures for processing XML [5]. We prefer the divide and conquer approach and reusability benefits of simple filters composed into larger systems. Some of the filters we used are general purpose (work on any well-formed XML) and others are ODF specific. These filters are implemented in either XSLT 2.0 or Java code. We often test individual templates/functions in the filters. We also test individial filters and sets of filters as well as whole pipelines as black boxes.

However, we also recognize the need to minimise the overheads of multiple XSLT transformations or Java filtering process. Past experience has made us avoid using temporary intermediate files to link pipeline stages together (disk IO is too slow). Similarly using in-memory buffers (Strings or ByteArrays in Java) to hold intermediate results is avoided as there is also an overhead to reparsing lexical XML. Our inter-stage communication should be efficient ideally using parsed, pre-processed or event-based XML. But the primary reason for starting the performance work described in this case study was to minimize the memory footprint.

1.3. Experimental method

In order to appreciate the numbers presented later in this paper it may help to understand some details of the systems used and experimental techniques.

- The system used to perform the measurements was an Apple MacBook Pro with an Intel Core 2 Duo processor running at 2.53Ghz. The system has 4 GBytes of RAM and was running MacOS 10.6.2
- More recent JVMs offer better performance and fewer bugs. We used the most recent JVM/JDK combination available on the above system. This was Java version 1.6.0_17, and more specifically the: `Java HotSpot(TM) 64-Bit Server VM (build 14.3-b01-101, mixed mode)`
- When measuring runtimes we used a large heap size, typically a **-Xmx2g** heap setting, to avoid excessive garbage collection which often occurs when using close to minimal heap sizes.

When measuring memory sizes we used -Xmx virtual machine arguments to determine to nearest 1MB the smallest heap size which does not cause an OutOfMemoryError.

Where possible timings were done post XSLT compilation and also avoided measuring the JVM start-up times. Memory measurements were made using the facilities provided by `java.lang.manangement` APIs included in Java 1.5 and subsequent releases.

The data files used when reporting results were large (over 700 pages) annual financial reports. Two revisions with several hundred differences were compared.

2. The JAXP Pipeline

The Java API for XML Processing (JAXP) [3] has been available for a number of years and allows Java programmers to construct XML processing pipelines. Its provides classes such as the `SAXTransformerFactory` to create `TransformerHandler` and `XMLFilter` instances. A JAXP pipeline is constructed and then invoked through a single trigger point (a single `transform()`, `parse()` or when used in conjunction with our components a `compare()` method).

The pipeline construction involves linking the stages togther using the `setParent()` method for an `XMLFilter` or `setResult()` for a `TransformerHandler`, often via an intermediary `SAXResult` object. All of the stages communicate via the SAX [6] `ContentHandler` interface. Logically it can be considered that events flow between the pipeline stages. In an ideal world all of the filters in a pipeline would stream and there would be very few large in memory data structures. However, in reality this is often the exception and many stages will use in-memory data structures as part of their processing. For example, in order to support navigation using all of the XPath axes in an XSLT processor, an in-memory tree, or array-based-tree, data structure is often convenient. Similarly, the Longest Common Subsequence (LCS) algorithms used in comparison work well with in-memory data strucutures. Only when chaining `XMLFilter` or other callback or 'event' based code together does information 'stream' down a pipeline.

Using JAXP pipelines the following overall result was obtained:

Minimum heap space	933 MB
Runtime (2GB heap)	5 min 14 sec

3. The s9api Pipeline

Introduced in Saxon [7] version 9.0, this API allows `XdmNode` trees to be used as the inputs and outputs of a transformation. This allows chaining via the `setDestination` method and a single trigger method as shown in Example 1.

Example 1. Chained s9api pipeline example

```
XsltTransformer stage1=
      comp.compile(new StreamSource(new File("stage1.xsl"))).load();
XsltTransformer stage2=
      comp.compile(new StreamSource(new File("stage2.xsl"))).load();
XsltTransformer stage3=
      comp.compile(new StreamSource(new File("stage3.xsl"))).load();
...
stage1.setDestination(stage2);
stage2.setDestination(stage3);
stage3.setDestination(...);
stage1.setInitialContextNode(...);
stage1.transform();
```

However as our earlier email discussion [2] indicated this approach consumed significant amounts of memory for long pipelines and appeared to have linear memory consumption with pipeline length. A 'multitriggered' model allows explicit control over intermediate trees, including explicit nullification of references to help the garbage collector. The code for a simplified three stage multi-triggered pipeline is shown in Example 2.

Example 2. Three stage s9api pipeline code example

```
XdmDestination stage1result= new XdmDestination();
stage1.setDestination(stage1result);
stage1.setInitialContextNode(in);
stage1.transform();

in= null;
XdmDestination stage2result= new XdmDestination();
stage2.setDestination(stage2result);
stage2.setInitialContextNode(stage1result.getXdmNode());
stage2.transform();
```

```
stage1result= null; stage1= null;
XdmDestination stage3result= new XdmDestination();
stage3.setDestination(stage3result);
stage3.setInitialContextNode(stage2result.getXdmNode());
stage3.transform();
```

The actual code used is more complex and uses a `java.util.List` to store the filters and then iterates over the list. It also handles Java filters (depicted in Figure 1) and holds the previous filter's `XdmNode` tree which is used as input while the result of the current filter is being generated as another `XdmNode` tree. Care was needed when writing this code to ensure objects could be garbage collected. For example, `XsltTransformer` objects may retain references to their input, or initial context trees after they are executed.

Using multi-triggered s9api pipelines the following initial comparison results were obtained:

Minimum heap space	457MB
Runtime (2GB heap)	4min 5sec

These results demonstrated a significant memory improvement and reasonable speed increase. However these initial results were known to be non-optimal and subsequent optimizations are discussed in Section 5.

The multi-triggered s9api implementation achieves the objective of reducing the memory requirements of the overall pipeline. We have observed, using both code profilers and the monitoring facilities provided by the `java.lang.management` package that for the execution of a pipeline step the required memory approximately corresponds to:

- The size of the input and output `XdmNode` trees of the step
- The size of the internal data structures used by the step
- The size of pipeline wide data strucutures such as the Saxon Processor and its associated `NamePool`.

As the pipeline steps are executed, the memory requirement fluctuates according to the current step. The overall requirement for the pipeline corresponds to the step with the largest input and output trees and internal data structures. In the case of the ODT Comparator used in this case study, this step is the XML comparator used in the centre of the main content.xml pipeline, which unlike many of the other pipeline steps has two large input trees and the largest output tree.

4. The Calabash Pipeline

Calabash [1] is an XProc [8] implementation being developed by Norman Walsh. Its implementation internally uses Saxon `XdmNode` trees. XProc allows us to implement pipelines without the custom Java code needed for the previous Pipelines. We chose to use Calabash as it supported our preferences for Java and XSLT 2.0. Other XProc implementations are available and could also have been used. XProc does not allow us to implement our pipelines directly as it does not support our use of Java filter components and Calabash did not provide an extension step to run these filters. However, using the s9api pipeline code for running a Java filter as a basis it was relatively easy to implement an extension step in Calabash for these filters. The step declaration is provided in Example 3.

Example 3. Calabash extension step for Java filters

```
<p:declare-step type="dx:java-filter"
  xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:dx="http://www.deltaxml.com/ns/extensions/xproc"
  xmlns:cx="http://xmlcalabash.com/ns/extensions"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <p:input port="source" primary="true" kind="document"
        sequence="false"/>
    <p:input port="parameters" kind="parameter"/>
    <p:output port="result" primary="true" sequence="false"/>
    <p:option name="classname" required="true" cx:type="xs:string"/>
</p:declare-step>
```

This step requires the full classname, which must extend a SAX `XMLFilterImpl`, to be specified as a string and also allows parameters to be specified. Java reflection is used to locate a method using the parameter name prefixed with 'set', as the method name, and which takes a single Java `String` argument. A fragment of the ODT pipeline illustrating such a step is provided in Example 4.

Example 4. Using the Java filter extension step

```
<dx:java-filter>
  <p:with-option name="classname"
    select="'com.deltaxml.pipe.filters.dx2.wbw.OrphanedWordOutfilter'"/>
  <p:with-param name="orphanedThresholdPercentage"
          select="$orphanPercentage"/>
  <p:with-param name="orphanedLengthLimit" select="$orphanLength"/>
</dx:java-filter>
```

This extension step certainly proved useful when making our transition from JAXP. We hope to make the extension step available for use in Calabash and possibly other XProc implementations.

The XProc implementation of the ODT comparison pipeline made use of the standard `p:xslt` step, a modified version of `cx:delta-xml` and the `dx:java-filter` step described above. Six files comprising around 1,000 lines of XProc statements were needed. The performance of this implementation is as follows:

Minimum heap space	1631 MB
Runtime (2GB heap)	4 min 42 sec

The runtime is impressive and an improvement over the JAXP implementation that was our starting point. The memory requirements are higher than we hoped, but it must be remembered that we are using a 0.x implementation and may improve in future releases.

While this paper concentrates on performance issues it is worth pointing out that the conversion of Java pipeline code into XProc was a fairly easy process, once we appreciated the intricacies of XProc pipeline construction. The five pipelines or `p:step-declaration`s used to implement the ODT comparator comprised around 1,000 lines of XProc declarations.

5. Optimizations

In this section we will discuss some of the optimizations used to improve performance (both runtime and memory) of our s9api implementation. These optimizations could also be applied to Calabash.

5.1. Java filter combining

With our initial, simple s9api implementation we chose to make every stage communicate via XdmNode trees. In some cases we knew this was less efficient than JAXP where adjacent Java based XMLFilters do 'stream'. Our initial implementation is depicted in Figure 1.

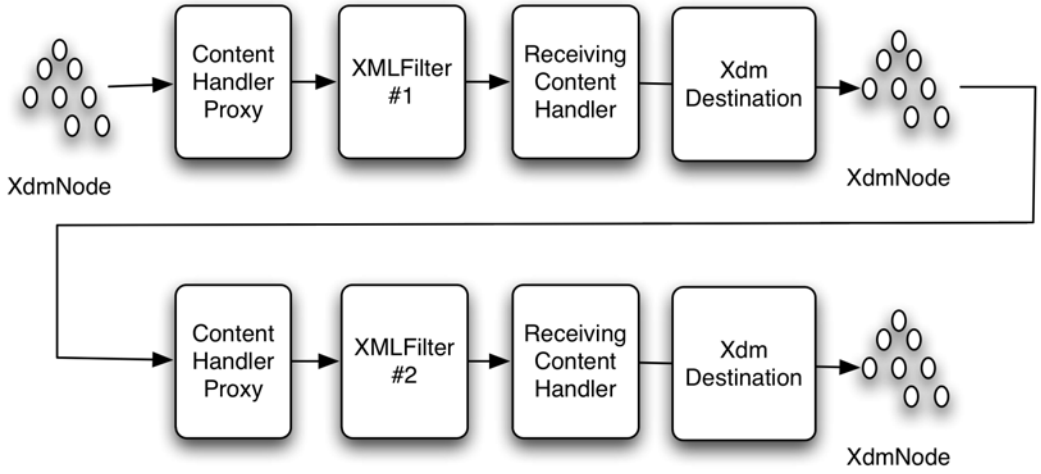

Figure 1. Adjacent java filters linked via Xdm trees

It is possible to detect adjacent Java or more generally streamable filters and connect them together more efficiently. This is depicted in Figure 2

Figure 2. Adjacent java filters combined to stream

The screen output shown in Example 5 captures the debugging/progress information from a sequence of five Java filters used in the main content.xml comparison result pipeline. At this stage of the pipeline these filters are processing fairly large amounts of data, over 100MBytes of none-whitespaced XML when serialized, which accounts for the relatively long runtimes.

Example 5. Progress report for 5 adjacent Java filters

```
finished r[0] (Id: com...filters.dx2.wbw.OrphanedWordOutfilter)
              (elapsed: 2897 ms, cpu: 2507 ms)
finished r[1] (Id: com...filters.dx2.wbw.OrphanedWordOutfilter)
              (elapsed: 2825 ms, cpu: 2468 ms)
finished r[2] (Id: com...filters.dx2.wbw.WordSpaceFixup)
              (elapsed: 3063 ms, cpu: 2634 ms)
finished r[3] (Id: com...pdf.filters.ChangeMetricsFilter)
              (elapsed: 2724 ms, cpu: 2400 ms)
finished r[4] (Id: com...filters.dx2.wbw.WordOutfilter)
              (elapsed: 2585 ms, cpu: 2257 ms)
```

When these are combined and streamed together the result becomes:

```
finished r[0]-r[4] (elapsed: 5347 ms, cpu: 4724 ms)
```

The output indicates that by combining and streaming these filters together we can halve or better their runtime. However the output also demonstrates a disadvantage of combining - it is harder to observe progress and the relative performance of individual filters. Furthermore, another disadvantage not demonstrated here is related to exception and error handling. When we process the filters individually we can report precisely which one throws for example a SAXException. When combined together it becomes much harder to indentify the source of an exception from a Java stack trace.

5.2. Java filter conversion

Our initial Java filters were based on the `org.xml.sax.XMLFilter` interface, which is reasonably well supported in JAXP. However, s9api and Saxon more generally, uses a different type of interface/event internally, defined by the `net.sf.saxon.event.Receiver` interface. This interface describes elements for example using integers (to locate data in the Saxon NamePool) and has seperate events/callbacks for attributes, instead of them being bundled with `startElement`. We initially used some conversion classes (as shown in Figure 1) to be able to run an XMLFilter using XdmNode trees as input/output. However there are some inefficiences, related to converting String data to/from integers referencing the name pool, in this process.

At the time of writing we have converted one of our Java filters so that instead of extending the `XMLFilterImpl` class it extends the Saxon `ProxyReceiver` class. The performance is greatly improved as shown in Example 6.

Example 6. Progress report for ProxyReceiver based Java filter

```
finished r[2] (Id: com...wbw.WordSpaceFixup)
              (elapsed: 992 ms, cpu: 809 ms)
```

The performance gained from removing the 'impedence mismatch' of event conversion looks promising and we are now working towards a goal of replacing filters and/or providing additional `Receiver` support where we use `ContentHandler` implementations in our code.

Finally, we must emphesise a note of caution included in the Saxon documentation. The `Receiver` interface is not a public interface of Saxon and could change in future releases. When using these classes, and our code more generally, we will be careful to ensure that it is used in conjunction with a known version of Saxon.

5.3. Additional optimizations

The two techniques discussed above could be used together to support the running of chains of `ProxyReceiver` based filters. Running chains of filters could also be further optimized so that rather than starting from their own XdmNode input tree they are combined with a previous step. So for example an `XsltTransformer` could use `setDestination()` to a `ProxyReceiver` also implementing `Destination`, to subsequent filters via `setUnderlyingReceiver()` and finally to an `XdmDestination`.

We have implemented the optimizations described above using Java code to improve the performance of our s9api based pipelines. However the code structures could also equally be used by Calabash and possibly other XProc processors. Our experience of writing extension steps suggests that a step supporting a list of Java filters (either XMLFilter or Receiver based) as an option or parameter could be easily implemented.

6. Conclusions

We believe this case study confirms that Saxon's s9api package and the Calabash XProc implementation are both viable alternatives to using the JAXP package for building XML processing pipelines. We found the best performance using custom Java code and s9api. As well as the performance benefits the s9api interface is easier to understand and generally more flexible than JAXP. While we are relatively new XProc and Calabash users we also found it provided good results and was reasonably easy to use given its younger heritage. XProc could be a more readily accessible pipelining technology for non-expert users, particiularly those who are not Java programmers.

We have looked at optimization techniques, particularly those related to using event-based and streaming filters coded in Java. These techniques are perhaps useful to the smaller subset of users who are willing to invest development time to gain application performance. We hope that the performance results we have presented are useful to other users considering optimization techniques. This is work in progress (we need to complete more conversions to `ProxyReceiver`), however, the initial optimizations look promising. Here is a summary of the results for the overall ODT Comparator performance:

Pipeline	**runtime (min:sec)**	**minimum memory (MB)**
JAXP	5:14	933
s9api	4:05	457
Calabash	4:42	1631
s9api + partial optimizations	3:12	360

Streaming XML processing provides good performance with a good memory footprint. However, it also complicates measurement of performance, as it becomes difficult to separate the cost of the computation from the communication. We are hoping that writing filters in Java for performance reasons will become a thing of the past, as improvements in XSLT performance or alternative streaming technologies become available.

6.1. Future possibilities

This case study has concentrated on pipelining techniques that we could easily apply to our existing code in the form of discrete pipeline components. There are other approaches that could be taken and may review in the future:

6.1.1. Linking using the saxon:next-in-chain attribute

The `saxon:next-in-chain` attribute can be used to create pipelines. One slight disadvantage is that it hardwires the pipeline structure into the filters and has slight reusability implications (a filter may be reused in different pipelines and may be followed by different filters).

6.1.2. Merging filters

Using temporary trees and modes it may be possible to merge filters together in a single transformation. This may save time, but we no longer have explicit control of the garbage collection of the intermediate trees. Another issue is that when existing filters already use modes, the design of an automatic merging process becomes slightly harder.

6.1.3. Change the pipeline/processing model

For very simple filters we are doing a lot of serialization and tree-building. Most of the time is spent running the 'identity template' in many filters. Perhaps we can instrument and quantify how much time is spent running this filter as opposed to other filters. If it is substantial, we could consider using a persistent in memory data structure (perhaps a form of DOM with XPath support) and then have Java methods for modifying and updating it. We prefer the XSLT processing model and DOM trees can consume a lot of memory; perhaps XQuery Update could be used as an alternative approach to replace one or more filter stages.

Bibliography

[1] Normam Walsh. XML Calabash. http://xmlcalabash.com/

[2] Nigel Whitaker. Filter Pipeline Performance, saxon-help email list, 21 May 2009 http://markmail.org/message/l3k5ajhcvh6gosgq

[3] Jeff Suttor, Norman Walsh (eds). JSR 206: Java API for XML Processing (JAXP) 1.3 Version 1.3, 30 September 2004. http://www.jcp.org/en/jsr/detail?id=206

[4] OASIS: Open Document Format for Office Applications (OpenDocument) Specification v1.1. OASIS Standard, 1 February 2007. http://docs.oasis-open.org/office/v1.1/OS/OpenDocument-v1.1.pdf

[5] Nigel Whitaker, Thomas Nichols. Powering Pipelines with JAXP XML 2004 (IDEAlliance) November 15-19, 2004, Washington, D.C., U.S.A. http://www.deltaxml.com/dxml/336/version/default/part/AttachmentData/data/deltaxml-paper-xml-2004.pdf

[6] David Megginson. Simple API for XML http://www.saxproject.org/

[7] Saxonica Ltd. The Saxon XSLT and XQuery Processor http://www.saxonica.com/

[8] Normam Walsh, Alex Milowski, Henry S. Thompson (eds). XProc: An XML Pipeline Language. W3C Working Draft, 5 January 2010 http://www.w3.org/TR/2010/WD-xproc-20100105/

[9] Michael Key (ed). XSL Transformations (XSLT) Version 2.0 W3C Recommendation, 23 January 2007http://www.w3.org/TR/xslt20/