

# Some results on XML processing in the CIC metadata harvesting project

Howard Ding

May 8, 2006

## 1 Introduction

The CIC metadata harvesting project at the University of Illinois uses the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) to collect descriptive metadata records from CIC member universities. These records are then reprocessed, normalized, and augmented before being re-offered to the UIUC and broader communities via the CIC metadata portal.<sup>1</sup> As of 2005, approximately five hundred thousand records were available through this portal, which may be searched in a variety of ways through three different interfaces. [2, 290-1] Here we shall be concerned with the actual reprocessing of the metadata and more specifically with the underlying computational system necessary to facilitate this reprocessing. A more complete description of the full system, the types of reprocessing done and the rationale for doing so, and some statistical analysis of the success of the reprocessing can be found in [2].

Naturally the original metadata records, harvested as XML documents, are optimized for use in their original contexts. In order to maximize their usefulness in the alternate context of the CIC portal, they must be transformed in a variety of ways, including adding information appropriate to this new context, regularizing them to more fully combine the many different repositories into a single resource behind a single portal, and cleaning up messy metadata that may have been harvested. All of these tasks involve operating on the underlying XML expression of the record with fairly heavy computational machinery. A variety of approaches are possible for doing this.

A combination of XSLT stylesheets and script programs (used for operations not possible or difficult in XSLT) is the most readily accessible way and the one used in the CIC project, but other methods are also possible. Janée and Frew [3] describe a Python-based system in which a metadata transformation language is implemented allowing both declarative and procedural transformations to be done (obviating the need for two different types of components, but at the cost of having to learning a separate, non-standard language). Euzenat and Tardiff

---

<sup>1</sup><http://cicharvest.grainger.uiuc.edu>

describe a similar system in [1], designed to remain largely compatible with XSLT while offering more powerful transformations as well.

Such approaches, though quite interesting, are beyond the scope of this project. We instead sought to see if any efficiencies could be gained by making smaller modifications to the current process. As reported in [2, 296], reprocessing the entire aggregation takes approximately 30 hours, so finding such efficiencies would be a worthwhile endeavor. We first undertook some profiling of the existing system and then tried some basic changes to it which had the potential to create savings in processing time, looking specifically at the XML processing and some alternate means of performing it.

## 2 Current System

For our purposes we assume the records have been harvested and stored as files. A variety of XSLT stylesheets have been created that perform pieces of the reprocessing. For a given repository, a ‘driver’ program written in VBScript determines which stylesheets need to be applied to the records and reads these stylesheets into MSXML 4.0 objects. The individual records are then read in and the XSLT transforms applied sequentially to obtain the processed record, which is then written back to the filesystem.<sup>2</sup>We looked at two repositories for this project, which we shall abbreviate as linux01 and aerialphotos. First we ran some initial profiling.<sup>3</sup> For all timings reported we ran three runs of the the same test and took the average to account for any transient variations of the computer caused by background processes and so forth. All timings are wall-clock time measured to the second (because this is the only type of information possible to get from VBScript). First we inserted code into the driver program that would allow us to either run the entire original program, run the XML transformations but omit the writing of the processed records, or to simply read in and parse the records but omit all further processing. The aim of this was to determine approximately how much time was spent in I/O; if this represented the bulk of the processing time (as it does in some scenarios, e.g. [3, 310]) then attempting to improve the XSLT processing itself would be a relatively fruitless endeavor. As the following tables show, this was not the case, as XSLT processing occupied well over half of the total time:

Table 1: Timings for original processing (s)

	full	no write	no XSLT processing
linux01	509.3	316.0	19.0
aerialphotos	125.3	104.3	4.3

<sup>2</sup>Of course it’s a bit more complicated than that, and there is some additional processing afterward, but this is the bit relevant for this study.

<sup>3</sup>All timings are on a Dell XPS M140 computer running WindowsXP with 1GB of RAM, a 1.73GHz Intel Pentium M processor, and a 5400rpm ATA hard drive.

Table 2: Percentage of time in each operation

	input	XSLT processing	output
linux01	3.7 %	58.3 %	38.0 %
aerialphotos	3.5 %	79.8 %	16.8 %

It is interesting that, and unclear exactly why, the XSLT processing and output stages for the two repositories have such different characteristics. The input for the linux01 repository was 5757 files of 8.18MB and the output 5167 files of 69.2MB with about 193 seconds writing. The input for aerialphotos was 1934 files of 2.6MB and the output 1025 files of 26.7MB with about 21 seconds of writing.<sup>4</sup> Intuitively, neither the extra number of files nor the extra amount of data being written seems to justify such a large increase in the writing time. It is possible that the structure of the XML being written may explain it, but to the eye it does not look so different in the two cases that we would have expected to see something like this. In any case, this data does allow us to conclude that for the type of processing that we are doing here we are justified in trying to seek possible improvements - that I/O operations are not an overwhelming bottleneck.

Also at this time we measured the time that it took to run each individual stylesheet. We did this simply by sequentially removing stylesheets from the XML file that records which stylesheets to run for each repository, running the program after each removal, and recording the time through XSLT processing but without writing.

Table 3: Percentage of time in individual stylesheets

	linux01	aerialphotos
topelements	5.9	3.7
collections.expand	15.3	9.1
cleaning.expand	1.8	1.3
standard.expand	17.1	7.4
geog.expand	6.1	17.5
thumbnail.expand	16.0	11.8
resourcetext.expand	16.9	11.1
normalizens	5.5	5.7
removeDuplicates	15.4	32.2

Clearly the time taken depends very strongly on the structure and contents of the metadata being processed. It does not seem that we would be able to make any firm conclusion about where improvement might happen, but this does at least give a rough idea of which stylesheets are most important in terms of processing time.

---

<sup>4</sup>These figures are somewhat crude, as not every file gets processed and output, but should at least help illuminate matters.

### 3 Effects of changing driver language

The first change that we made was to change the language of the program driving the transformations from its original language (the interpreted VBScript) to a compiled language (Common Lisp, using Lispworks 4.4 personal edition). The actual XML processing is still performed by the MSXML library. This investigation was prompted by the observation that the insertion of the relatively simple profiling code described above caused a 3-4% slowdown in the execution time of the program. Though most of the work was seemingly being done by ActiveX objects and so should be independent of the implementation language, it seemed worth checking by translating to another language and measuring whether there seemed to be any time penalty paid for working in VBScript. The translated program tries to mirror the VBScript algorithm closely, although this was not completely possible.

Table 4: Timings with Lisp program (s)

	full	no write	no XSLT processing
linux01	487.0	329.7	14.0
aerialphotos	126.3	106.7	2.0

We see that the times are not very much different than the VBScript times (the linux01 is a little bit less), but there is an oddity with the linux01 repository. The write operation takes quite a bit less time, and the XSLT processing takes more. We do not have a good explanation for this phenomenon; since both of these operations are being performed primarily by the MSXML library code (the write entirely by that code), we did not expect to see this kind of discrepancy. Overall, it does not look like the choice of language to drive the XSLT processing makes too much difference in the processing time.

### 4 Effects of changing parser

Next we desired to change XML parsers and XSLT processing engines to see if that had any effect on the time taken. One might reasonably assume that different engines are implemented with greater or lesser overall efficiency. Moreover, the strategies employed by the engines in doing the processing may also have an effect depending on the nature of the processing. For example, in [4] a lazy<sup>5</sup> strategy for XSLT processing is demonstrated that is intended for XSL pipelines. Compared to traditional XSLT engines they see dramatic performance improvements for certain scenarios, notably when at some point in the pipeline

---

<sup>5</sup>Lazy here is analogous to the lazy/strict distinction in programming languages (e.g. between Haskell and ML). Roughly, in a strict computation, the result of a calculation is computed immediately before being passed on the next part of the computation. In a lazy computation, intermediate results are not computed until they are actually needed to produce some part of the output. A simple conceptual model is that this trades space for (potentially) time, but architectural considerations make it more complicated than that.

not many elements are getting through to the next stage. This makes intuitive sense; because of the lazy processing, only the prior calculations truly *necessary* to derive these few elements are performed, and much calculation that would be done and thrown away in a strict processor is avoided. This exact strategy is unlikely to be helpful in the CIC processing, as for the most part all the elements that existed in one stage are used in computing the next, but it does reinforce the idea that it is worth trying out different engines because behind the scenes choices in their implementation may strongly affect their performance on a given set of data.

The first change we made was to try the MSXML 6.0 library in place of the MSXML 4.0 library. We modified the original VBScript program to use MSXML 6.0 objects. In addition to instantiating these objects in place of the 4.0 version objects, a few other minimal changes had to be made, owing to security changes made to the newer library. In particular, when an XML document is being used as a stylesheet, its ProhibitDTD property must be set to false and its AllowDocumentFunction property to true. The original program did not set these at all (they may be new to MSXML 6.0). In addition resolveExternals must be set to true; in the original program this was set to false. With these changes the 6.0 library correctly transformed the documents in the same way as the original program. We used the same method to measure its performance.

Table 5: MSXML 6.0 timings (s)

	full	no write	no XSLT processing
linux01	508.0	337.3	19.0
aerialphotos	132.3	111.0	4.3

The total times are close. It appears that for the aerialphotos repository there was a bit more processing time and that the other stages were about the same. For the linux01 repository the total time was almost the same, but for some reason there was more processing time and less writing time required. From these two examples it would seem that we cannot expect MSXML 6.0 to work more quickly, although clearly it is doing something differently.

Finally we examined an entirely different XSLT engine, the Java-based Saxon transformer (version 8.7) running under the JRE 1.5.0. The XML parser used was Xerces. This required something of a combination of the above two experiments; as well as using a different parser/transformer we had to reimplement the driving program yet again in Java. There were some technical difficulties getting an exact analog to the VBScript program, so we present two variations that at least give partial information about Saxon's performance. We would have liked to have also tested the Xalan engine (also a Java engine). In theory this should have been easily done as a drop in replacement to Saxon, but their were real-world problems with doing this, as the same programs that worked properly with Saxon exhibited some difficulties with Xalan.<sup>6</sup>We will be content

---

<sup>6</sup>These included some problems with the namespaces and interestingly what appeared to

to present the results obtained with Saxon.

In the course of porting the program we uncovered a small issue with the stylesheets. In the stylesheets the “\” character had been used as a path separator, and the MS parsers accepted this. Saxon demanded “/” as the path separator. With this change the stylesheets worked properly.

The first step was simply to get a working Java program that did the transformations properly as a basis for further experiments. Some false starts trying to replicate the previous method (storing intermediate results in computer objects representing DOM documents) were unfruitful, so to get a working program we resorted to serializing intermediate results along the pipeline to strings. Naturally this produces extra parsing overhead and so should not be expected to be the optimal method, but it did lead to a correct program.

Table 6: Saxon timings, stringifying method (s)

	full	no write	no XSLT processing
linux01	630.0	598.7	16.0
aerialphotos	257.0	253.0	5.0

Unsurprisingly, both repositories took longer with this method and processor, the linux01 by about 25% and aerialphotos by about 100%. We speculate that the greater average size of the records in the latter repository caused it to increase significantly more; recall that with this processing method the XML parser has to reparse the record before application of each stylesheet, and the record has to be converted to a string again after each transformation is done.

A final method we tried with the Saxon parser is the “Pipe” method. The Java XML API allows construction of an XSLT pipeline where the output of each stylesheet is connected automatically to the input of the next, with the initial reading of the XML record serving as input to the first stylesheet and the output of the last sent to a stream (here one connected to a file). Exactly how the information is being passed from one stylesheet to the next is opaque to the user, so in this case it makes less sense to measure the individual parts of the process (since it is being set up and run as an indivisible unit), so we just report overall times.

Table 7: Saxon timings, Pipe method (s)

	full processing time
linux01	613.7
aerialphotos	235.0

These are a little bit better than the stringifying method times, but not much so. The method is still somewhat crude, in that there is no compilation of the pipeline as a whole.

---

be a memory leak.

## 5 Conclusions and further directions

The parts of the current system that we tested seem to work relatively efficiently. None of the changes that we tried had a significant positive effect on the processing time. While it's disappointing not to have found a wonderful change that would have made a difference, it is heartening to know that the current system is good. And if the situation were reversed and the original system had been, say, a Java system, we actually would have found a possible improvement. The differences found do point out the desirability of being able easily to test different transformation methods so that as existing technologies change and new ones become available one can easily test them to determine if they offer any benefit. Though we have not produced it here, it would be ideal, for example, to have a system into which one could easily drop various transformation engines and profile their performance, instead of having to make special arrangements to test each one. Unfortunately, the lack of easy interoperability between the various technologies makes this a difficult task.

One notable difficulty with XSLT processing is the seeming lack of a way to optimize *pipelines* of transformations. When the various engines compile stylesheets in some way, they are only doing them individually; there seems to be little provision for specifying a series of transformations and then having a compiler analyze the *entire* pipeline to search for efficiencies. The techniques used in [4] are similar in spirit to this, and a specialized system as described in [3] might also allow such optimization work to be done, so perhaps we will see this enter the mainstream a bit more in the future.<sup>7</sup>

This independent study allowed us to get a better feel for the technologies and techniques underlying large-scale metadata processing. Clearly there are many interactions between LIS concerns and software engineering in this sort of project, and probing into a few of the specifics of what happens and why helped reinforce this. It was somewhat unfortunate that we lacked the time to investigate the actual XSLT stylesheets themselves more thoroughly, or to investigate the trade-offs between expressing desired metadata transformations via such stylesheets versus directly in code. Becoming aware of such issues and thinking about the trade offs involved has nevertheless been useful. As computerized processing of metadata becomes more and more important, having a supply of people familiar with the issues on both sides of the fence will be also grow in importance to the library and information community.

## References

- [1] Euzenat, J. & Tardiff, L. (2001). XML transaction flow processing. *Markup Languages: Theory and Practice*, 3(3), 285-311.

---

<sup>7</sup>Systems like Apache Cocoon seem to be intended for XSLT pipelines, but it was unclear that even they do any of this sort of optimization. That might be worth looking into more carefully as well.

- [2] Foulonneau, M. & Cole, T. (2005). Strategies for reprocessing aggregated metadata. *Research and Advanced Technologies for Digital Libraries. 9th European Conference, ECDL 2005. Proceedings (Lecture Notes in Computer Science Vol. 3652)*, 290-301.
- [3] Janée, G. & Frew. J. (2005). A hybrid declarative/procedural metadata mapping language based on Python. *Research and Advanced Technologies for Digital Libraries. 9th European Conference, ECDL 2005. Proceedings (Lecture Notes in Computer Science Vol. 3652)*, 302-313.
- [4] Schott, S. & Noga, M. (2003). Lazy XSL transformations. *Proceedings of the 2003 ACM Symposium on Document Engineering*, 9-18.