Improving Pattern Matching Performance in XSLT

John Lumley

jωL Research & Saxonica

<john@jwlresearch.com>

Michael Kay

Saxonica

<mike@saxonica.com>

Abstract

This paper discusses improving the performance of XSLT programs that use very large numbers of similar patterns in their push-mode templates. The experimentation focusses around stylesheets used for processing DITA document frameworks, where much of the document logical structure is encoded in `@class` attributes. The processing stylesheets, often defined in XSLT1.0, use string-containment tests on these attributes to describe push-template applicability. For some cases this can mean a few hundred string tests have to be performed for every element node in the input document to determine which template to evaluate, which in sometimes means up to 30% of the entire processing time is taken up with such pattern matching. This paper examines methods, within XSLT implementations, to ameliorate this situation, including using sets of pattern preconditions and pre-tokenization of the class-describing attributes. How such optimisation may be configured for an XSLT implementation is discussed.

Keywords: XSLT, Pattern matching

1. Introduction

XSLT's push mode of processing [5], where templates are invoked by matching XPath-based patterns that describe conditions on nodes to which they are applicable, is one of the really powerful features of that language. It allows very precise declarative description of the cases for which a template is considered relevant, and along with a well-defined mechanism of priority, and precedence, permits specialisation and overriding of 'libraries' to encourage significant code reuse. Whilst other features of XSLT are valuable, push-mode pattern matching is almost certainly the most important.

Consequently much effort has been expended on developing XSLT-based processing libraries, for many types of XML processing, most notably in 'document engineering', such as DocBook and DITA, which use pattern-matching templates extensively. Typically a processing step might involve the use of hundreds of templates which have to be 'checked' for applicablity against XML nodes that are being processed in a push fashion. One of the challenges for the implementor of an XSLT engine is to ensure that for most common cases, this matching process is efficient.

Various aspects of overall XSLT performance have been studied and reported [1] [3] including optimization rewriting [2]. In this paper we will examine some cases where, owing to the nature of the XML vocabularies being processed and the design of the large XSLT processing stylesheets employed, the *default matching process* in one XSLT implementation (Saxon) can be rather expensive, in some cases taking about a third of all the transform execution time. We'll discuss possible additions and modifications to the pattern-matching techniques to improve performance and show their effects. Some knowledge of XSLT/XPath is assumed.

The paper is organised as follows:

- We first describe “push-mode” processing in XSLT and what the process for matching template patterns is in detail.
- How this process is performed in the Saxon implementation is presented, along with some general remarks about the problems of many templates matching predicated generic, as opposed to named, elements.
- We discuss in detail measurements of the pattern matching performance when processing a large sample DITA document.
- Possible improvements using sets of *common preconditions* to partition applicable templates are outlined and performance measurements using a variety of these tactics in processing the sample are discussed.
- Some other approaches, involving more detailed knowledge of stylesheet expectations are discussed briefly.

doi:10.14337/XMLLondon15.Lumley01

- We speculate on methods to define and introduce such tuning features into an XSLT implementation.

2. XSLT push mode

XSLT's push-mode of processing takes a set of items (usually one or more nodes from the input document such as elements, attributes or text) and for each finds a stylesheet template declaration whose pattern matches the item (the "context item") in question. Assuming one is found, the body of the template, which can contain both result tree fragments and XSLT instructions, is executed to generate a result which is then added to the current *result tree*. This push mode is often exploited highly recursively, descending large source input trees to accumulate result transformations, usually as modified trees. To understand the problems with large sets of such templates, which is often the case in industrial-scale document processing applications, we need to describe more closely what this pattern matching process is.

When processing a candiate item ("context item") in XSLT via the `xsl:apply-templates` instruction the following is the effective declarative procedure:

1. All the templates that have `@match` patterns and operate in the "current" mode are considered candidates.
2. For each template so chosen the pattern (which is a modified form of an XPath expression) is tested against the context item to determine a boolean value. Only those yielding true are considered.
3. The templates with the highest *import precedence* are retained. (Templates in an imported, as opposed to included, stylesheet have precedence lower than those in the stylesheet that declares the importation, or any following "sibling" imported stylesheets.)
4. From these only those with the highest explicit or implicit priority are considered. (Patterns have an implicit priority level calculated based on a 'specificity' formula, such that more specific cases (e.g. `piece[@class = 'my.class']`) have higher priority than less specific ones (e.g. `piece`), and thus supporting a natural style for general and specific case programming. Rules can override this by declaring an explict priority.)
5. Members of the remaining set of templates are all potential candidates:
   - If the resulting set is empty, the result is an empty sequence.
   - If it has just a single template member, then the result of the `xsl:apply-templates` is the (non-error) result of executing the sequence constructor of that template with the tested item as the context item.
   - If the resulting template set has more than one member, then it is an implementation choice as to whether a dynamic error is thrown[1]. If not, then the last in "sequence" is chosen and its body executed.

What this list doesn't prescribe is how the process is to be implemented. Clearly there are a number of possibilities of improving performance, by for example examining candidate templates in a suitable order, or pre-classifying subsets of the possible templates. In this paper we examine some possibilities which look deeper into the template patterns themselves.

3. Template rules in Saxon

The algorithm used for matching template patterns in the Saxon processor has been unchanged for many years [1], and works well in common cases. In simplified form, it is as follows:

For template rules whose match pattern identifies the name of the element or attribute to be matched (for example `match="para"`, or `match="para[1]"`, or `match="section/para"`), the template rule is placed on a list of rules that match that specific element or attribute primary QName . Other rules are placed on one of a set of generic lists, arranged by type (`document-node()`, `text()`, `element(*)...`) . Both the named and generic lists are ordered by "rank", a metric that combines the notions of import precedence, template priority, and ordering of rules within the stylesheet.

When apply-templates is called to process a specific element, Saxon finds the highest-ranking matching rule on the list for that specific element name, and also the highest-ranking matching rule on the generic list[2]. It then chooses whichever of these two has higher rank. The search of the generic list is abandoned as soon as it can be established there will not be any matching rule with higher rank than a rule found on the specific list. But note that, as we'll see later in our example, once a match has been found in either the specific or the generic list, that list *must still be searched* for other matching candidates of similar rank.

In current versions of Saxon each pattern in the rule chain is examined in turn, to determine a boolean matches value. Of course the boolean match processing is

[1] In XSLT 3.0 this choice can be controlled via stylesheet declarations.
[2] A template can be referenced from multiple lists when its match conditions contains a union of patterns.

processed lazily, and in strict sequence, with falsity propagating as quickly as possible, so for example, ancestor patterns are only examined if the `self::` pattern proves true.

For very many stylesheets, this works well, because most rules specify an element name, and there are typically not many rules for each element name, so typically each element that is processed is matched against at most half a dozen rules on the specific list for its element name; usually the generic list does not need to be considered, because rules on the generic list usually have lower rank than rules on the element-specific list.

The algorithm becomes ineffective, however, when the stylesheet defines very few rules that match specific element names, and many rules that are generic. Typically such stylesheets match elements according to predicates applied to their attributes, rather than matching them by name. A worst-case example of such coding can be found in the DITA-OT family of stylesheets[1]. Consequently these have therefore been used as a test case for exploring improvements to Saxon's pattern matching algorithm. This is even more problematic when the stylesheets use a large range of import precedences, as is also true in the example, where as we'll see there are some 35 different well-populated ranks. Moreover even the named lists gain little as time is dominated totally by checking the unnamed sets.

3.1. Generic match patterns

As hinted above, the presence of a lot of patterns of the form `*[ predicate ]` mean that measures such as indexing patterns on primary element name become ineffective. Are there other indexation schemes that can be employed? Clearly if we have many match patterns of the form:

```
*[@x = 'a']
*[@x = 'b']
*[@x = 'c']
```

and it is possible to determine statically that these predicates are mutually exclusive, then should be possible to construct a hash index whereby we can lookup the value of attribute `@x`, and directly determine which of the patterns applies.

Unfortunately real life is more complicated. A framework where almost all stylesheet template patterns match generic rather than named elements is DITA-OT, where the patterns take the form

```
*[contains(@x, 'a')]
*[contains(@x, 'b')]
*[contains(@x, 'c')]
```

As it happens, in DITA these patterns are designed to be mutually exclusive, so only one of them will ever match. But there is no way an optimizer can know that and we cannot thus meaningfully generate indices. So if we are going to avoid a sequential search of all the patterns, we need a different approach. The rest of this paper examines what this might be, after discussing a specific DITA document processing example in detail.

4. Processing a DITA document

The DITA-OT framework is a set of mainly XSLT (1.0/2.0) tools for processing DITA documents, used extensively in automatic generation of technical documentation. DITA itself describes document components in XML trees, using the `@class` attribute extensively, with class membership described through a whitespace-separated set of class names. (For a fuller description, see Class attribute syntax in the DITA documentation.)

One consequence of this is that many of the processing templates within the DITA-OT framework describe applicability through a match "is this element in class *xxx* ". Unfortunately, within an XSLT1.0 context, where tokenization isn't present, this is typically described as a predicated match pattern

```
*[contains(@class,' classname  ')]
```

(Additional leading and trailing spaces are added to the class attribute value to support this generic `contains()` match.) A little thought would suggest that when a large number of such patterns all compete to examine the `@class` attributes of pretty much every element in a document then the pattern-matching process might be very expensive. And so it turns out.

Note

This predicated match pattern appears so frequently throughout this paper, that an abbreviation `@C{ classname }` will often be used in tables and figures to replace this construct.

While examining a (different) DITA processing performance issue for a client, Saxonica carried out some measurements on the size of this pattern-matching problem. The chosen situation was one of the stages of expansion of a DITA document into a PDF result, via an XSL-FO route. In particular we examined the conversion of a fully formed DITA source into XEP-specific XSL-

[1] By contrast the Docbook https://github.com/docbook/docbook toolsets, of similar size, use almost entirely named element pattern matches.

FO source (target `transform.topic2fo.main` in the DITA-OT build script architecture). Through most of the rest of this paper, we'll study in detail the processing of a particular source document through this XSLT transformation.

4.1. Source document and transform

The processed document had the following characteristics:

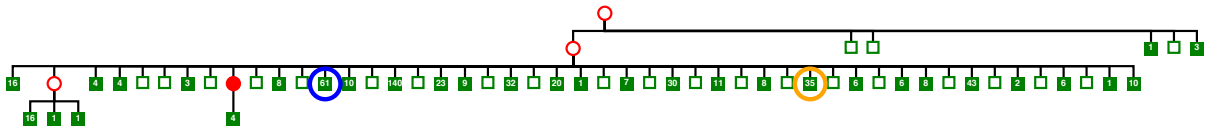
- An 80 page specification for a electronic component, involving lots of tabular descriptions of bit-significances etc.
- Source file: 2.66 MB (246kB of redundant whitespace)
- Source XML tree: 13,066 elements, 46,831 attributes, 6,093 non-whitespace text nodes – total 65,990 significant nodes; tree depth: maximum: 13, average: ~10; sibling width: maximum: 57, average: ~2.
- Result XML tree: 19,441 elements, 91,048 attributes, 6,140 non-whitespace text nodes
- Final PDF document: ~80 pages.
- The document is very table-heavy – 262 tables with 1,309 rows and 4,863 cells, i.e. almost 50% of all the document elements describe table components.
- All bar two elements of the source document contain a `@class` attribute, and there are 43 different values for its text value, the most frequent of which is used 3,683 times.

The processing XSLT transformation had the following characteristics:

- 58 source files.
- 70 pattern-matching modes.
- 418 pattern-matching templates, 155 named templates. (258 of the matching templates are in the `#default` mode.)
- 5 user-defined functions.
- Of the 58 source files, only 33 contain templates. Those that don't either act as importation expanders or contain global parameters or attribute sets.
- All stylesheet connection is via `xsl:import`; there is no use of inclusion and no multiple importation. The importation tree is relatively shallow, at most with a depth of 4 and looks like Figure 1. DITA-OT stylesheet importation tree for DITA→FO.

  where solid nodes indicate stylesheets that contain templates and circles denote stylesheets that import other stylesheets. (The two leaf nodes circled are stylesheets containing the most frequently used templates. The consequences of this are discussed in Section 6.1, " "Un-disambiguating" rules".)

Figure 1. DITA-OT stylesheet importation tree for DITA→FO



4.2. Processing characteristics

Using Saxon 9.6EE on a quad-core i7 1.6 GHz laptop running 64-bit Windows7 with 4GB of RAM, the processing took around 15 seconds. But of more interest are some of the internal statistics. In processing this document to completion, 75,950 template rules 'executed', i.e. their patterns matched and their sequence constructor bodies were processed further[1]. (This is comensurate with a model where most nodes are only 'touched' once during processing.) Determining which rule to execute at each stage took approximately 4.5 seconds, i.e. *~ 30% of all processing involved template pattern matching.*

Of the 70 pattern matching modes in the source XSLT, only 35 were active on this document and only three have significant performance impact, accounting for 96% of all the calls and 99.7% of all the time taken matching template patterns.

Table 1. Significant Modes

Mode	Purpose	# invocations	% invocations	time / ms	%time
`#default`	General	13,095	17.2	4,330	97.8
`toc`	Table of Contents	22,088	29.1	51	1.1
`bookmark`	Bookmarks	37,752	49.7	33	0.8

Whilst the proportion of the number of invocations depends upon characteristics of the document and the DITA-OT framework, the performance costs per node depend upon the complexity of the patterns involved. Thus if we examine the patterns for the `#default` mode, it immediately becomes apparent why there is such disparity.

Table 2. Mode Patterns

Mode	Purpose	# template patterns in mode			#templates matched
		element(*)	element(*named*)	attribute(*named*)	
`#default`	General	240	19	8	39
`toc`	Table of Contents	2	4	0	3
`bookmark`	Bookmarks	2	5	0	3

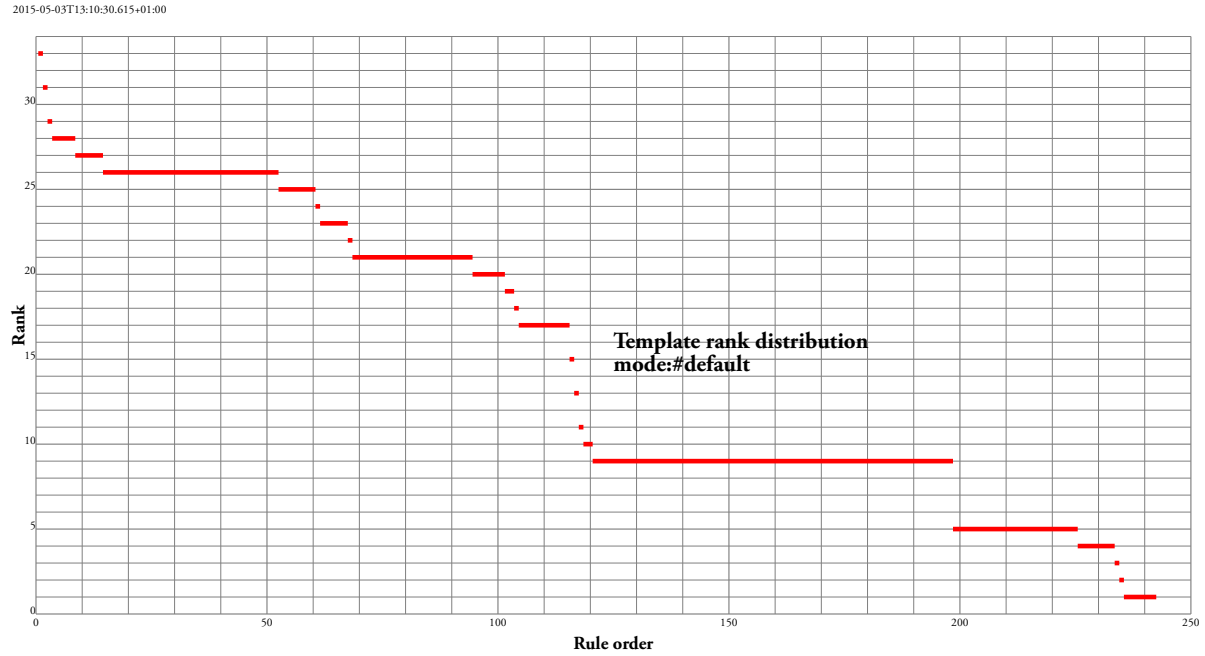
Clearly the number of the template rules that need to be checked for unnamed elements (`*`) in the `#default` mode dominates. How do these 240 templates differ? Firstly as the DITA -OT framework uses a large number of files via `xsl:import` inclusions, they have strongly differing precedences [2]. Template match patterns also have differing implicit or explicit priorities. Together these two properties constitute a *rank*, precedence before priority - matching higher rank patterns are chosen over lower. When multiple patterns match at the same rank optionally either the last in sequence is chosen or a dynamic error is thrown.

In this case, the 240 templates are spread across 25 different ranks, with the sequence order distribution shown in Figure 2, “Template rule order and precedence ranking”.

[1] A union pattern (`pattern1 | pattern2`) is considered to be a set of separate matches for this analysis - one for each pattern.

[2] There is no use of the `xsl:apply-imports` instruction within the framework, implying a simple overriding model. `xsl:next-match` is not used either, though that wasn't present in XSLT1.0

Figure 2. Template rule order and precedence ranking



Missing ranks involve templates matching named elements and attributes. The overall total of 35 ranks is in line with the approximately 33 imported stylesheets of the framework. Within a rank rules are ordered in reverse document order as when ambiguous rules are permitted *later* rules are chosen; hence they are placed earlier within order within a rank. Details for the most heavily used ranks are given in the following table:

Table 3. Heavily populated pattern ranks in mode #default

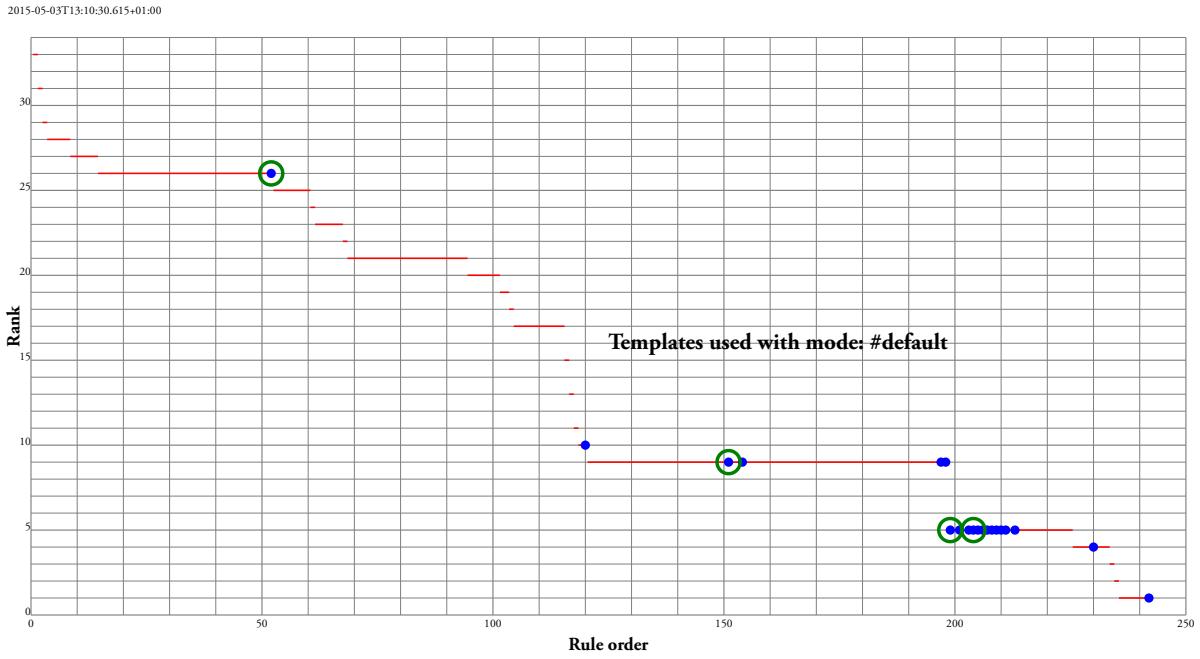
Rank	27	26	25	23	21	20	17	9	5	4	1
start	9	15	53	62	69	95	105	121	199	226	236
end	14	52	60	67	94	101	115	198	225	233	242
size	6	38	8	6	26	7	11	78	27	8	7

Rank 5 contains mostly templates associated with tables, from the stylesheet `tables.xsl` and these table-matching templates appear to be unique, i.e. no templates in other stylesheets would be anticipated to match. Rank 26 (from `pr-domain.xsl`) contains templates for the programming domain.

As already remarked, all templates of a given rank are candidates to match in preference to those of a lower rank. Thus for example, when processing a node whose correctly matching template is that with order number 150 and rank 9, all the 122 templates of higher rank must be eliminated, *and all the 78 templates of equal rank tested for possible conflict*, i.e. a total of just under 200 template match conditions must be examined.

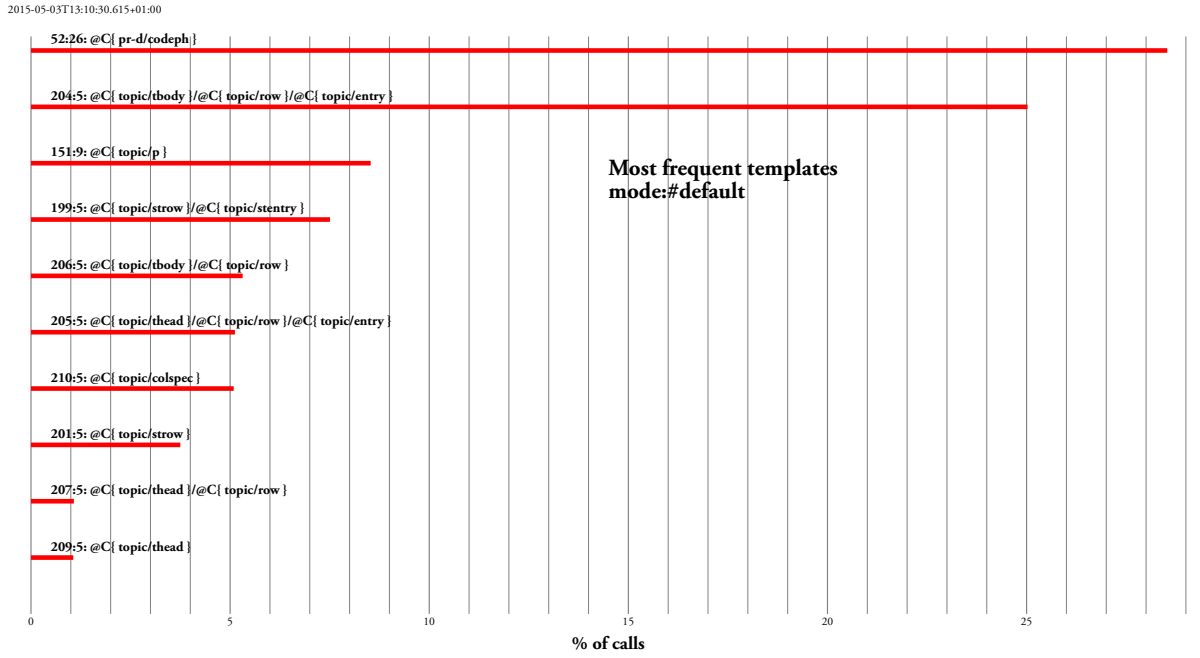
We've examined the rank ordering of all the templates used in the `#default` mode, but which are actually matched within the processing of this sample document? Figure 3, “Rule order and precedence of matched templates” shows the distribution of the 39 templates that were invoked.

Figure 3. Rule order and precedence of matched templates



The blue dots indicate order/rank of matched templates, the green circles surround the four most frequently matched of these, the implications of which are discussed below. What are the most frequently matched templates? Figure 4, "Most frequently matched templates" shows the percentage of all matches taken by the ten most significant, labelled with rule order, rank and (abbreviated) match pattern.

Figure 4. Most frequently matched templates



For this document the most commonly matched template in the `#default` mode, accounting for 28% of unnamed element matches, has order number 52 (it matches `pr-d-codeph`) but the next most called (25%, matching topic table entries) has order number 204 and rank 5. (Their positions are circled in green in Figure 3, "Rule order and precedence of matched templates".) The next 6 most commonly matched templates account for 35% of calls collectively and are in ranks 9 and 5.

So we have the situation that whilst for 28% of the successful element matches 50 patterns must be checked each time (i.e. the end of rank 26), for more than 60% of the matches, either 200 or 225 patterns must be checked.

Thus far we haven't looked at what these patterns are, merely their required order of checking. Let's examine the top seven unnamed element patterns in the `#default` mode:

Table 4. Most frequent patterns in mode #default

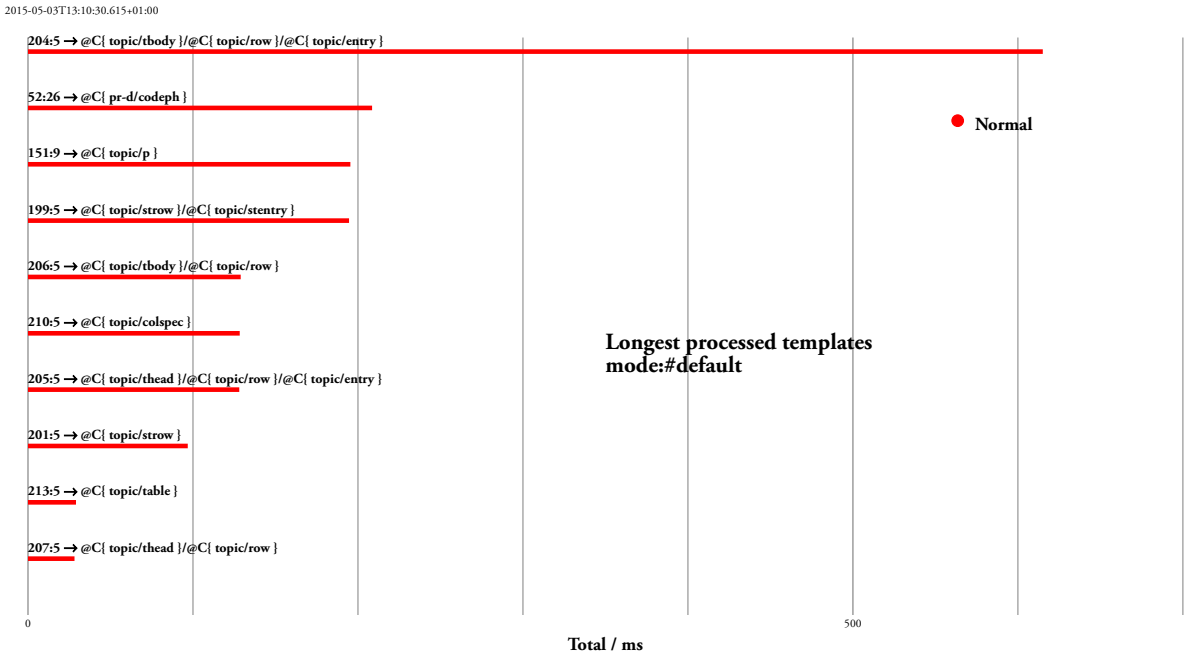
Order	Rank	% calls in mode	Pattern
52	26	28.5	`@C{ pr-d/codeph }`
204	5	25.0	`@C{ topic/tbody }/@C{ topic/row }/@C{ topic/entry }`
151	9	8.5	`@C{ topic/p }`
199	5	7.5	`@C{ topic/strow }/@C{ topic/stentry }`
206	5	5.3	`@C{ topic/tbody }/@C{ topic/row }`
205	5	5.1	`@C{ topic/thead }/@C{ topic/row }/@C{ topic/entry }`
210	5	5.1	`@C{ topic/colspec }`

(`@C{ xxx }` is an abbreviation for `*[contains(@class,' xxx ')]`) Here we can see the problem - each pattern must perform 'free-position' string matching on an attribute of at least the element node under test and sometimes within one or even two ancestors. And it turns out that of these 240 template rules, *all bar one* of them have a similar form[1]. So for templates whose order is 200+, more than 200 other string matches of very similar form have been performed, on *every element* processed through `xsl:apply-templates`

When we look at the amount of time consumed the picture is similar.

[1] We are somewhat puzzled by the redundancy in representation present – table entries are represented both by an element `entry` and a token within `@class`. Are there circumstances where a table cell is *not* represented by an element `entry`? If this is not the case, then why use the `*[....]` pattern rather than one keyed on the element QName? We understand that support for extensibility of element-vocabulary was one reason. Far be it for the authors to criticise the design choices of DITA in its representation of class membership, or the DITA-OT framework for the processing architecture, but *this is no way to run a railway*.

Figure 5. Pattern matching time for the most frequently matched templates



Given the architecture where class membership is described in the `@class` attribute, an obvious question is whether in practice elements can be members of multiple classes. The input clearly shows this to be so – 3,864 of the 13,066 elements have multiple class "tokens", the vast majority being - `topic/ph pr-d/codeph`, which according to the DITA reference, declares that the given element is a structural element equivalent to both a phrase in a generic topic and a code-phrase in a programming domain.

Clearly for this type of stylesheet the issue is that very many of the templates, being processed independently, have to carry out the same sort of operation multiple times on the same node. What methods might be available to reduce the processing, preferably to a minimum? In the rest of this paper we discuss some possible improvements of the following general types:

- Rule preconditions: determining common boolean preconditions that must be satisfied for a (large) number of rules to possibly match – the results for a specific node can be cached and rules that are bound to fail can be excluded rapidly. These approaches have the property that they are *heuristic* in performance improvement but retain correct stylesheet behaviour.
- Other methods that require *oracle* guarantees about the stylesheet behaviour, effectively allowing short-cuts which are not generally applicable to all stylesheets. The cases include:
  - Suspending rule ambiguity checking, and potential template/stylesheet reordering.
  - Pre-tokenizing suitable properties: exploiting higher-level knowledge in replacing patterns with access to deeper structure.
  - Using `key()` structures.

5. Preconditions

With long sequences of patterns, many of which have some similarities, one possibility is to detemine a smaller set of precondition patterns that can be tested as required for a node before the "main" pattern is checked. The value of such a precondition for a given node can be computed only when required (i.e. the first time when a pattern that uses that precondition has to be checked) and stored to avoid subsequent recomputation. The hope of course is to eliminate quickly higher-rank patterns that cannot match as one or more of their preconditions has already been determined to fail. For example if there were a large set of templates with matches of the form `chapter/` *`node-condition`* , such as:

```
chapter/title[condition1],
chapter/title[condition2],
chapter/para,  chapter/section ...
```

then they all share the requirement that `exists(parent::chapter)` must be true for the pattern to match. Thus computing whether this precondition is

satisfied for a given node *once* may aid performance in several possible ways:

- If a node does *not* have a `chapter` parent, then this need be determined only *once* for the node and each of these templates can be ruled out immediately.
- The pattern might be partially-evaluated within the context that the precondition is true, e.g. precondition(`exists(parent::chapter)`) reduces the patterns to

```
precondition:exists(parent::chapter) :
   title[condition1], title[condition2],
   para,  section ...
```

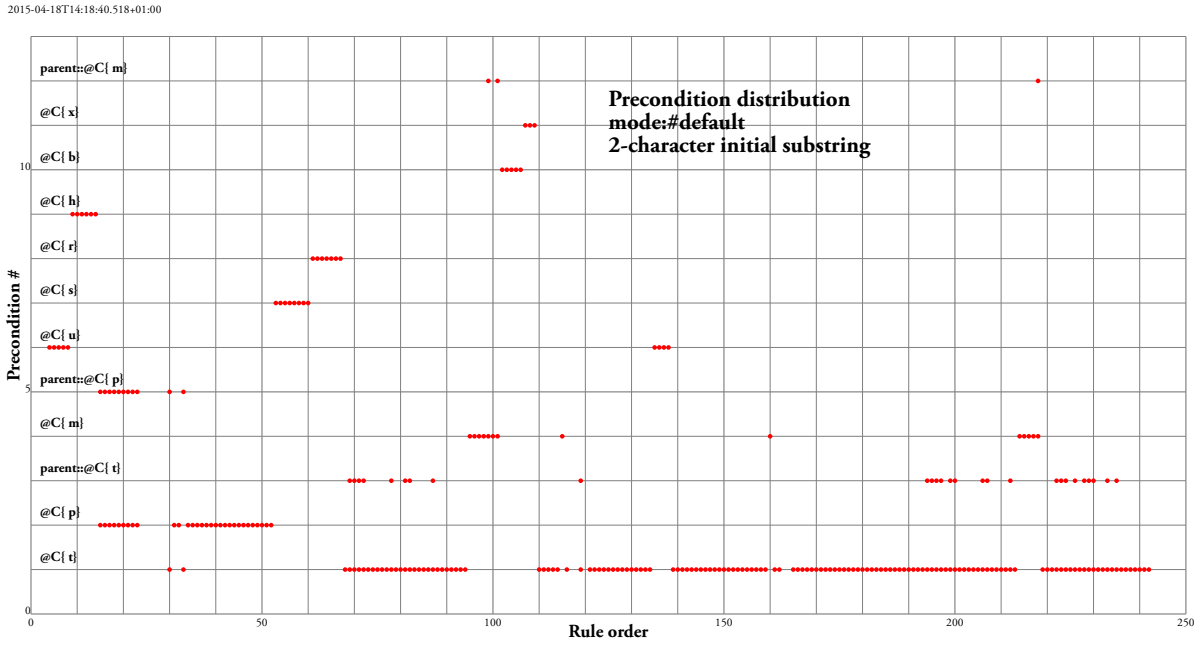
In our DITA example, using `exists(@class)` will gain us little – almost every element in a DITA document has a `@class` attribute and 90% of all element matches predicate upon it. We could choose to use the `contains(@class,...)` as a more discriminating condition. Of the 239 template rules that share that test for the context item (the node under test within `xsl:apply-templates`) there are 204 distinct values for the check (the largest common set has just 7 members, most have of course 1).

In some cases these preconditions might be common, especially when tested on ancestor nodes. For example rules #204 and #205 both test for `topic/entry` on the element and `topic/row` on the parent, differing only whether the grandparent is a table body or head. In this case, and especially with our sample document which is *very* table heavy, the precondition "pair" `contains(@class,' topic/entry ') and contains(../@class, ' table/row ')` might be beneficial (tested on #204, but result available for #205), but it is admittedly a very special case. What other more general preconditions might be useful?

A necessary precondition for `contains(@class,string)` is `contains(@class,any-substring-of(string))` so some expression of this form might be useful. Choosing to use just the first character of the comparator string, which might normally be expected to be of some use in many cases, fails miserably here, as due to the class representation model being used within DITA-OT, a leading space is appended to the class token comparand (effectively implementing an equivalent of `tokenize(@class,'\s+') = 'entity-class'`) – no gain there then.

If we choose to use the first two characters, we get 12 different preconditions, three of which apply to a `parent::*` context. The distribution of the use of these preconditions across the rule order is shown in Figure 6, "Distribution of 2 character initial substring preconditions":

Figure 6. Distribution of 2 character initial substring preconditions



Note that these preconditions are not mutually exclusive – some of the compound cases involve two conditions, one on the context element and the other on its parent. This is the case for 43% of the element template matches on the sample document. By changing the fixed-length

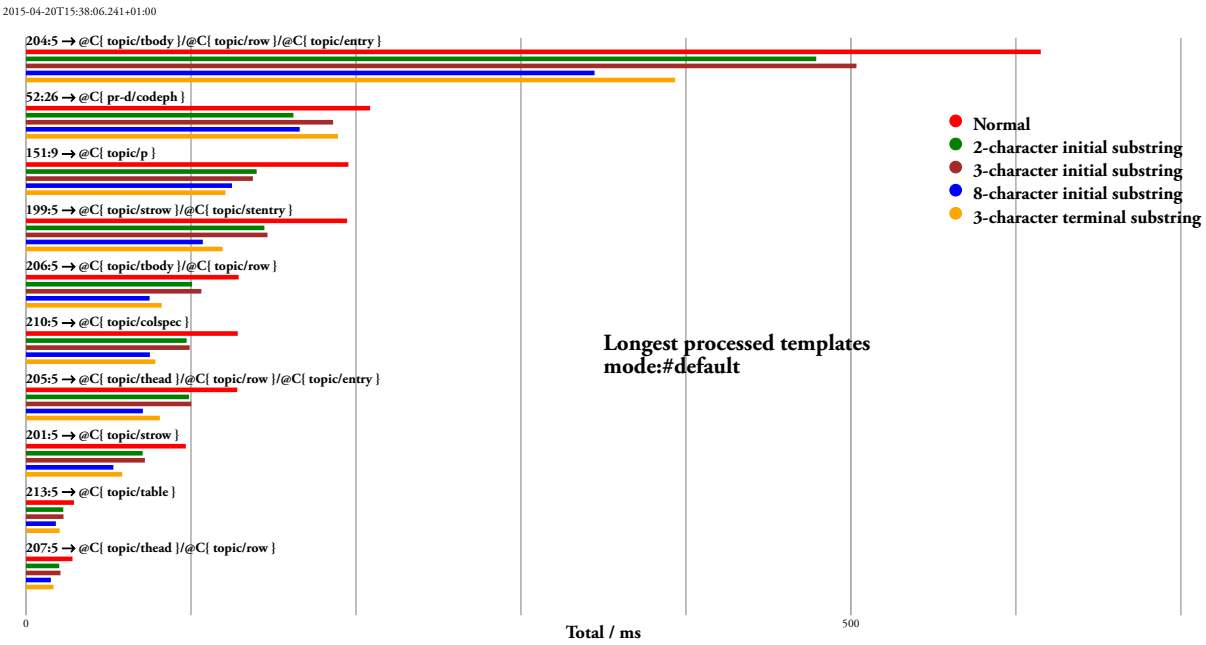
initial substrings used we get a variety of balances between precondition group sizes:

Table 5. Precondition group sizes as a function of initial substring discriminant

Substring length	# preconditions	Largest reference set
2	12	146
3 - 5	14	121
6	16	121
7	46	121
8	75	17

We can modify our applicable rule search such that each rule has a set of required preconditions (described by a list of indices into a cache of expressions and boolean values) which are tested before the main match is then processed. The rest of the rule list processing machinery is unaltered. The effect on the performance is shown in Figure 7, "Effect of differing substring preconditions":

Figure 7. Effect of differing substring preconditions



Note that when we are using these substring preconditions, there is little we can do to "pre-evaluate" the patterns themselves. Just because `contains(.,'abc')` is a precondition for `contains(.,'abcdef')`, does not of course imply `contains(.,'def')` is now sufficent, unlike in other cases where a "true" condition reduces the expression.

Now of course in this example there is some implicit structure in the class token using the solidus (/)[1]. Unfortunately this insight gains us little - using the substring before we get 14 groups, using the substring after we get 197! Obviously a better approach is to use a more infomation-theoretic partitioning. For example 121 template rules match a class substring `' topic/...` but adding one more character to the discriminant for this case replaces this group with 19 subgroups, the largest being 12-15 in size.

There is nothing that says we are restricted to initial substrings. Choosing the last 3 characters (which of course end with a space) gives us 53 different preconditions, with the largest group being 19 in size. The effect is similar, as is shown in Figure 7, "Effect of differing substring preconditions".

A recursive approach (extending the length of the substring for a particular group until subsequent subgroups are smaller than a proportion of the size of the

[1] It denotes DITA element equivalence to a given base element within a given topic.

original set or some minimum size) can give us a suitable partitioning. Doing this for this for 5% or a minimum of 30 gives us 41 preconditions, with a largest reference group of 26. A general rule of thumb suggests that when the number of groups is similar to the size of each group, the testing workload should be minimised.

On a more information-theoretic basis, we could generate some optimal partioning tree. However we have an issue that currently this would be done at compile-time, when, whilst we know the frequency of pattern components spread across the template spaces, we don't know the relative frequencies of execution on documents at run time. A possibility might be to collect statistics from representative training runs, which are then used to tune a subsequent compilation[1].

The performance figures above use Saxon's default rule list representation with precondition references added to the rules. Another possibility is to split the rule list into a number of separate lists where within each sublist rules which would *fail* a common precondition have been eliminated, then choose between the different lists at search start, based on checking a small number of those preconditions. For example, consider the two most frequent preconditions in Figure 6, "Distribution of 2 character initial substring preconditions" – `@C{ t}`, which qualifies 146 of the 240 rules and `@C{ p}` which qualifies 30. Satisfying the first condition does not restrict matches to just those rules which have that precondition; given the nature of the `contains()` function (and DITA's possibility of multiple class tag values) it would be entirely possible for one of the rules having the second precondition to match also. Rather the *failure* of `@C{ t}` rules out all those 146 rules, leaving a list of just 94 to be checked.

So now we have to look at the inverse of the problem. Table 4, "Most frequent patterns in mode #default" shows that just six patterns, all predicated on `@C{ t}`, account for at least 56% of all the template matches. Hence failure of this condition (which as we've seen limits the rules to be checked to 94) would only be invoked for a maximum of 45% of all calls. Failing `@C{ p}` will be frequent of course (for 70% of the elements), but it only eliminates 30 rules, albeit at high rank order.

6. Other possibilities

Using preconditions, as described above, does not change the correctness of the stylesheet behaviour under any circumstances. However, if the stylesheet designer can make certain guarantees about the overall stylesheet operation, then there are a number of other possibilities we might consider

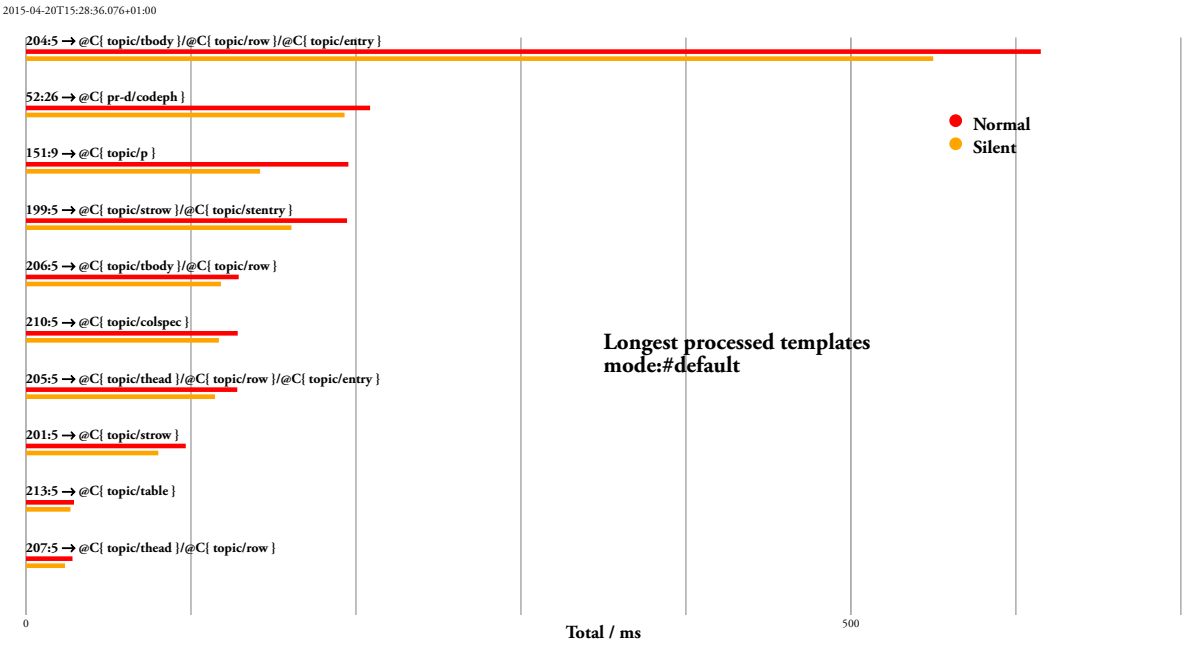
6.1. "Un-disambiguating" rules

We have noticed that in the execution of this stylesheet on the example source document, there actually is no ambiguity in applicable rules – all template patterns for a given precedence/priority rank have mutually exclusive patterns *on the nodes present in the example DITA document*. Hence we can assume that once a pattern for a template matches, all others of similar rank can be discarded. In effect we are suspending the checking of rule ambiguity, and provided that the mutual exclusivity is true for all practical purposes, which cannot be determined statically, the stylesheet will still continue to function correctly.

In our example, where there are many templates sharing the same rank (e.g. the 78 of rank 9, or 25 of rank 5) we can eliminate further search to "rank end". The effect may be slight but can be worth exploring. For our example we get the following:

[1] XSLT's package delivery mechanism might be a useful aid to this.

Figure 8. Effect of removing rule ambiguity



The effects are most marked on rules #151, where the number of rank 9 rules to be tested drops from 78 to 31 (total rules checked 198 → 151) and #204, where only 5 of the rank 5 rules need checking, as against 25 (total rules checked 225 → 204). Interestingly, as we might expect, rule #52 (which is the most frequently called) gains no benefit, as it is the last member of rank 26 as shown in Figure 3, "Rule order and precedence of matched templates".

We might also speculate on the effect if the 38 rules of rank 26 are re-ordered so that rule #52 is tested *first* and hence the number of rules checked for such nodes drops 52 → 16, suggesting time taken might drop to a third of its previous level. This would of course be predicated on user-provided guarantees of mutually exclusive applicability of rules[1]. In the case of rule #52 simple movement of the template from the beginning to the end of its source file might have similar effect!

In a similar manner for rule #204, it is imported through the `tables.xsl` stylesheet, whose position is circled in blue in Figure 1, "DITA-OT stylesheet importation tree for DITA→FO". If these importations are mutually exclusive, and they may well be, moving the importation of that file to the *end* of the importation list in its parent stylesheet increases the precedence of its templates and hence rank. The orange circled stylesheet contains rule #52. Such rearrangement of source declaration orders may be possible by collecting statistics from representative training runs to detect such conditions.

Within XSLT2.0 it is an implementation choice as to whether conflict raises an error or the last applicable rule in declaration order is chosen. In Saxon, when warnings are silent, the last is always chosen regardless, other rules not being checked. In XSLT3.0 (which this DITA framework predates) this behaviour can be controlled by a `@on-multiple-match="use-last|fail"` property on a suitable `xsl:mode` declaration and issuing of warnings (which imply other rules must be checked) similarly.

6.2. Pretokenizing

The DITA architecture is effectively embedding structure (multiple class membership) within the `@class` attribute. If we can be guaranteed that this is the case, then an option might be to generate preconditions that operate on those implicit tokens whilst tokenising the appropriate accessor once for each node. So for example `*[contains(@class,' topic/entry ')]` would be considered equivalent to the pattern

[1] We would be interested in situations within DITA-OT where there might be some expectation of non-exclusive rules at the same import precedence being written.

`*[tokenize(@class,'\s+') = 'topic/entry')]` [1] which can then be further converted into a pair:

```
$tokens.class := tokenize(@class,'\s+')
        →  ('foo','topic/entry')
test:
   $tokens.class = 'topic/entry'
```

where the node would be tokenized exactly once for each containment-tested attribute (when the condition is first required) and then need only be tested for value membership against the token set in further rules examining the same attribute properties. In most cases the `@class` attribute contains three tags (of which the first is either '+' or '-' which is never tested by templates, at least in the current test, so the string literal sequences to be tested are very short.

Now we define these tests as specialist tokenisation preconditions (they may of course be shared between rules) and index into the collection from the rules. And unlike with the substrings, we *can* project the effect of a true precondition into the pattern viz:

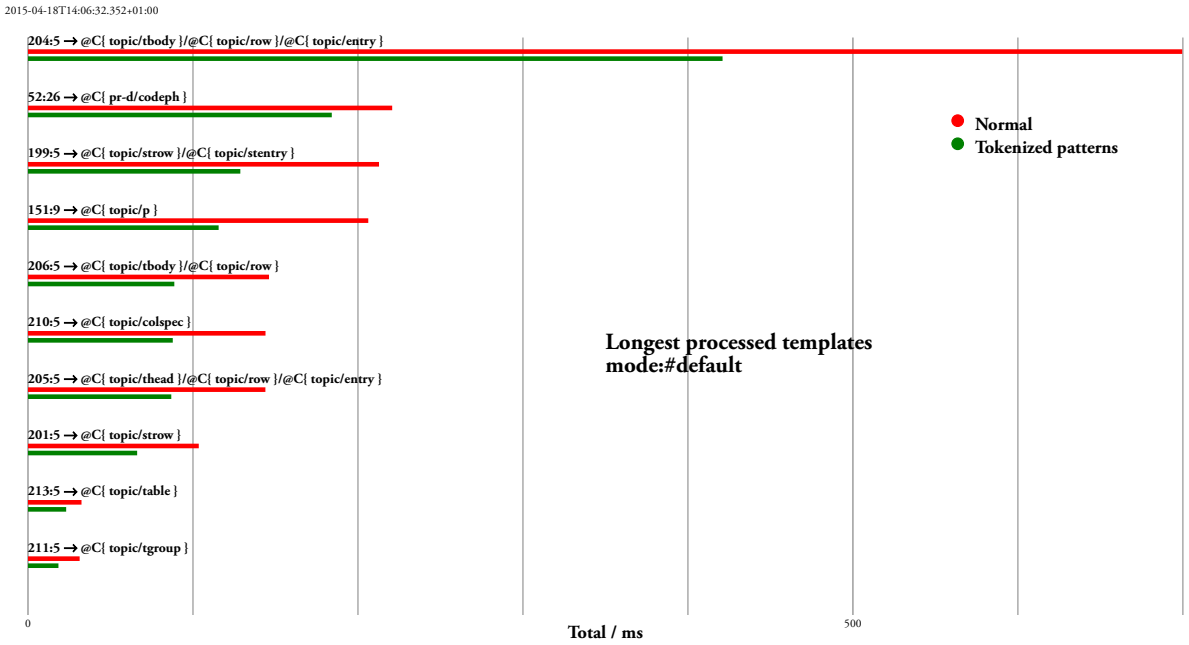
```
R1: *[contains(@class,' topic/entry ')]
R2: *[contains(@class,' topic/row ')]
R3: *[contains(@class,' topic/row ')]/
      *[contains(@class,' topic/entry ')]
→
R1: *[tokenize(@class,'\s+') = 'topic/entry')]
R2: *[tokenize(@class,'\s+') = 'topic/row')]
R3: *[tokenize(@class,'\s+') = 'topic/row')]/
      *[tokenize(@class,'\s+') = 'topic/entry')]
→
$tokens.class :=
    tokenize(@class,'\s+')
$tokens.parent.class :=
    tokenize(parent::*/@class,'\s+')
$preconditionM := $tokens.class = 'topic/entry'
$preconditionN := $tokens.class = 'topic/row'
$preconditionP := $tokens.parent.class = 'topic/row'

R1:  $preconditionM && *
R2:  $preconditionN && *
R3:  $preconditionP && $preconditionM && *
```

where the token variables and precondition references are held within the rule-processing structure. Within Saxon these element match rules (*) would be indexed on the "unnamed element" list, so the last part of each of the final rule patterns would always yield true. In this case these rules have been reduced to just a conjunction of their preconditions.

The preconditions only call for the appropriate tokenisation when it is first needed (they are effectively single-assignment local variables with a scope for the pattern match for a single node, and evaluated lazily) – so that other preconditions involving differing values only need to check within their own sequence comparison. Obviously for a predicate which already uses explicit tokenisation mechanisms (such as processing semicolon-separated `@style` descriptions on SVG and the like) then this technique can be used similarly. For our example document, we get the following performance improvements:

[1] It is only guaranteed equivalent if the `@class` value string starts and finishes with at least a single space.

Figure 9. Effect of pre-tokenizing pattern test inputs



The number of preconditions is now large (~200, corresponding to each possible tag value mentioned in the stylesheet) but most are referenced only once. However they all share a single tokenisation of the `@class` attribute (or that of the parent's in some cases)

Note

A generalisation of this technique into evaluating and then crossreferring to common subexpressions is a possibility. In the case here the binding between preconditions and the evaluated variable is very tight (the variable value for a given node is merely a (small) finite list of strings). Extension to a more generic sequenced-value approach would probably be considerably more complex.

6.3. Using key() mechanisms

From early on XSLT has defined a `key()` mechanism to speed searching for applicable nodes within an XML tree. Using the `xsl:key` declaration a set of nodes can be classified into a number of subsets dependent upon an expression evaluated for each node[1]. Usually support for this within an XSLT implementation is efficient, the key being computed only once. Thus it is tempting to see whether a suitable set of keys can be generated and used within modified patterns. The approach is basically:

```
*[contains(@attr,'string')]
       → *[key('attr','string',.)[1] is .]
```

where effectively the key has been defined by:

```
<xsl:key name="attr"
   select="*"
   use="let $e := . return
    (string1, ... stringN)[contains($e/@attr,.)]"/>
```

The key has indexed all the nodes whose `@attr` contains any of the substrings mentioned within the templates, based on that substring. The pattern uses this key, subsetting the nodes to just those which are `descendant-or-self::*` of the element being tested – if the first node is the target node then there is a match.

We implemented this scheme, but unsurprisingly rather than improve, matters deteriorate significantly. In computing the key (which Saxon does on the first request via the `key()` call) *every* document element is processed, for *every possible contained substring* mentioned in the template sets. Equally well, the predicated key lookup starting at a given node (`[key('attr','string',.)[1] is .]`) involves searching through all the document-ordered nodes already computed for the key (which in our case of course means pretty much all the elements in the entire document) to find the current focus. A

[1] A node can be a member of several subsets, as the key determination can produce several values (e.g. `xsl:key match="car" use="@year,@colour"`).

moment's thought suggests that will have $O(n^2)$ performance. (If the templates were of the form `*[@attr='string']` a key approach might work – certainly `<xsl:key name="attr" select="*" use="@attr"/>` will be very much cheaper.)

7. Generalisation?

In the introduction we mentioned both DITA and Docbook being significant large document-engineering frameworks. Our experimentation has focussed on DITA given the expensive nature of processing its class representation. We were curious to see if similar issues might appear in processing Docbook documents – we believe this not to be the case. A survey of one of the steps (conversion of Docbook into HTML[1]) which is of similar "size" to those within DITA-OT, shows that of the 1500 pattern matching templates within 59 files, only 113 are against unnamed elements or attributes, and none of the 190 modes has more than two. The vast majority of patterns are described for named elements and thus would be fully indexed within Saxon. Hence we anticipate the methods discussed in this paper would not be necessary for that framework.

We have shown that extracting a set of preconditions, where evaluating one precondition for a particular node can eliminate many match patterns, is an effective strategy for the DITA stylesheets we have been studying. This then raises the next question: can the technique be generalized so that it is suitable for inclusion in a general-purpose XSLT processor, producing performance benefits for a sufficiently large set of stylesheets to justify its existence?

This divides into two sub-questions: firstly, is the general strategy of extracting preconditions general-purpose enough? We think it is. Secondly, what about the specific rules that we have found to work well on the DITA stylesheets? Here, we are not convinced – they are intimately tied up with the way DITA-OT decomposes the class representation tags.

We believe that for the same general approach to work with different kinds of stylesheets, we may need to make the rules for extracting preconditions in some way configurable. So we might consider shipping the product with a set of rules that work well for DITA, and other sets of rules that work well for other XML vocabularies. We could consider defining a vocabulary allowing the rules to be written declaratively (see John Snelson's paper on declarative XQuery rewrite rules [4]) for some possibilities. The designers of an XML vocabulary could then perhaps ship a Saxon optimizer plug-in that applies rules appropriate to the specific vocabulary. Saxon could perhaps select an appropriate plug-in from the repertoire available based on the namespaces in use in the particular stylesheet.

8. Conclusions

In this paper we have examined performance issues in processing a relatively large document with an XSLT transform containing a large number of generic templates whose match computation can be expensive, and where large numbers of pattern matches occur very late in "rank order". We've shown that by choosing suitable shared preconditions for rules, which need only be computed once for a node under test, we can ameliorate the effect of such long rank sequences in pattern sets. Alternatively, by choosing to add some "higher-level" knowledge, declaring that a given set of patterns is in effect implementing a tokenisation, we can also improve pattern matching.

As implementors of a major XSLT processor, our next step is to examine ways that such heuristics can be added and configured in the product. Some of these may be very specific declarations within a configuration. Others might be associated with running training sets, collecting statistics and proposing specific tunings. Watch this space...

Saxonica would like to thank its (anonymous) client who was very willing to let us study the processing of one of his real DITA documents in detail. Hopefully we'll be able to repay him soon with some welcome "tune-up".

References

[1] Michael Kay. *Saxon: Anatomy of an XSLT processor*. 2005. http://www.ibm.com/developerworks/library/x-xslt2/

[2] Michael Kay. *Writing an XSLT Optimizer in XSLT*. Extreme Markup Languages. 2007. http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html

[1] `docbook/xsl/html/docbook_custom.xsl` in the Oxygen 16.1 implementation

[3] Michael Kay and Debbie Lockett. *Benchmarking XSLT Performance*. XML London 2014. June 2014. doi:10.14337/XMLLondon14.Kay01

[4] John Snelson. *Declarative XQuery Rewrites for Profit or Pleasure*. XML Prague 2011. March 2011. 211-225. http://archive.xmlprague.cz/2011/files/xmlprague-2011-proceedings.pdf

[5] *XSL Transformations (XSLT) Version 3.0*. 2014. World Wide Web Consortium (W3C). http://www.w3.org/TR/xslt-30/