Saxon - Bug #4273

Performance problem in 9.9HE related to use of bulk copy

2019-08-07 22:39 - Michael Kay

Status:	Closed	**Start date:**	2019-08-07
Priority:	Normal	**Due date:**	
Assignee:	Michael Kay	**% Done:**	100%
Category:	Performance	**Estimated time:**	0:00 hour
Sprint/Milestone:		**Spent time:**	0:00 hour
Legacy ID:		**Fixed in Maintenance Release:**	9.9.1.5
Applies to branch:	9.9, trunk	**Platforms:**	
Fix Committed on Branch:	9.9, trunk		

Description

A test stylesheet runs in under 4 seconds on Saxon 9.8 HE or Saxon 9.9 EE, but takes over 30 minutes on Saxon 9.9 HE.

See forum post https://saxonica.plan.io/boards/3/topics/7554 for details.

The HE performance is dominated by calls on TinyTree.bulkCopy(); bulk copying is not used at all under EE.

History

#1 - 2019-08-07 23:00 - Michael Kay

The first use of bulkCopy() on the HE path is for the xsl:copy-of instruction in ResponseTemplate.xslt line 2258.

The reason that EE is not using a bulkCopy() operation here is that the pipeline includes an extra step for stripping type annotations during the copy. This step is not needed on HE because we know there can never be any (non-trivial) type annotations. It ought to be possible to avoid this step in EE, because we know that the tree being copied has no non-trivial type annotations.

Let's first focus on why bulkCopy() has such bad performance for this use case, and when we've fixed it, we'll make sure that it is used in EE as well as in HE.

#2 - 2019-08-07 23:31 - Michael Kay

Just for the record, I've confirmed that setting the switch TinyTree.useBulkCopy to FALSE makes the problem go away.

I'm having trouble, however, seeing why the bulk copy is taking so long. I suspect it's only certain cases that are slow and the difficulty is in isolating them.

#3 - 2019-08-07 23:54 - Michael Kay

I've put timing information in for each call of CopyOf.copyOneNode(), and measured it with bulk copy enabled and disabled. In the typical case, bulk copy is taking 500 to 700 nanoseconds compared with 1000 to 1400 nanoseconds without bulk copy.

But (a) there are a few cases where where bulk copy is taking much longer (consistently around 450,000 ns), and (b) there are a lot more calls on copyOneNode() when bulk copy is enabled -- 44000 calls and counting, compared with just 768 calls when disabled.

This is hard to explain. My first idea (to be tested) is that these "rogue" calls on bulk copy are creating incorrect temporary trees with too many nodes, and that this increases the cost of subsequent copy operations.

#4 - 2019-08-08 00:27 - Michael Kay

Further monitoring suggests that the number of nodes being added to the tree is the same whether bulk copy is enabled or not; further, the occasional high timings are correlated with the size of the subtree being bulk copied (which is usually 1 or 2 nodes, but occasionally 134 or 135).

So the cost of individual copy operations doesn't seem to be a problem: the question remains, why are we doing more (many more) of these operations when bulk copy is enabled>

#5 - 2019-08-08 01:29 - Michael Kay

I've now traced all the calls to CopyOf.processLeavingTail() and the sequence of calls is exactly the same whether or not bulkCopy is enabled; the counter of the total number of nodes produced by copyOneNode() operations is also identical.

Specifically, in both cases we have 61264 calls on CopyOf.processLeavingTail(), and 2001831 calls on CopyOf.copyOneNode(). But with bulk copy

enabled, we only have 136532 calls on TinyTree.bulkCopy() (i.e. about 7% of the calls on copyOneNode) so we somehow need to focus the measurement on the calls where a different path is taken.

All but one of the calls on copyOneNode() are copying element nodes, but the number of calls on TinyElementImpl.copy() is only 1518691. The other 483140 calls are directed to TinyTextualElement.copy() (which handles element nodes containing a single text node). Copying of tiny textual elements does not use the bulk copy logic.

We rejected 1220228 candidates for bulk copy because they do not have child nodes. Actually, I added this check during the bug investigation, it is not in the actual product: this is probably why the time with bulk copy is now down to 3m rather than 30m -- which at least has the virtue of making measurements much more viable.

#6 - 2019-08-08 10:34 - Michael Kay

OK, I think I've found where the problem is. It's in the logic (within TinyTree.bulkCopy()):

```
                        if (!foundDefaultNamespace) {
                              // if there is no default namespace in force for the copied subtree,
                              // but there is a default namespace for the target tree, then we need
                              // to insert an xmlns="" undeclaration to ensure that the default namespace
                              // does not leak into the subtree. (Arguably we should do this only if
                              // the subtree contains elements that use the default namespace??)
                              // TODO: search only ancestors of the insertion position
                              boolean targetDeclaresDefaultNamespace = false;
                              for (int n = 0; n < numberOfNamespaces; n++) {
                                  if (namespaceBinding[n].getPrefix().isEmpty()) {
                                      targetDeclaresDefaultNamespace = true;
                                  }
                              }
                              if (targetDeclaresDefaultNamespace) {
                                  inScopeNamespaces.add(NamespaceBinding.DEFAULT_UNDECLARATION);
                              }
                          }
```

The problem is that this source document contains many namespace declarations on elements other than the root, including default namespace declarations, so numberOfNamespaces is unusually high: in fact it is 307. So at the very least we should break out of the loop as soon as we find a default namespace declaration; preferably (as the TODO suggests) we should search only the relevant part of the tree.

I have verified that this code is the problem by removing it: the performance regression then goes away. Unfortunately of course the code is there for a reason and can't simply be eliminated.

#7 - 2019-08-08 11:51 - Michael Kay

It's not as simple as just searching the namespaces more efficiently. We need to create fewer namespace bindings on the tree. If we monitor the code by doing:

```
                              System.err.println("Searching " + numberOfNamespaces + " namespaces");
                              for (int n = 0; n < numberOfNamespaces; n++) {
```

we get a trace like this (small extract):

```
Searching 424838 namespaces
Searching 424845 namespaces
Searching 424855 namespaces
Searching 424867 namespaces
Searching 424875 namespaces
Searching 424905 namespaces
Searching 424913 namespaces
Searching 424953 namespaces
Searching 424959 namespaces
Searching 424971 namespaces
```

So we've clearly got a quadratic problem here: we're adding more and more namespace bindings, which is increasing the cost of the search. The length of the search is a symptom; the root cause is the excessive number of namespaces.

The problem seems to be that when we copy an element E from the source tree to the target tree, we are adding all E's in-scope namespaces to the target tree without checking whether they are duplicates. This was a conscious decision to avoid the cost of checking for duplicates, but in this case it's clearly the wrong decision.

#8 - 2019-08-08 17:32 - Michael Kay

Solving this is complicated by the fact that we need to determine the in-scope namespaces of the "target parent" node to which the copied nodes are being attached. But this "target parent" is a node in a tree that is still under construction. To determine its namespaces we need to navigate upwards to ancestors. But the parent pointers aren't yet in place in the partially-constructed tree.

I think this can be solved by (effectively) moving the bulkCopy code from the TinyTree to the TinyBuilder. The TinyBuilder maintains scaffolding during

the course of tree construction that we can use for this situation.

#9 - 2019-08-08 18:32 - Michael Kay

I've now fixed this, the test case is working, and the execution time is 4.07 seconds with bulkCopy enabled, 4.26 seconds with bulkCopy disabled.

Unfortunately a handful of regression tests are failing so I will need to do further work before closing this.

I took a look at why bulkCopy isn't used when validation="strip" is in use (which is in effect the default for Saxon-EE). There's a comment in the code that explains this is because validation="strip" needs to preserve the ID and IDREF properties of nodes, and the bulkCopy() code currently isn't doing this. We should fix this.

#10 - 2019-08-12 11:52 - Michael Kay

Test copy-1221 is failing under 9.9 whether or not bulk copy is enabled. This is a fairly recent test for an edge case involving copy-namespaces="no" and I think it is unrelated to this bug. The test is working on the development branch but I think the changes to get it working under 9.9 are too disruptive and it's not a serious enough problem to justify the destabilisation.

When using the HE test driver, we also get failures in copy-1220 and copy-3803.

copy-1220 fails only if bulk copy is enabled, copy-3803 fails either way. Both tests work under EE. So the priority seems to be to examine copy-1220 and see why bulk copy is producing incorrect results.

#11 - 2019-08-12 12:28 - Michael Kay

I have fixed copy-1220. The code for copying namespaces from the source node to the target node wasn't correctly searching namespaces declared on ancestors of the source node.

copy-3803 turns out to be a problem in the test itself. It requires support for higher order functions but is not marked with that dependency. In fact the dependency was added only to improve diagnostics when the test fails, so it's better to remove the dependency.

There is one remaining test, function-1032, that fails under HE with bulk copy enabled, but not with bulk copy disabled. The primary reason this test is failing is that Saxon-HE is ignoring the attribute xsl:function/@new-each-time="no" (moreover, it does so with a messy error message that mentions saxon:memo-function, which the test does not actually use). I'm not sure why the failure only occurs with bulk copy enabled, but I think that's incidental. I will raise a separate issue on this.

#12 - 2019-08-18 13:56 - Michael Kay

I took another look at the use of bulk copy in EE, when a SkipValidator is generally present in the pipeline. I came to the conclusion that the restriction here is unnecessary, and removing it doesn't appear to cause any tests to fail, so I'm taking it out. With this, I'm happy that the changes for 9.9 are fine, and now I just need to take a look that things are working OK on the development branch (where the bulk copy code is somewhat different because of changes to representation of namespaces on the TinyTree).

#13 - 2019-09-04 13:54 - Michael Kay

The original test case (in bugs/2019/4273-staal-olsen) is not yet fixed on the development branch. It is running in 4.3 seconds on 9.9, but is not finished after a couple of minutes on 10.0 - with heavy CPU usage.

The performance is bad whether or not bulk copy is enabled; and in fact it isn't using bulk copy even when it is enabled. The reason for this is that there is a SkipValidator in the pipeline. Unlike 9.9, the SkipValidator appears in the pipeline AFTER the ComplexContentOutputter.

#14 - 2019-09-04 13:56 - Michael Kay

- Fix Committed on Branch 9.9 added

#15 - 2019-09-04 17:28 - Michael Kay

I've now made changes so bulk copy is used despite the presence of the SkipValidator, and this gives an execution time of 42s - still far too long,

Surprisingly, Java profiling suggests that this has nothing to do with copying or namespaces. Instead, the dominant samples are in evaluating local-name() and AtomizingIterator.next().

The -TP:profile.html output is strange - it contains two reports, as if the transformation was done twice. This is because TimingTraceListener.close() is called twice - once from the finally{} block in XsltController, and once from PushToReceiver$DocImpl.sendEndEvent(). I think it makes sense here for the close() to be indempotent.

Apart from that, the figures in the profile for 10.0 are largely consistent with the 9.9 figures, with one glaring exception: the single execution of the "Initial" template takes 42580ms in place of 3096ms.

There's no obvious difference in the -explain output; the code in both cases is remarkably similar. Note that it is all in 1.0 compatibility mode.

If I change the version attribute to 2.0 I get static errors of the form:

```
Error in {func:ConvertDate($value)} at char 0 in xsl:value-of/@select on line 1413 column 62 of ResponseTempla
tes.xslt:
```

```
XPST0017 Cannot find a 1-argument function named
Q{xalan://diadem.dirigent.plugin.helpers.XsltFunctions}ConvertDate()
```

I guess these are extension function calls in code that is never executed. I added dummy implementations of these functions and ran with version="2.0", execution time unchanged at ~42s.

The Java profile shows many hits in ContentValidator.startElement(). But I can't see why that's expensive. (On the other hand, I can't see why the code is needed in the case of skip validation...). Cutting it out only gives a marginal saving, to 39s. When we cut it out, the samples revert to SkipValidator.startElement(). So perhaps the cost is further down the pipeline...

The next thing in the pipeline is a TinyBuilder, and each call on startElement() seems to supply a new NamespaceMap with identical content.

Looking more closely, I think this has nothing to do with tree copying: rather the problem is that the new TinyTree namespace design for 10.0 does not copy well with this kind of input XML, where we see thousands of sibling elements redeclaring the same namespaces:

```
<CadastralDistrictCode xmlns="urn:oio:ebst:diadem:property" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns
:xsi="http://www.w3.org/2001/XMLSchema-instance">610452</CadastralDistrictCode>
                           <LandParcelIdentifier xmlns="urn:oio:ebst:diadem:property" xmlns:xsd="http://www.w
3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">18ao</LandParcelIdentifier>
                           <CadastralDistrictName xmlns="urn:oio:ebst:diadem:property" xmlns:xsd="http://www.
w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">Gl. Hasseris By, Hasseris</Cadast
ralDistrictName>
                           <LandParcelRegistrationAreaMeasure xmlns="urn:oio:ebst:diadem:property" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">35</LandParcelRegistr
ationAreaMeasure>
                           <AccessAddresses xmlns="urn:oio:ebst:diadem:property" xmlns:xsd="http://www.w3.org
/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

At present we reuse a NamespaceMap when a child element has the same namespaces as its parent, but we do not reuse them across siblings. As a result the source tree has thousands of NamespaceMap objects, and a consequence of this is that we can't subsequently recognise quickly that two elements have the same namespace context. It looks as if, for this kind of use case, the TinyBuilder may need to cache namespace maps.

#16 - 2019-09-04 23:32 - Michael Kay

Caching the pool of NamespaceMap objects in ReceivingContentHandler greatly reduces the number of NamespaceMaps on the source tree, from around 8000 to just 18, but it has little impact on the transformation time, which still stands at 39s. And we are still seeing on the Java profile, a heavy hotspot at SkipValidator.startElement().

If I change SkipValidator.startElement() to essentially do nothing, the hotspot is revealed a little more clearly:

```
        net.sf.saxon.tree.tiny.TinyTree.addNamespaces(TinyTree.java:925)
 net.sf.saxon.tree.tiny.TinyBuilder.startElement(TinyBuilder.java:321)
 com.saxonica.ee.validate.SkipValidator.startElement(SkipValidator.java:100)
```

Changing TinyTree.addNamespaces() to compare NamespaceMaps using "equals()" rather than "==" turns out to do the trick: elapsed time down to 2 seconds (or 1.37s with -repeat:10). The cache in ReceivingContentHandler is then no longer needed.

With bulk copy disabled, we also get decent performance, this time about 1.5s.

#17 - 2019-09-04 23:32 - Michael Kay

- Status changed from New to Resolved

- Fix Committed on Branch trunk added

Marking as resolved.

#18 - 2019-09-05 16:43 - O'Neil Delpratt

- Status changed from Resolved to Closed

- % Done changed from 0 to 100

- Fixed in Maintenance Release 9.9.1.5 added

Bug fix applied in the Saxon 9.9.1.5 maintenance release.