Benchmarking XSLT Performance

Michael Kay

Saxonica

Debbie Lockett

Saxonica

Abstract

This paper presents a new benchmarking framework for XSLT. The project, called XT-Speedo[1], is open source and we hope that it will attract a community of developers. The tangible deliverable consists of a set of test material, a set of test drivers for various XSLT processors, and tools for analyzing the test results. Underpinning these deliverables is a methodology and set of measurement objectives that influence the design and selection of material for the test suite, which are also described in this paper.

1. Objectives and Motivation

Performance of XSLT is, of course, important, though we need to qualify that by pointing out that performance is not the only thing that matters.

For Saxonica as a developer of one of the leading XSLT engines, performance is not actually our number one objective. Our first objective is standards conformance; the second is usability, and performance comes third. This means that we will (almost!) never sacrifice standards conformance in order to achieve improved performance, and we are prepared to take a performance hit in order to improve usability, for example by maintaining extra run-time information that is needed only for diagnostics when things go wrong.

We choose these priorities because we think this is what the market wants. We are probably rather obsessive about corner cases when it comes to standards conformance, but being obsessive about cases that most users don't care about means that we almost always get things right in the more common cases where users care a lot. Standards conformance is typically a major objective for vendors who can't take their place in the market for granted, and for Saxonica, achieving standards conformance was what gave us a place at the top table of XSLT vendors alongside the likes of Microsoft and IBM.

As for usability, we believe that far more users notice sub-standard usability than notice sub-standard performance. Most users only care that performance is good enough, not that it is the best available. If one processor runs a transformation in 0.1s and another does the same work in 0.2s, most users won't notice the difference. They are much more likely to end up using your product because they discover, from experience, that its error messages are more helpful.

But even though performance is our third priority, it's still extremely important. Saxonica has users running some seriously impressive workloads, and when we get things wrong, they notice.

In measuring performance, our main objective is to avoid regression between successive releases. Such regression is probably the most obvious source of complaints; the last thing users want if they move forward to a new release is to find that their particular workload runs more slowly. Although we have always included some performance tests in our standard quality assurance checklist before release, experience has shown that these are inadequate, and too many mistakes slip through.

Another significant objective is to be able to test the impact of product changes. When we introduce a change (perhaps a new optimization) that is designed to improve performance,we will often do some ad-hoc tests to ensure that the particular test case it is tackling shows the expected improvement. But assessing the overall impact of the change is much more difficult.

[1] *XT-Speedo - https://github.com/Saxonica/XT-Speedo/*

doi:10.14337/XMLLondon14.Kay01

Some readers may be surprised that we do very little competitive benchmarking of our own product against products from other vendors. We did more of this in the early days, before Saxon became well established. One explanation is that the primary reason people adopt Saxon has always been that they want access to XSLT 2.0, and are switching from a product that only supports XSLT 1.0. In that situation, the only thing they want to be sure of is that the change will not cause an unacceptable performance hit. We've had lots of positive feedback from users making this transition, and very little negative feedback, so we haven't felt any pressure to improve our competitive position, although we have always been aware that some of the competitors' products have excellent performance.

As regard competitive performance, it's worth emphasizing that this is not a race in which the winner takes all. Firstly, speed is not a one-dimensional metric, so there will never be a single winner regardless what you choose to measure. Secondly, being within 5% of the leader is good enough for all practical purposes, since if two products differ only by 5% in performance, then users will choose between the products based on other factors. If a user has a performance problem, it's most unlikely that a 5% improvement will solve it.

2. Previous work

There have been previous benchmarking environments for XSLT.

An early attempt was the XSLTMark benchmark from Datapower, which later became part of IBM. Datapower offered this as a free download from their web site for a number of years, but it disappeared at around the time of the IBM acquisition. The licensing terms were unspecified, so although we (and no doubt others) have continued to use this benchmark in-house, we have no authority to make it public. XSLTMark suffered a common problem when comparing results across different products: different drivers measured different things. For some products, the cost of compiling the stylesheet was included in the execution cost, for others it was discounted. This made product comparisons fairly useless, but it was still a useful tool for comparing successive releases of the same product.[1][2]

XSLT processors have come on a long way since 2001, but it's very hard to find any more recent data that compares their performance.

Another benchmarking effort that appeared on the web and then disappeared leaving very little trace was Bumblebee. The main problem with this effort was that the drivers were not open source, and there was insufficient information provided to create your own drivers. We used it in Saxonica for a while, but when we found that we couldn't update or tweak the drivers to experiment with different settings, we abandoned it.

Sarvega published results of a benchmarking study in 2003, but the results were only available on a commercial basis, and the tools needed to reproduce the results were not made available at all.

A more recent effort is XSLTMark II[3], produced by Viktor Mašíček as an M.Sc thesis at Charles University, Prague. Mašíček is concerned to measure more than just performance, but although he recognizes the importance of other qualities such as usability, he does not attempt any scientific measurement. He also on occasions fails to distinguish correctness from usability, for example when he commends XSLT 1.0 processors that reject XSLT 2.0 constructs rather than ignoring them, which might well be the right thing to do from a usability perspective, but happens to be non-conformant with the XSLT 1.0 specification.

On performance, Mašíček's reported results suffer from a failure to distinguish the different factors that contribute to execution time. In the case of Java processors, he makes insufficient effort to discount the effects of Java VM warm-up time; for example, he reports Saxon-HE 9.4 as running three times slower than Saxon 6.5, an effect which can only be explained by the fact that Saxon-HE is larger and therefore takes longer to load. For some workloads this start-up cost matters to users, but for production web sites running thousands of transformations per hour, as well as for one-off transformations of very large documents, start-up cost is irrelevant. The same arguments apply to the cost of stylesheet compilation; Mašíček makes no attempt to distinguish compilation time from execution time, but for many workloads, compilation time can be ignored.

Another problem with Mašíček's benchmark is that he runs every processor in the same environment (PHP). For processors that are not designed to run in this environment, this creates an artificial overhead. For example, to invoke Java processors he uses what is essentially a command-line invocation. The overhead of invoking a Java processor in this way swamps the actual transformation cost, so the measurements are entirely untypical of what can be achieved in a native Java environment.

[1] A description of the XSLTMark benchmark, including an acknowledgement of the problems in using it for cross-product comparisons http://www.xml.com/pub/a/2001/03/28/xsltmark/index.html

[2] A set of XSLTMark results from 2001 http://www.xml.com/pub/a/2001/03/28/xsltmark/results.html

[3] XSLTMark II http://xsltbenchmarking.masicek.net

Nevertheless Mašíček's work is valuable, in particular the collection of stylesheets that he has collected, and our work builds on this.

Other relevant work includes benchmarks for XQuery and XPath. For XQuery the XMark benchmark[1] is the best known. This includes a test data generator which can be used to create data files of different sizes, which therefore enables measurement of how query performance scales with data size. This is particularly useful to enable comparison of strategies for join optimization in different products. We have used this benchmarking framework extensively in Saxonica, and have adapted some of the 20 queries to XSLT, but they are not very typical of real-world XSLT workloads and this limits their usefulness.

A more recent paper from 2006 surveys XQuery benchmarks[2]. This paper contains much useful discussion of the characteristics of a good benchmark and the kind of information it can yield if analysed intelligently; in particular, the importance of determining the way in which performance varies (for example, with document size or query complexity) rather than collecting simple numbers.

3. The design of the XT-Speedo benchmark

The XT-Speedo benchmark has been designed to measure the performance of different products for a broad range of test transformations. In particular we have chosen to measure the time for three processes (where possible): compiling the stylesheet, file to file transform, and tree to tree transform. Since different vendors may have varying priorities, and since it is interesting in itself to see the difference in performance for these different parts of the transform process, we considered taking these three measurements to be useful.

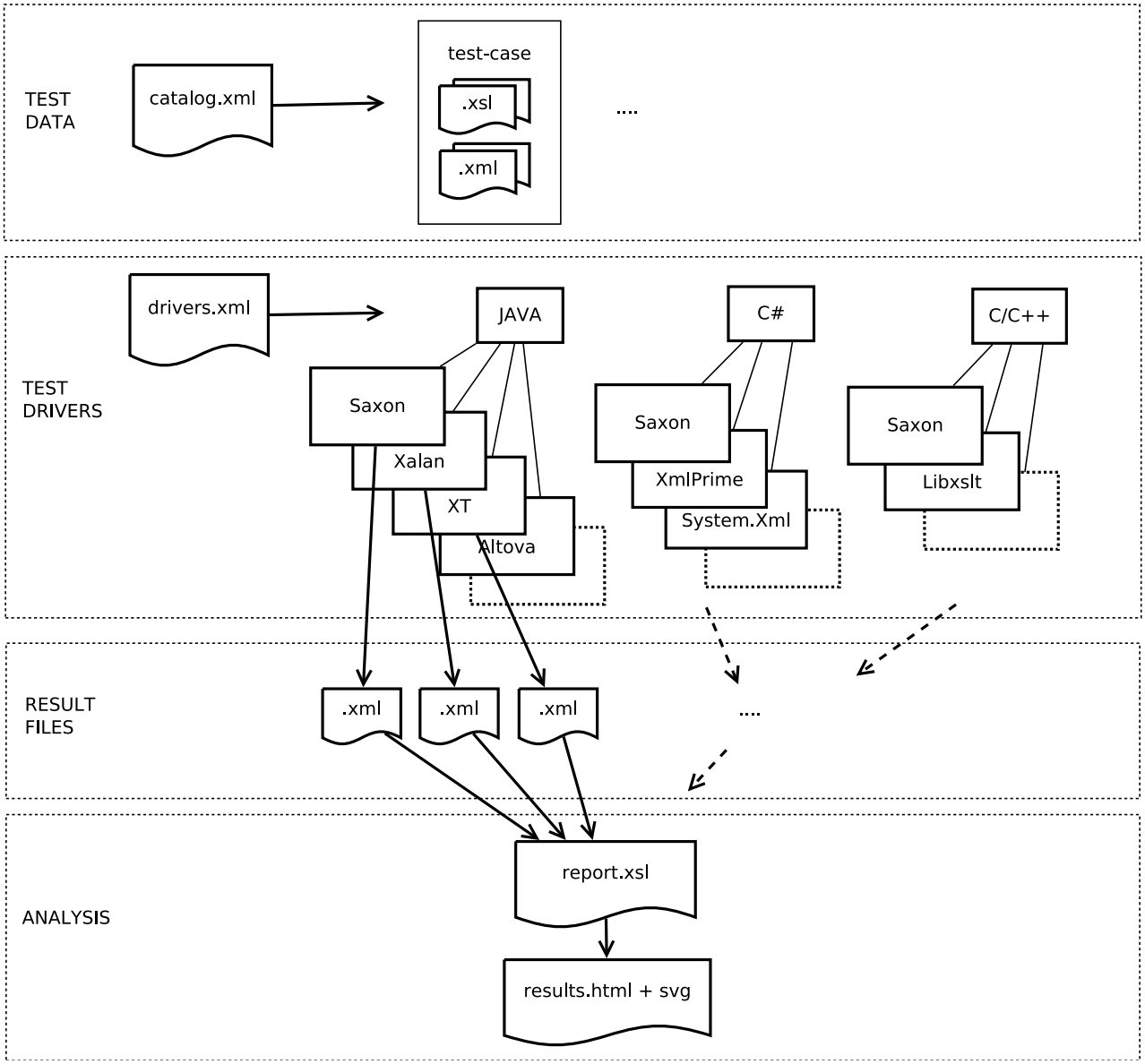
We have produced XT-Speedo benchmark packages on the Java and .NET platforms, and in C/C++, to run different test drivers for the different products which are available in different environments. For each environment, the packages each contain a class to run the benchmark tests (called Speedo on Java, and RunSpeedo on .NET), and an abstract class called IDriver, which is subclassed for each product-under-test. The IDriver interface defines methods to compile a stylesheet, to load a schema, to build a source document, and to run tree-to-tree or file-to-file transformations.

Not every product supports all these options. Some, obviously, are not schema-aware. Some, such as Altova's RaptorXML, do not provide an explicit interface to compile the stylesheet (perhaps they rely on caching instead). Some, such as James Clark's XT, do not separate tree construction from transformation. A further complication is that we want to compare the performance of all processors when running XSLT 1.0 tests, but we are also interested in XSLT 2.0 and XSLT 3.0 performance, so we have to accommodate the fact that for different processors, we may have different subsets of the measurements, available over different subsets of the tests. We therefore use XML-based catalog files to control which tests are run, using which processors.

The design of the benchmark is illustrated by the following schematic:

[1] XMark benchmark http://www.xml-benchmark.org/index.html
[2] XQuery benchmarks survey http://leo.saclay.inria.fr/events/EXPDB2006/PAPERS/Afanasiev.pdf

Figure 1. Architecture diagram



The main command line input for the Speedo benchmark (i.e. input for the 'run' method of the Speedo class) is the tests catalog file and the drivers catalog file. An option is also provided to supply the location of the output directory for result files. Further options are provided to select which test cases from the catalog to use — primarily by providing a regular expression name pattern, but also for example by choosing to skip tests which are slow.

The tests catalog ("catalog.xml") identifies a collection of test cases, gleaned from various sources, plus some newly created ones. Each test case is a particular transformation for the product to process. The catalog contains a description of the test transformation, links to the source XML file and XSL stylesheet (and in some cases, an XSD schema), and an XPath assertion to be applied to the output of the transformation to check the results are plausible. (It's not a primary aim of this exercise to check the conformance of processors to the specification; but we want to weed out results that are wildly out, especially cases where the processor fails to perform the transformation at all.)

The drivers catalog ("drivers.xml") contains the driver data, including: name, implementation class, implementation language, XSLT version. This is where initialization option settings and test run options can be added. In the case of Saxon, we might have several entries in the drivers catalog that run Saxon with different configuration options (optimization on or off, bytecode generation on or off, and so on), allowing us to assess the impact of these configuration switches. The drivers catalog can also indicate that particular tests should not be run with a particular driver (because they are known to crash, for example, or because they are excessively slow.)

Because different processors run in different environments, collecting a full set of data for all processors requires more than one program run. As a minimum, there will be three runs, one for Java processors, one for .NET, and one for C/C++. But if we want to measure different versions of Saxon, or performance on different hardware or operating systems,then additional runs will be needed. (We have experimented with a mechanism that calibrates the hardware speed and adjusts performance measurements to compensate for differences. However, this mechanism is not fully operational.)

Each execution of the benchmark runs all the selected tests from the test catalog (as selected by the specified configuration), and produces one XML results document per driver. For each test case, measurements are taken for the times (in milliseconds) to perform three processes: compiling the stylesheet, file to file transform, and tree to tree transform. Each test case is run multiple times (by default set at 20) and the average times for these processes are taken, in order to eliminate warm-up time and aberrations caused by activities such as garbage collection. (For Java in particular, hotspot compilation causes dramatic improvements in execution time the more often the code is run, with performance often stabilizing only after a minute or so.) These average process times are recorded in the result file, indexed by the test cases, where the level of success of the test run is also recorded - 'success' if the run is fine, 'wrong answer' if transforms take place but the assertion test fails (so the transformation result is not as expected), and 'failure' if the transformation fails at some point.

A selected subset of these result files can then be collated using the "report.xsl" XSLT stylesheet. Using the results XML files as input, this stylesheet produces HTML documents (including SVG graphics) to view the results. The results tables give times relative to a selected baseline driver (chosen in the drivers catalog). The main overview page contains, for each driver, a measure of the overall performance relative to the baseline for each of our three processes: file to file transform, tree to tree transform, and stylesheet compile. The formatted report for each driver contains tables with rows for each test case, giving the relative times, and actual times, for the three processes.

As our "bottom-line" metric we use the sum of the process times (over all tests) for the driver, relative to the sum for the baseline driver. This of course gives greater weighting to tests which take longer; a different choice of computation could well give a different picture. We also give the minimum and maximum values of the relative times for the individual tests, to give an idea of the spread. We fully recognize that this is highly arbitrary; there are other ways of doing the aggregation that would give different results. XSLTMark, for example, divides execution time by source document size to give a transformation speed measured in bytes per second, and averages across these speeds. In some cases, as we will see, relative performance of different processors varies substantially from one test to another, and because our test collection makes no serious attempt to be representative of any real workload, an average across all the tests can fairly be dismissed as meaningless. In defence, users of the benchmark are free to substitute a different set of tests that reflects their own choice of workload more accurately.

4. Test Data

The selection of data files and stylesheets used in the benchmark should not be regarded as being fixed in concrete. XT-Speedo is intended as a framework for measuring XSLT performance, not primarily as a set of test programs which claim any kind of canonical status. The variability of results across the different test data sets often provides more information than any aggregate numbers. The data included in the benchmark is a motley collection, including some things that just happened to be available (for example, files from the original Datapower XSLTMark benchmark), some that we wrote specially because we wanted to investigate a particular area of performance, some that we have used in the past to study particular performance issues reported by Saxonica customers, and also a translation of the XMark XQuery benchmark, allowing us to see how XSLT and XQuery performance compare. The XMark data is particularly useful because it allows one to study how performance varies as a function of the size of the source data set.

Any attempts to aggregate results over all the tests are inevitably flawed. While a figure that averages performance across a range of different tasks is likely to be more reliable than a figure for one task alone, it would be quite wrong to assume that the tests in this benchmark collection are representative of any real production workload. Although they are nearly all real programs designed to perform (or at least emulate) a useful task rather than to stress obscure corners of the processor, some of them perform rather untypical tasks, such as parsing XPath expressions.

It is therefore quite legitimate, and positively encouraged, to run the XT-Speedo benchmark with different data files that better characterize the workload for which performance data is required. Unlike some classic industry benchmarks such as TPC, we have no aspiration to define a performance metric that vendors can publish on billboards to proclaim that their product is 32.87% faster than the competition. Rather, the benchmark is a resource that anyone can use to compare different workloads in different environments in any way that suits their purposes.

5. The Problem of Bias

We are acutely aware that our results are not impartial. We know that our own motivations are divided between wanting to know the truth about how our product rates against the competition, and wanting our own product to perform well.

The problem of bias arises from several sources: requirements, expertise and motivation.

- *Requirements:* in designing the benchmark, we are choosing what to measure, and the metrics we choose reflect our assumptions about what we think is important. For example, we consider compile-time performance much less important than run-time performance. But others might have different priorities. Similarly, all our measurements focus on latency rather than throughput (the time to execute transformations in a single thread). We quickly found that one particular processor, Altova RaptorXML, fares very badly on this metric, because it is designed to execute in an HTTP server environment and is clearly optimized for throughput rather than latency. The fact that it scores very badly on our measurements does not mean it would score equally badly if we chose different metrics.
- *Expertise:* we know how to get the best possible performance out of our own processor, but we have far less knowledge of the products of our competitors. We've seen third-party benchmarks that ran Saxon in hideously inefficient ways (for example, taking the input from a DOM tree, which can increase transformation time by a factor of ten), and we know that we are at risk of making equally bad choices when running other products that we are less familiar with.
- *Motivation:* we naturally want to get the best possible results for our own product, so if the results don't look good, we will instinctively try again with different settings. We don't have the same motivation for other products, so we are less likely to make the effort. To take an example of this effect, when we compared Saxon/C against libxml our first attempts showed Saxon/C in a very poor light. We naturally investigated, and found a gross error in the way the measurements were being computed. Can we honestly say that we would have investigated as thoroughly if the results had been the other way around?

The bias is there despite our best intentions. We want good data on our competitors' products; we don't want to deceive ourselves. Our best defence against bias is to make the benchmark open source. Our hope is that we will get contributions from others whose bias is different, in particular, who will apply the same diligence to other products as we apply to our own. Meanwhile, however, the biased results are the only ones available.

Because we know there is bias, we refrain from publishing detailed data for our competitors' products in this paper. The results are available on the web site, where they can be corrected if they turn out to be wrong. They will also be presented in the conference, but as a snapshot of current results, not as part of the permanent record.

The problem of bias arises far less when we are comparing our own product, Saxon, running in different versions and configurations. As explained earlier, this is in fact our primary motivation for producing the benchmark. So in the next section these results can be seen as more impartial than the competitive rankings.

6. Selected Results

In this section we will present some of the results we have obtained by running the benchmark, and our analysis of these results. We focus on five particular comparisons: an overall comparison of all processors on the Java platform (all of which, with the exception of Saxon, are XSLT 1.0 processors); a comparison of Saxon on the Java and .NET platforms; a comparison of Saxon with XMLPrime, this being the only other XSLT 2.0 processor we were able to study; a comparison of Saxon 9.5 with a current development snapshot of the forthcoming Saxon 9.6 release.; and a comparison of the new Saxon/C processor with libxslt.

6.1. Ranking of Java Processors

A number of XSLT processors have been developed for the Java platform: as well as Saxon, there are several versions of Xalan, including the XSLTC processor which was developed separately but is now bundled with the Xalan distribution; there is James Clark's original XT processor; there is the no-longer-available jd.xslt, and there is IBM's commercial Websphere processor. Of these, Saxon and Websphere are the only two processors that support XSLT 2.0, and the only two that are still actively developed. For commercial reasons we have not been able to include Websphere in our study. A comparative study of Java processors is therefore confined to XSLT 1.0, and the interesting question for us is how Saxon stacks up against products that have been around and stable for many years.

Here are the XT-Speedo results we are currently getting, using Saxon EE 9.5 as the baseline (recall that we do not get tree to tree transform times for XT).

Table 1. Results overview table for Java drivers

	Times relative to SaxonEE-9.5-J driver (smaller values represent faster times)		
Driver	File to file transform	Tree to tree transform	Stylesheet compile
Saxon-6	1.108 min = 0.271, max = 4.33	3.915 min = 0.178, max = 429.71	0.204 min = 0.081, max = 0.779
SaxonHE-9.5-J	1.076 min = 0.512, max = 3.329	1.279 min = 0.325, max = 98.844	0.212 min = 0.083, max = 1.802
Xalan	2.452 min = 0.467, max = 13.379	8.911 min = 0.37, max = 568.723	0.283 min = 0.1, max = 1.284
XSLTC	0.989 min = 0.273, max = 3.407	3.142 min = 0.182, max = 257.989	0.544 min = 0.203, max = 3.195
XT	1.398 min = 0.336, max = 7.717	NaN min = NaN, max = NaN	0.23 min = 0.088, max = 0.886

The tree-to-tree transformation times shown here illustrate the difficulty of getting good measurements on the Java platform. For all processors, the "max" figure indicates the presence of outliers in the results that create a completely distorted bottom line. If these rogue results are excluded, the figures end up being much closer to the file-to-file timings. So we'll concentrate on the file-to-file numbers as they appear to show a more regular picture.

These figures show Saxon-EE performing 7% faster than Saxon-HE on average, which is not as large a margin as we would like given the investment we have made in features such as optimization and byte-code generation, but perhaps reflects that these advanced techniques make little impression on straightforward transformations which dominate the test suite. The 10% edge over the old Saxon 6 processor (which implemented XSLT 1.0 only) is also a satisfactory outcome. In fact these figures mask the fact that there are a few transform times where Saxon-EE dramatically outperforms the other processors because of the way in which it optimizes joins: for the test xmark-q8-4 Saxon-EE is almost 100 times faster than Saxon-HE (most processors have quadratic performance on this test, which Saxon-EE optimization reduces to near-linear).

James Clark's original XT processor is now of largely historic interest, but in the early years it was noted for its lightning-fast speed, so it is good to note that we are now 40% faster.

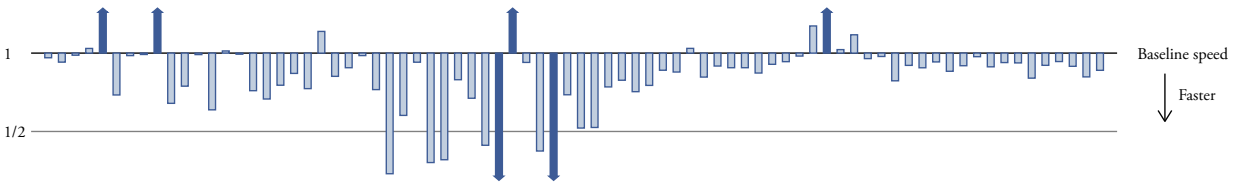
Saxon's very significant advantage over the interpretive version of Xalan should not surprise anyone who has compared the two. The fact that Xalan, being the default XSLT processor in Java, is both the most widely-used and the slowest of these products by a significant margin, tends to reinforce the message at the beginning of this paper that coming first in the performance race brings no guarantee of market leadership.

The only product ahead of Saxon-EE is the XSLTC processor (which is bundled with Xalan). This processor makes heavy use of bytecode generation and the results appear to demonstrate that there are still advances to be made in this area. (Saxon-EE also uses bytecode generation, but primarily for the XPath part of the processing. Most of our measurements of the effect of bytecode generation have been with XQuery, where we generally record a boost of around 25%, but with wide variations. It is not surprising that the speed-up we get for XSLT should be lower.)

Let's take a closer look at the comparison of Saxon-EE with XSLTC results (see charts below). Here we see in more detail that for file to file transform, in the majority of tests XSLTC is just a few percentage points better than Saxon-EE, with XSLTC generally slightly faster (with just a few exceptions). For many of these tests, especially the XMark queries which dominate the right-hand half of the chart, the actual performance for file-to-file transformation is dominated by parsing and serialization costs, and it appears to be in these areas that XSLTC has the edge.

Tests whose results are outside the 95th percentile range are shown with an arrow to indicate they are off the scale. The actual numbers are available in the detailed results listings. In this particular example, the outliers are not extreme; for all tests the ratio between XSLTC speed and Saxon-EE speed is somewhere between 0.25 and 4.

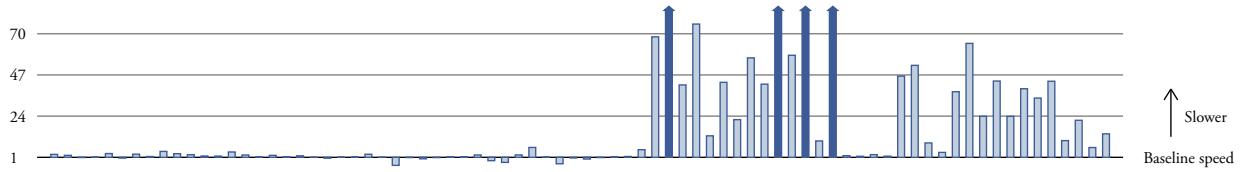
Figure 2. XSLTC file to file transform speeds relative to SaxonEE-9.5-J



For tree-to-tree transformation we see a very different picture (below). For these transformations the times for XSLTC are generally close to Saxon or a little slower on the left-hand part of the chart where source documents are mainly rather small, but on the right-hand side, where most of the source documents are 1-4Mb in size, XSLTC is significantly slower than Saxon-EE. What isn't immediately obvious from these charts is that for Saxon, the tree-to-tree time for the larger source documents is a tiny fraction of the file-to-file time (in one typical example, 0.24ms rather than 20.4ms), but in the XSLTC case the timings for the two scenarios are much closer

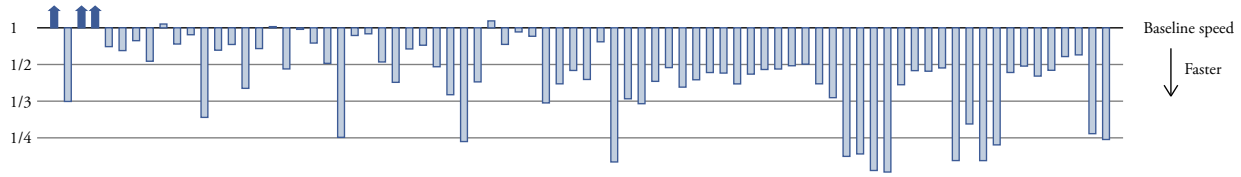
(10.1ms compared with 16.2ms). We suspect that we are running XSLTC sub-optimally here by providing a DOM as input. Perhaps it is not using the source tree that we supply directly, but rebuilding it internally into its own format.

Figure 3. XSLTC tree to tree transform speeds relative to SaxonEE-9.5-J



Our final metric is for stylesheet compile time. Here the figures are fairly uniform across all tests, with XSLTC compiling in around half the time of Saxon-EE on average:

Figure 4. XSLTC stylesheet compile speeds relative to SaxonEE-9.5-J



Both Saxon-EE and XSLTC compile to bytecode, so it is not surprising that both are significantly slower at compile time than the products that are pure interpreters. It is noteworthy that Saxon-EE takes significantly longer to compile the stylesheet than all the other processors. We could argue that this is by design; we deliberately do as much work as possible at compile time in order to improve run-time execution speed. On the other hand, there are workloads (the DocBook rendering of this conference paper is an example) where compiling the stylesheet takes longer than the actual transformation, and there is definitely an opportunity here for Saxon to do better.

The conclusion we can draw from these results is that while Saxon is not always the fastest, it performs well overall. In particular, for anyone wanting to move forward to XSLT 2.0 for the functionality and productivity benefits it offers, or who is attracted to Saxon because the product is actively developed and supported, performance is not an obstacle. For some workloads, users have seen significant performance benefits by moving to Saxon, but as the numbers show, this cannot be expected to apply in every case.

The other apparent result, subject to confirmation, is that the area where Saxon-EE has most improvement potential is in parsing and serialization, not in transformation proper. There are a great many workloads where XML parsing (of the input) and serialization (of the output) dominate the actual transformation time.

6.2. Comparing Saxon on Java with Saxon on .NET

Saxon on .NET starts with a disadvantage: the product is written in Java, and then cross-compiled using the IKVMC compiler to the IL code supported on the .NET platform. This inevitably introduces a performance penalty.

The question for some time has been how large this penalty is, and we have had conflicting reports on this over the years. Sometimes we see an overhead of around 25%, but sometimes the .NET performance is reported to be five times slower.

Here are the XT-Speedo results we are currently getting: File to file transform relative time average 3.829 (min 1.136, max 8.716), tree to tree transform relative time average 3.598 (min 0.239, max 8.938), stylesheet compile relative time average 2.386 (min 0.168, max 3.454).

Figure 5. SaxonHE-9.5-.NET file to file transform speeds relative to SaxonHE-9.5-J

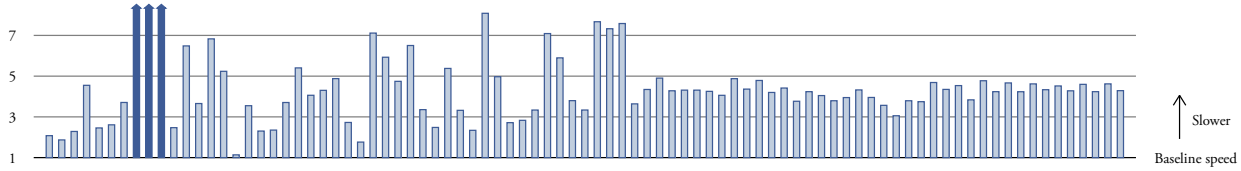


Figure 6. SaxonHE-9.5-.NET tree to tree transform speeds relative to SaxonHE-9.5-J

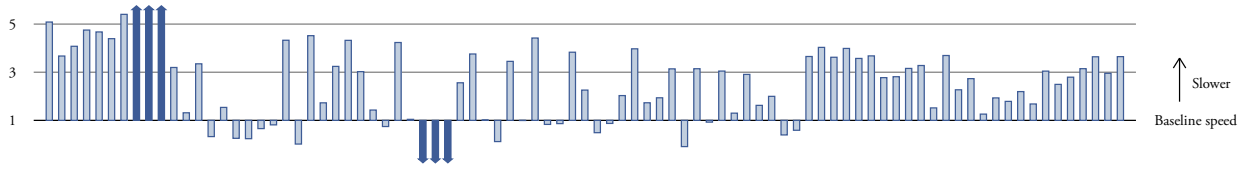
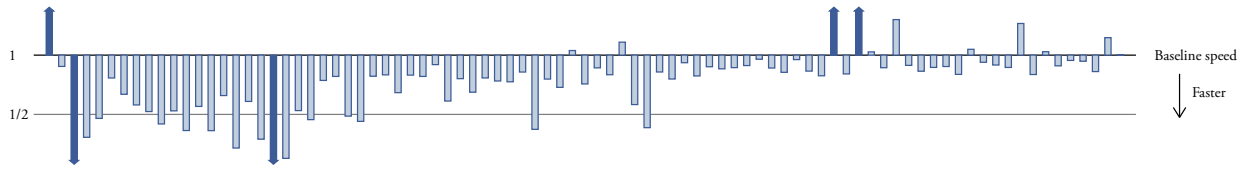


Figure 7. SaxonHE-9.5-.NET stylesheet compile speeds relative to SaxonHE-9.5-J



It is clear that generally Saxon on .NET is indeed running transforms 3 to 4 times slower than on Java, with some variation for different tests. Perhaps surprisingly, Saxon on .NET sometimes performs tree to tree transforms faster than on Java. By looking at the table of results (not shown here) we see that .NET speeds for tree to tree transform are only ever faster for transforms which are very quick - those which take less than 2ms (but .NET is not uniformly faster for these). Generally performance worsens for longer transforms, but we may note that in general the scaled pairs of xmark tests have similar relative times.

The chart for compile times is particularly remarkable, because most of the compilations are actually faster on .NET, but the average is still slower: the explanation for this paradox is that most of the stylesheets are very small, but one test (on the far left of the chart) compiles the DocBook stylesheets, which are much larger than all the others combined. Again the performance ratio seems worse for longer runs. One possibility we need to explore is that we are measuring different things on the two platforms (what are the precise semantics of the instrumentation APIs we are using?), or that the measurements are somehow suffering from rounding errors.

We do not yet fully understand the reasons for the discrepancies in these results. We have established that there are no significant differences in the code path with Saxon, and we know that the overhead imposed by IKVMC is not more than 25% or so. So far, our investigations suggest that the problem lies somewhere in the OpenJDK library. Saxon on .NET uses the OpenJDK java library, cross-compiled to .NET using IKVMC, and the data makes us suspect that there are parts of this library that perform significantly worse than the equivalent library delivered with the Oracle/Sun JDK. We have confirmed this by building Saxon on the Java platform to run with OpenJDK rather than with the Oracle/Sun libraries. Hopefully, armed with these measurements, we can identify a specific cause within the OpenJDK and eliminate it. As always, good measurement data is the prerequisite to solving performance problems, and we now for the first time have that data.

6.3. Comparing Saxon with XMLPrime

For various reasons (which would make an interesting subject for another talk), none of the XSLT 2.0 processors currently on the market are pure open source products. Products from IBM, Intel, and MarkLogic are purely commercial, while those from Saxonica, Altova, and XMLPrime provide limited free versions in one form or another, but offer only commercial licenses for the full product capability. This of course makes product comparisons much more difficult and expensive.

The other XSLT 2.0 processors we have included in our study are Altova's RaptorXML and XMLPrime. In the case of Altova RaptorXML, the product architecture is so different that the figures we obtained were not meaningful to compare; each transformation requires an HTTP request, and our performance data was dominated by the costs of these requests. No doubt much better figures could be obtained for Altova if we did the measurements a different way, but for the present we have discarded the numbers as not useful.

XMLPrime, on the other hand, has a very similar architecture to Saxon; indeed, a cursory glance at its structure shows that it was strongly influenced by Saxon's design. So measurements here should be useful.

The most interesting result here is to show relative speeds for each test case. We see that the pictures for file to file and tree to tree are closely related. In general, XmlPrime is running just a little slower than Saxon EE, but it is sometimes faster. There are just a few cases where XmlPrime is much (more than 5 times) slower than Saxon, and here we see the difference for both file to file and tree to tree transform times. These cases are all among the tests which take longest, and so are meaningful. In contrast, the cases for which XmlPrime is much faster than Saxon are all very quick tests, so we may consider these numbers to be less reliable and meaningful - the fact that these cases are different for file to file and tree to tree transforms, where we have already said that the results correlate strongly, backs this up.

Because Saxon is faster on some tests, while XMLPrime is faster on others, any "bottom line" comparison of the two products is highly sensitive to the choice of test material, and to the way in which the results for different tests are aggregated. The formula we use for aggregation shows file to file transform relative time average 4.581 (min 0.266, max 189.62), tree to tree transform relative time average 9.753 (min 0.29, max 469.095), stylesheet compile relative time average 1.18 (min 0.218, max 21.681). But from the charts, we see that these figures would change greatly if a few anomalous cases were removed. If outliers are discarded, XmlPrime is on average perhaps about 2 times slower that Saxon on transformation time.

We see more variation in the stylesheet compile times, and here more generally XmlPrime performs a little faster. There are two anomalous cases, for which XmlPrime is more than 20 times slower - these are cases that also showed far worse transform times.

Figure 8. XmlPrime file to file transform speeds relative to SaxonEE-9.5-J

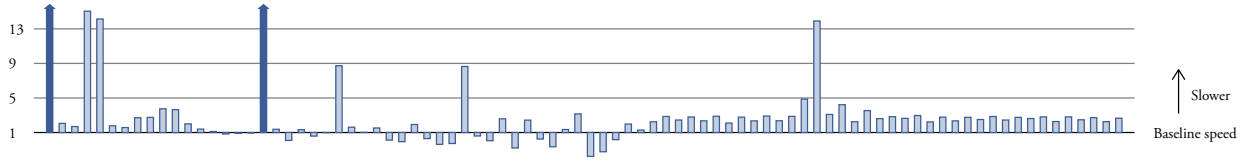


Figure 9. XmlPrime tree to tree transform speeds relative to SaxonEE-9.5-J

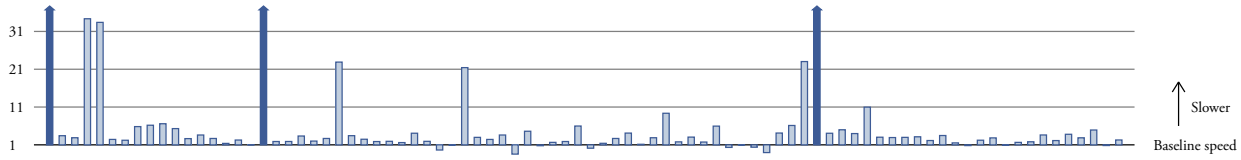
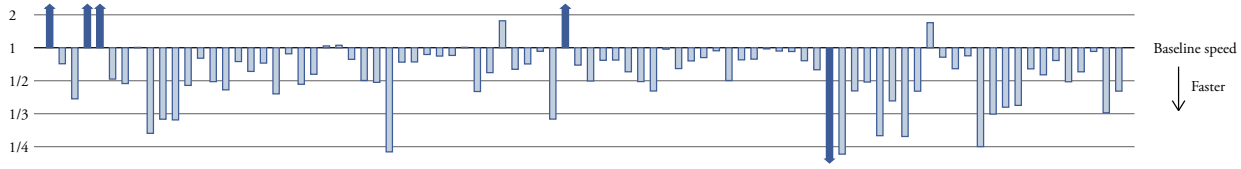


Figure 10. XmlPrime stylesheet compile speeds relative to SaxonEE-9.5-J



The most appropriate way of using these results is not to compute a crude ranking, but to try to understand where each product is stronger and where it is weaker. However, isolating the features of the individual tests to achieve such an understanding is not an easy task.

6.4. Comparing Saxon 9.5 with Saxon 9.6

As already stated, a key aim in writing this benchmark was to enable us to avoid regression when shipping new Saxon releases. Measuring Saxon 9.6 (currently under development) with the current 9.5 release is therefore particularly relevant.

Looking across all tests, this is the data we are currently seeing:

Figure 11. SaxonEE-9.6 file to file transform speeds relative to SaxonEE-9.5

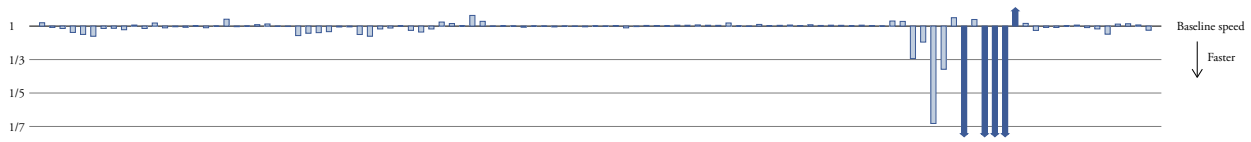


Figure 12. SaxonEE-9.6 tree to tree transform speeds relative to SaxonEE-9.5

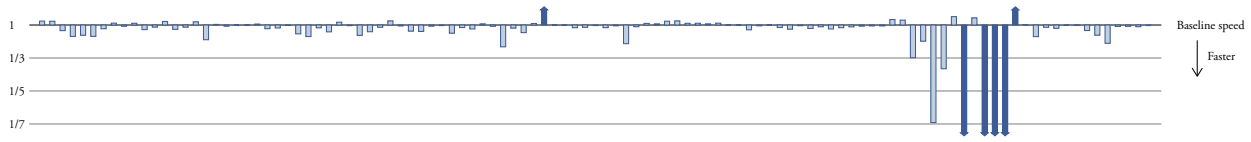
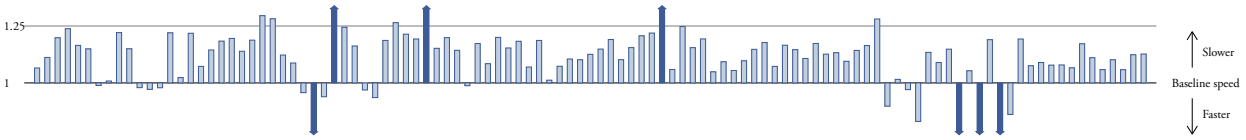


Figure 13. SaxonEE-9.6 stylesheet compile speeds relative to SaxonEE-9.5



What we would expect to find here is that for the majority of tests, the performance is much the same between the two releases, but for a minority of tests, the performance may benefit from deliberate enhancements in particular areas, or it may reveal performance bugs that we need to address before shipping the final product.

There is a lot of noise in the results. There's no reason at all why the performance ratio between the two releases should be different for tree-to-tree transforms than for file-to-file transforms. The fact that some of the ratios are significantly different can be taken as a measure of the inaccuracies that arise during measurement of Java performance, caused (we believe) by the failure to suppress unrelated activity on the system under test, for example Java garbage collection, network traffic or virus checking.

Nevertheless, the overall picture is good. Most tests are showing a performance ratio close to one, and a cluster of tests are showing a substantial improvement.

This cluster of tests was specifically designed to assess the effectivness of a redesign in 9.6 in the implementation of maps. Maps are a new feature in XSLT 3.0, providing a data structure akin to what some languages call "dictionaries" or "associative arrays": a set of key-value pairs providing efficient access to the value associated with any key. As befits a functional language, the maps in XSLT 3.0 are immutable, and herein lies the performance challenge. In Saxon 9.5, the implementation uses a layering of hash maps and deltas, with deltas being absorbed and consolidated when they reach a certain size. In Saxon 9.6, this has been replaced with a hash trie, similar to the structure used to implement immutable maps in Scala.

To ensure that the new implementation performs better than the old, we wrote a number of tests specifically focused on creating, using, and modifying maps. (We can note in passing that the existence of these tests immediately means that our aggregate performance results attach disproportionate importance to this area.)

Our first results from these tests were very discouraging: they showed the new code running five times slower than the old, and we were almost ready to discard it. However, closer study revealed the reason for the discrepancy: the implementation was caching data relating to the types of the keys and values held in the map, and this cache was not being maintained correctly. Fixing this problem gave us new data which showed map construction and retrieval taking a very similar amount of time to the old release, addition of new entries being slightly faster, and removal of existing entries dramatically faster. Sufficient evidence to justify accepting the new code into the release.

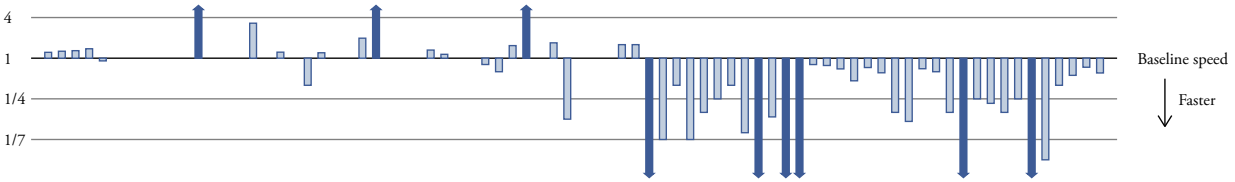
Drilling down even further, we can see variation between the different map tests. For example, test wordmap8 is about 12 times faster on Saxon 9.6, whereas wordmap8a is 1.5 times slower. The two tests are very similar: both construct an index containing all the words in a source document. The difference between the two tests is that wordmap8a, after adding a new entry to the map, counts how many entries are now present in the map using the expression `count(map:keys($map))`. The implication is that in the new data structure, the one thing that runs slower is enumerating the keys present in the map. We can live with this.

The results also show that compile-time performance has got a little worse across all tests in 9.6. This is something we may address before a final release.

6.5. Comparing Saxon/C with libxslt

The newest addition to the Saxonica product stable is Saxon/C: a version of the product issued as a native DLL (or .so) library, suitable for calling from C or C++ applications, together with an interface offering a PHP extension API. This area has for many years been the preserve of the open source libxslt product, which has an excellent reputation, but which (like most of the open source XSLT 1.0 processors) has not been upgraded to XSLT 2.0. Saxonica is aiming to fulfil the demand for an XSLT 2.0 processor in this important space with a version of Saxon that is cross-compiled to native code using the Excelsior JET Java cross-compiler. Clearly the main attraction of Saxon/C to libxslt users will be the ability to take advantage of XSLT 2.0 features, but they will want assurance that performance is adequate.

Figure 14. Saxon/C tree to tree transform speeds relative to libxslt



Our first results comparing Saxon/C with libxslt are shown below. The XT-Speedo driver for libxslt is currently failing many tests when run in file-to-file transform mode, so we present only the tree-to-tree comparison. These show Saxon/C consistently performing better for the tests with larger source documents, and a wide range of results for tests with smaller documents. In the vast majority of cases, however, the speed ratio between the products is between 0.5 and 2.0, so most users are likely to be content. Producing a single metric for the speed ratio is not something we can sensibly do, since it will depend entirely on the selection of tests to run; the only thing we can say with certainty is that Saxon/C consistently performs better for larger source documents.

7. Conclusions

XT-Speedo was written primarily as a resource for use within Saxonica, to enable us to test and compare the performance of our various products. We developed it because the existing performance tests we had been using were woefully inadequate, and because there were too many cases slipping through where, for some particular workload, new Saxon releases showed regression over previous releases despite the release as a whole passing all performance tests.

We have published XT-Speedo as an open source project for a variety of reasons. We want others to be able to reproduce our results and perhaps show us where we have got things wrong or made invalid assumptions. We want others to be able to contribute test data and drivers which we can then benefit from. We also hope that others might be able to take it into areas that we haven't yet tackled, like measuring throughput in a server environment with a concurrent workload.

In this paper we have shown results for a number of performance investigations where we have already found that XT-Speedo gives us new insights into the behaviour of our own products. In some cases the data confirms what we knew and gives us confidence that all is well; in other cases it suggests directions for more detailed investigation, or for remedial action.

We stress again that performance is just one aspect of product quality, one facet that can be used to compare competitive products. It is not the only metric to be used, and is not even our top objective. But we don't want poor performance ever to be a reason for anyone not to use our software. From that perspective, it's not a major concern if Saxon doesn't come at the top of every league table, but the measurements we have taken so far give us confidence that we are in every case within a few percentage points of the leader. More importantly, they tell us where it is possible to do even better.