An XSLT compiler written in XSLT: can it perform?

Michael Kay
Saxonica
<mike@saxonica.com>

John Lumley
jwL Research, Saxonica
<john@jwlresearch.com>

Abstract

This paper discusses the implementation of an XSLT 3.0 compiler written in XSLT 3.0. XSLT is a language designed for transforming XML trees, and since the input and output of the compiler are both XML trees, compilation can be seen as a special case of the class of problems for which XSLT was designed. Nevertheless, the peculiar challenges of multi-phase compilation in a declarative language create performance challenges, and much of the paper is concerned with a discussion of how the performance requirements were met.

1. Introduction

Over the past 18 months we have been working on a new compiler for XSLT, written in XSLT itself: see [1], [2]. At the time of writing, this is nearing functional completeness: it can handle over 95% of the applicable test cases in the W3C XSLT suite. In this paper we'll give a brief outline of the structure of this compiler (we'll call it XX), comparing and constrasting with the established Saxon compiler written in Java (which we will call XJ). And before we do that, we'll give a reminder of the motivation for writing it, from which we can derive some success criteria to decide whether it is fit for release.

Having got close to functional completeness, we now need to assess the compiler's performance, and the main part of this paper will be concerned with the process of getting the compiler to a point where the performance requirements are satisfied.

Because the compiler is, at one level, simply a fairly advanced XSLT 3.0 stylesheet, we hope that the methodology we describe for studying and improving its performance will be relevant to anyone else who has the task of creating performant XSLT 3.0 stylesheets.

2. Motivation

When XSLT 1.0 first emerged in 1999, at least a dozen implementations appeared within a year or two, many of them of excellent quality. Each typically targeted one particular platform: Java, Windows, Python, C, browsers, or whatever. Whatever your choice of platform, there was an XSLT 1.0 processor available (although on the browsers in particular, it took a few years before this goal was achieved).

For a variety of reasons, the W3C's goal of following up XSLT 1.0 with a quick 1.1 upgrade didn't happen, and it was over seven years before XSLT 2.0 came along, followed by a ten year wait for XSLT 3.0. By this time there was a sizeable XSLT user community, but very few of the original XSLT 1.0 vendors had an appetite for the development work needed to implement 2.0 or 3.0. By this stage the number of companies still developing XSLT technology was down to three: Altova and Saxonica, who both had commercial products that brought in enough revenue to fund further development, and a startup, Exselt, which had aspirations to do the same.

This pattern is not at all unusual for successful programming languages. If you look at any successful programming language, the number of implementations has declined over time as a few "winners" have emerged. But the effect of this is that the implementations that remain after the market consolidates come under pressure to cover a broader range of platforms, and that is what is happening with XSLT.

The bottom line is: there is a demand and an opportunity to deliver an XSLT processor that runs on a broader range of platforms. Over the past few years Saxon has slowly (and by a variety of bridge technologies) migrated from its original Java base to cover .NET, C, and Javascript. Currently we see demand from Node.js users. We're also having to think about how to move forward on .NET, because the bridge technology we use there (IKVM) is no longer being actively developed or maintained.

The traditional way to make a programming language portable is to write the compiler in its own language. This was pioneered by Martin Richards with BCPL in the late 1960s, and it has been the norm ever since.

Many people react with a slight horror to the idea of writing an XSLT compiler in XSLT. Surely a language that is mainly used for simple XML-to-HTML conversion isn't up to that job? Well, the language has come on a long way since version 1.0. Today it is a full functional programming language, with higher order functions and a rich set of data types. Moreover, XSLT is designed for performing transformations on trees, and transforming trees is exactly what a compiler does. So the language ought to be up to the job, and if it isn't then we would like to know why.

As we submit this paper, we have produced an almost-complete working XSLT compiler in XSLT 3.0, without encountering any serious obstacles in the lan-

guage that made the task insuperable. We'll give an outline description of how it works in the next section. But the challenging question when we started was always going to be: will it perform? Answering that question is the main purpose of this paper.

Back in 2007, Michael Kay gave a paper on writing an XSLT optimizer in XSLT: see [3]. At that time, one conclusion was that tree copying needed to be much more efficient; the paper gave an example of how a particular optimization rewrite could only be achieved by an expensive copying operation applied to a complete tree. Many optimizations are likely to involve recursive tree rewrites which perform copying of the tree; there is a serious need to optimize this design pattern.

At XML Prague 2018 (see [4]) the same author returned to this question of efficient copying of subtrees, with a proposal for new mechanisms that would allow subtrees to be virtually copied from one tree to another. One of the things examined in this paper is how much of a contribution this makes to the performance of the XSLT compiler (spoiler: the results are disappointing).

3. The Compilers

In this section we will give an outline description of two XSLT compilers: the traditional Saxon compiler, written in Java, which for the purposes of this paper we will call XJ (for "XSLT compiler written in Java"), and the new compiler, which we will call XX (for "XSLT compiler written in XSLT").

Both compilers take as input a source XSLT stylesheet (or more specifically in XSLT 3.0 a source XSLT package, because XSLT 3.0 allows packages to be compiled independently and then subsequently linked to form an executable stylesheet), and both are capable of producing as output an SEF file, which is essentially the compiled and optimized expression tree, serialized in either XML or JSON. The expression tree can then form the input to further operations: it can be directly interpreted, or executable code can be generated in a chosen intermediate or machine language. But we're not concerned in this paper with how it is used, only with how it is generated. The SEF file is designed to be portable. (We have made a few concessions to optimize for a particular target platform, but that should really be done as a post-processing phase.)

3.1. The XJ Compiler

In this section we will give an outline description of how the traditional XSLT compiler in Saxon (written in Java) operates. This compiler has been incrementally developed over a period of 20 years since Saxon was first released, and this description is necessarily an abstraction of the actual code.

It's conventional to describe a compiler as operating in a sequence of phases, even if the phases aren't strictly sequential, and I shall follow this convention. The main phases of the XJ compiler are as follows:

- The XSLT source code is processed using a standard SAX parser to produce a sequence of events representing elements and attributes.
- The content handler that receives this stream of events performs a number of operations on the events before constructing a tree representation of the code in memory. This can be regarded as a pre-processing phase. The main operations during this phase (which operates in streaming mode) are:
  - Static variables and parameters are evaluated
  - Shadow attributes are expanded into regular attributes
  - `use-when` expressions are evaluated and applied
  - `xsl:include` and `xsl:import` declarations are processed.
  - Whitespace text nodes, comments, and processing instructions are stripped.

  The result of this phase is a set of in-memory trees, one for each module in the stylesheet package being compiled. These trees use the standard Saxon "linked tree" data structure, a DOM-like structure where element nodes are represented by custom objects (subclassing the standard `Element` class) to hold properties and methods specific to individual XSLT elements such as `xsl:variable` and `xsl:call-template`.
- Indexing: the top-level components in the stylesheet (such as global variables, named templates, functions, and attribute sets) are indexed by name.
- Attribute processing: for each element in the stylesheet, the attributes are validated and processed as appropriate. This is restricted to processing that can be carried out locally. Attributes containing XPath expressions and XSLT patterns, and other constructs such as type declarations, are parsed at this stage; the result of parsing is an in-memory expression tree.
- Contextual validation: elements are validated "in context" to ensure that they appear in the proper place with the proper content model, and that consistency rules are satisfied. Also during this phase, the first type-checking analysis is carried out, confined to one XPath expression at a time. Type checking infers a static type for each expression and checks this against the required type. If the inferred type and the required type are disjoint, a static error is reported. If the required type subsumes the inferred type, all is well and no further action is needed. If the inferred type overlaps the required type, run-time type checking code is inserted into the expression tree.
- Expression tree generation (referred to, rather unhelpfully, as "compiling"). This phase changes the data representation from the decorated XDM tree used

so far to a pure tree of Java objects representing instructions and expressions to be evaluated. At this stage the boundaries between XSLT and XPath constructs disappear into a single homogenous tree; it becomes impossible to tell, for example, whether a conditional expression originated as an XPath `if-then-else` expression or as an XSLT `xsl:if` instruction.

- A second type-checking phase follows. This uses the same logic as the previous type-checking, but more type information is now available, so the job can be done more thoroughly.
- Optimization: this optional phase walks the expression tree looking for rewrite opportunities. For example, constant expressions can be evaluated eagerly; expressions can be lifted out of loops; unnecessary sort operations (of nodes into document order) can be eliminated; nested-loop joins can be replaced with indexed joins.
- When XSLT 3.0 streaming is in use, the stylesheet tree is checked for conformance to the streamability rules, and prepared for streamed execution. There is also an option to perform the streamability analysis prior to optimization, to ensure strict conformance with the streaming rules in the language specification (optimization will sometimes rewrite a non-streamable expression into a streamable form, which the language specification does not allow).
- Finally, a stylesheet export file (SEF file) may be generated, or Java bytecode may be written for parts of the stylesheet.

Some of these steps by default are deferred until execution time. When a large stylesheet such as the DocBook or DITA stylesheets is used to process a small source document, many of the template rules in the stylesheet will never fire. Saxon therefore avoids doing the detailed compilation and optimization work on these template rules until it is known that they are needed. Bytecode generation is deferred even longer, so it can focus on the hot-spot code that is executed most frequently.

The unit of compilation is an XSLT package, so there is a process of linking together the compiled forms of multiple packages. Currently a SEF file contains a package together with all the packages it uses, expanded recursively. The SEF file is a direct serialization of the expression tree in XML or JSON syntax. It is typically several times larger than the original XSLT source code. [1]

[1]SEF files generated by the XX compiler are currently rather larger than those generated by XJ. This is partly because XJ has a more aggressive optimizer, which tends to eliminate unnecessary constructs (such as run-time type checks) from the expression tree; and partly because XX leaves annotations on the SEF tree that might be needed in a subsequent optimization phase, but which are not used at run-time. The SEF representation of the XX compiler as produced by XJ is around 2Mb in expanded human-readable XML form; the corresponding version produced by XX is around 6Mb.

3.2. The XX Compiler

The XSLT compiler written in XSLT was developed as a continuation of work on adding dynamic XPath functionality to Saxon-JS ([1])). That project had constructed a robust XPath expression compiler, supporting most of the XPath 3.1 functionality, with the major exception of higher-order functions. Written in JavaScript, it generated an SEF tree for subsequent evaluation within a Saxon-JS context, and in addition determined the static type of the results of this expression.

Given the robustness of this compiler, we set about seeing if an XSLT compiler could be written, using XSLT as the implementation language and employing this XPath compiler, to support some degree of XSLT compilation support within a browser-client. Initial progress on simpler stylesheets was promising, and it was possible to run (and pass!) many of the tests from the XSLT3 test suites. We could even demonstrate a simple interactive XSLT editor/compiler/executor running in a browser. Details of this early progress and the main aspects of the design can be found in [2])

Progress was promising, but it needed a lot of detailed work to expand the functionality to support large areas of the XSLT specification correctly. For example issues such as tracking `xpath-default-namespaces`, namespace-prefix mappings and correctly determining import precedence have many corner cases that, whilst possibly very very rare in use, are actually required for conformance to the XSLT3.0 specification.

At the same time, the possibility of using the compiler within different platform environments, most noteably Node.js, increased the need to build to a very high degree of conformance to specification, while also placing demands on usability (in the form of error messages: the error messages output by a compiler are as important as the executable code), and tolerable levels of both compiling and execution performance. Performance is of course the major topic of this paper, but the work necessary to gain levels of conformance took a lot longer than might originally have been supposed, and work on usability of diagnostics has really only just started. The methodology used had two main parts:

- Checking the compiler against test-cases from the XSLT-3.0 test suite. This was mainly carried out within an interactive web page (running under Saxon-JS) that permitted tests to be selected, run, results checked against test assertions and intermediate compilation stages examined. For example the earliest work looked at compiling stylesheets that used the `xsl:choose` instruction and iteratively coding until all the thirty-odd tests were passing.
- At a later stage, the compiler had advanced sufficiently that it became possible to consider it compiling its own source, which whilst not a sufficient condition is certainly a necessary one. The test would be that after some 3-4 stages of self-compilation, the compiled-compiler 'export' tree would be constant. This

was found to be very useful indeed — for example it uncovered an issue where template rules weren't being rank-ordered correctly, only at the third round of self-compilation.

In this section we'll start by briefly discussing the (top-level) design of the compiler, but will concentrate more on considering the compiler as a program written in XSLT, before it was 'performance optimised'.

In drawing up the original design, a primary requirement was to ease the inevitable and lengthy debugging process. Consequently the design emphasised visibility of internal structures and in several parts used a multiplicity of result trees where essential processing could perhaps have been arranged in a single pass. The top-level design has some six major sequential phases, with a complete tree generated after each stage. These were:

- The first phase, called `static`, handles inclusion/importation of all stylesheet modules, together with XSLT3.0's features of static variables, conditional inclusion and shadow attributes. The result of this phase is a single XDM tree representing the merged stylesheet modules, after processing of `use-when` and shadow attributes, decorated with additional attributes to retain information that would otherwise be lost: original source code location, base URIs, namespace context, import precedence, and attributes such as `exclude-result-prefixes` inherited from the original source structure. [2]

- A normalisation phase where the primary syntax of the stylesheet/package is checked, and some normalisation of common terms (such as boolean-valued attributes 'strings', 'yes','false','0' etc), is carried out. In the absence of a full schema processor, syntax checking involves two stages: firstly a map-driven check that the XSLT element is known, has all required and no unknown attributes and has permitted child and parent elements. Secondly a series of template rules to check more detailed syntax requirements, such as `xsl:otherwise` only being the last child of `xsl:choose` and cases where either `@select` or a sequence constructor child, but not both, are permitted on an element.

- Primary compilation of the *XSLT* declarations and instructions. This phase converts the tree from the source XSLT vocabulary to the SEF vocabulary. This involves collecting a simple static context of declaration signatures (user functions, named templates) and known resources (keys, accumulators, attribute sets, decimal formats) and then processing each top level declaration to produce the necessary SEF instruction trees by recursive push processing, using the static context to check for XSLT-referred resource existence. Note that dur-

[2]We are still debating whether there would be benefits in splitting up this monolithic tree into a "forest" of smaller trees, one for each stylesheet component.

ing this phase XPath expressions and patterns are left as specific single pseudo-instructions for processing during the next phase[3].

- Compilation of the *XPath* and *pattern* expressions, and type-checking of the consequent bindings to variable and parameter values. In this phase the pseudo-instructions are compiled using a `saxon:compile-XPath` extension function, passing both the expression and complete static context (global function signatures, global and local variables with statically determined types, in-scope namespaces, context item type etc.), returning a compiled expression tree and inferred static type. These are then interpolated into the compilation tree recursively, type-checking bindings from the the XPath space to the XSLT space, i.e. typed XSLT variables and functions.

  For pragmatic reasons, the XPath parsing is done in Java or Javascript, not in XSLT. Writing an XPath parser in XSLT is of course possible, but we already had parsers in Java and Javascript, so it was easier to continue using them.

- Link-editing the cross-component references in a *component-binding* phase. References to user functions, named templates, attribute sets and accumulators needed to be resolved to the appropriate component ID and indirected via a binding vector attached to each component[4]

  . After this point the SEF tree is complete and only needs the addition of a checksum and serialization into the final desired SEF file.

Each of these phases involves a set of XSLT template rules organized into one major mode (with a default behaviour of `shallow-copy`), constructing a new result tree, but often there are subsidiary modes used to process special cases. For example, a compiled XPath expression that refers to the (function) `current()` is converted to a `let` expression that records the context item, with any reference to `current()` in the expression tree replaced with a reference to the `let` variable.

The code makes extensive use of tunnel parameters, and very little use of global variables. Indexes (for example, indexes of named templates, functions, and global variables in the stylesheet being compiled) are generally represented using XSLT 3.0 maps held in tunnel parameters.

It's worth stating at this point that the compiler currently does not use a number of XSLT3.0 features at all, for example attribute sets, keys,accumulators, `xsl:import`, schema-awareness, streaming, and higher-order functions. One reason for this was to make it easier to bootstrap the compiler; if it only uses a subset of the language, then it only needs to be able to compile that subset in order to compile itself. Late addition of support for higher-order functions in the XPath compiler makes the latter a distinct possibility, though in early debugging they

[3]In theory XPath compilation could occur during this phase, but the complexity of debugging ruled this out until a very late stage of optimisation.

[4]This derives from combination of separately-compiled packages, where component internals need not be disturbed.

may have been counter-productive. It should also be noted that separate package compilation is not yet supported, so `xsl:stylesheet`, `xsl:transform` and `xsl:package` are treated synonymously.

A run of the compiler can be configured to stop after any particular stage of this process, enabling the tree to be examined in detail.

We'll now discuss this program not as an XSLT compiler, but as an example of a large XSLT transformation, often using its self-compilation as a sample stress-testing workload.

The XX compiler is defined in some 33 modules, many corresponding to the relevant section of the XSLT specification. Internally there is much use of static-controlled inclusion (`@use-when`) to accommodate different debugging, operational and optimisation configurations, but when this phase has been completed, the program (source) tree has some 536 declarations, covering 4,200 elements and some 7,200 attributes, plus another 13,500 attributes added during inclusion to track original source properties, referred to above. The largest declaration (the template that 'XSLT-compiles' the main stylesheet) has 275 elements, the deepest declaration (the primary static processing template) is a tree up to 12 elements deep.

Reporting of syntax errors in the user stylesheet being compiled is currently directed to the `xsl:message` output stream. Compilation continues after an error, at least until the end of the current processing phase. The relevant error-handling code can be overridden (in the usual XSLT manner) in a customization layer to adapt to the needs of different platforms and processing environments.

Top level processing is a chain of five XSLT variables bound to the push ('apply-templates') processing of the previous (tree) result of the chain. We'll examine each of these in turn:

3.2.1. Static inclusion

The XSLT architecture for collecting all the relevant sections of the package source is complicated mainly by two features: firstly the use of static global variables as a method of *meta-programming*, controlling conditional source inclusion, either through `@use-when` decorations or even through *shadow attributes* on inclusion/importation references. Secondly it is critical to compute the import precedence of components, which requires tracking importation depth of the original source. Other minor inconveniences include the possibility of the XSLT version property changing between source components and the need to keep track of original source locations (module names and line numbers).

As static variables can only be global (and hence direct children of a stylesheet) and their scope is (almost) `following-sibling::*/ descendant-or-self::*`, the logic for this phase needs to traverse the top-level sibling declarations maintaining state as it goes (to hold information about the static vari-

ables encountered. The XSLT 3.0 `xsl:iterate` instruction is ideally suited to this task. The body of the `xsl:iterate` instruction collects definitions of static variables in the form of a map. Each component is then processed by template application in mode `static`, collecting the sequence of processed components as a parameter of the iteration. Static expressions may be encountered as the values of static variables, in `[xsl:]use-when` attributes, and between curly braces in shadow attributes; in all cases they are evaluated using the XSLT 3.0 `xsl:evaluate` instruction, with in-scope static variables supplied as the `@with-params` property.[5]The result of the evaluation affects subsequent processing:

- For `[xsl:]use-when`, the result determines whether the relevant subtree is processed using recursive `xsl:apply-templates`, or discarded
- For static variables and parameters, the result is added to a map binding the names of variables to their values, which is made available to following sibling elements as a parameter of the controlling `xsl:iterate`, and to their descendant instructions via tunnel parameters.
- For shadow attributes, the result is injected into the tree as a normal (non-shadow) attribute. For example the shadow attribute `_streamable="{$STREAMING}"` might be rewritten as `streamable="true"`.

Errors found during the evaluation of static XPath expressions will result in exceptions during `xsl:evaluate` evaluation - these are caught and reported.

After each component has been processed through the `static` phase, it is typically added to the `$parts` parameter of the current iteration. In cases where the component was the declaration of a static variable or parameter, the `@select` expression is evaluated (with `xsl:evaluate` and the current bindings of static variables) and its binding added to the set of active static variables.

Processed components which are `xsl:include|xsl:import` declarations are handled within the same iteration. After processing the `@href` property is resolved to recover the target stylesheet[6]. The stylesheet is then read and processed in the `static` mode. The result of this a map with two members — the processed components and the number of prior imports. The processed components are then allocated an importation precedence (recorded as an attribute) dependent upon importation depth/position and any previous precedence and added to the set of components of the including stylesheet[7]. Finally the complete

[5]There is a minor problem here, in that `use-when` expressions are allowed access to some functions, such as `fn:system-property()`, which are not available within `xsl:evaluate`. In a few cases like this we have been obliged to implement language extensions.

[6]A stack of import/included stylesheets is a parameter of the main stylesheet template, the check against self or mutual recursive inclusion.

[7]This complexity is due to the possibility of an importation, referenced via an inclusion, preceding a high-level importation - something permitted in XSLT3.0. Note that the current XX compiler does not itself use `xsl:import` - linkage is entirely through `xsl:include`.

sequence of self and included components are returned as a map with the 'local' importation information. At the very top level the stylesheet is formed by copying all active components into the result tree.

In more general XSLT terms, the processing involves recursive template application for the entire (extended) source tree, with stateful iteration of the body of stylesheets, evaluation and interpolation of static variables with that iteration and a complex multiple-copy mechanism for recording/adjusting importation precedence.

3.2.2. Normalisation

The normalisation phase makes intensive use of XSLT template rules. Generally, each constraint that the stylesheet needs to satisfy (for example, that the `type` and `validation` attributes are mutually exclusive) is expressed as a template rule. Each element in the use stylesheet is processed by multiple rules, achieved by use of the `xsl:next-match` instruction.

The normalisation phase has two main mechanisms. The first involves checking any `xsl:*` element for primary syntax correctness — is the element name known, does it have all required attributes or any un-permitted attributes, do any 'typed' attributes (e.g. boolean) have permitted values and are parent/child elements correct? A simple schema-like data structure[8] was built from which a map *element-name => {permitted attributes, required attributes, permitted parents, permitted children...}* was computed, and this is used during the first stage of syntax checking through a high-priority template. The second mechanism is more ad-hoc, and comprises a large set of templates matching either error conditions such as:

```
<xsl:template match="xsl:choose[empty(xsl:when)]" mode="normalize">
    <xsl:sequence select="f:missingChild(., 'xsl:when')"/>
</xsl:template>
```

which checks that a 'choose' must have a when 'clause', or normalising a value, such as:

```
<xsl:template match="xsl:*/@use-attribute-sets" mode="normalize">
   <xsl:attribute name="use-attribute-sets"
                  select="tokenize(.) ! f:EQName(., current()/..)"/>
</xsl:template>
```

which normalises attribute set names to EQNames.

As far as XSLT processing is concerned, this phases builds one tree in a single pass over the source tree.

[8]Derived from the syntax definitions published with the XSLT3.0 specification

3.2.3. XSLT compilation

The main compilation of the XSLT package involves three main processes — collecting (properties of) all the global resources of the package, such as named templates, user-defined functions, and decimal formats; collecting all template rules into same-mode groups; and a recursive descent compilation of XSLT instructions of each component.

The process for the first is to define a set of some dozen variables, which are then passed as tunnel parameters in subsequent processing, such as:

```
<xsl:variable name="named-template-signatures" as="map(*)">
  <xsl:map>
    <xsl:for-each-group select="f:precedence-sort(xsl:template)"
                        group-by="@name">
      <xsl:variable name="highest" select="
               let $highest-precedence :=
                     max(current-group()/@ex:precedence)
               return
                 current-group()[@ex:precedence = $highest-precedence]"/>
      <xsl:if test="count($highest) gt 1">
        <xsl:sequence select="f:syntax-error('XTSE0660',
            'Multiple declarations of ' || name() || ' name=' || @name ||
            ' at highest import precedence')"/>
      </xsl:if>
      <xsl:variable name="params"
                    select="$highest/xsl:param[not(@tunnel eq 'true')]"/>
      <xsl:map-entry key="$highest/@name" select="map{
            'params': f:string-map($params/map:entry(@name,
                                map{'required': @required eq 'true',
                                    'type': @as})),
            'required': $params[@required eq 'true']/@name,
            'type': ($highest/@as,'item()*')[1]
            }"/>
    </xsl:for-each-group>
  </xsl:map>
</xsl:variable>
```

which both checks for conflicting named templates, handles differing precedences and returns a map of names/signatures. This can then of course be referenced in compiling a `xsl:call-template` instruction to check both the existence of the requested template and the names/types of its parameters, as well as the implied result type.

All template rules are first expanded into 'single mode' instances by copying for each referred `@mode` token[9]

. From this all used modes can be determined and for each a mode component is constructed and populated with the relevant compiled templates. A pattern

matching template is compiled with a simple push template, that leaves the `@match` as a `pattern` pseudo-instruction, and the body as a compiled instruction tree. The design currently involves the construction of *three* trees for each template during this stage.

The bulk of the XSLT compiling is a single recursive set of templates, some of which check for error conditions[10], most of which generate an SEF instruction and recursively process attributes and children, such as:

```
<xsl:template match="xsl:if" mode="sef">
  <xsl:param name="attr" as="attribute()*" select="()"/>
  <choose>
    <xsl:sequence select="$attr"/>
    <xsl:call-template name="record-location"/>
    <xsl:apply-templates select="@test" mode="create.xpath"/>
    <xsl:call-template name="sequence-constructor"/>
    <true/>
    <empty/>
  </choose>
</xsl:template>
```

which generates a `choose` instruction for `xsl:if`, with any required attributes attached (often to identify the role of the instruction in its parent), records the source location, creates an `xpath` pseudo-instruction for the test expression, adds the sequence constructor and appends an empty 'otherwise' case.

Local `xsl:variable` and `xsl:param` instructions are replaced by `VARDEF` and `PARAMDEF` elements for processing during XPath compiling.

The final result of this phase is a `package` with a series of component children corresponding to compiled top-level declarations and modes with their template rules.

3.2.4. XPath compiling and type checking

In this phase the package is processed to compile the `xpath` and `pattern` pseudo-instructions, determine types of variables, parameters, templates and functions and propogate and type-check cross-references. As such the key action is an iteration through the children of any element that contains `VARDEF` or `PARAMDEF` children, accumulating variable/type bindings that are passed to the XPath compiler. Unlike the similar process during the static phase, in this case the architecture is to use a recursive named template, to build a nested tree of `let` bindings, propogating variable type bindings downwards and sequence constructor result types back upwards. In this case the result type is held as an `@sType` attribute value. The

[9]This has the unfortunate effect of duplicating bodies (and compilation effort thereof) for multi-mode templates — an indexed design might be a possibility, but may require SEF additions

[10]And perhaps should exist in in the normalisation phase

top of this process determines the type of a component's result, which can be checked against any declaration (I.e.`@as`)

This phase requires a static type system and checker which generates a small map structure (*baseType, cardinality.....*) from the XSchema string representation and uses this to compare supplied and required types, determining whether there is match, total conflict or a need for runtime checking. Written in XSLT, one drawback is that the type of result trees is returned as a string on an attribute, requiring reparsing[11].

Some instructions require special processing during this phase. Some, e.g. `forEach`, alter the type of the context item for evaluation of their sequence constructor body. Others, such as `choose`, return a result whose type is the union of those of their 'action' child instructions. These are handled by separate templates for each case.

Finally the `pattern` instructions are compiled. For accumulators and keys their result trees are left on their parent declaration. For template rules, in addition, the default priority of the compiled pattern is calculated if required and with a priority and import precedence available for every template rule in a mode, they can be rank ordered.

3.2.5. Component binding

At this point all the compiling is complete, but all the cross-component references must be linked. This is via a two stage process: firstly building a component 'name' to component number ('id') map. Then each component is processed in turn, collecting all descendant references (`call-template`, user-function calls, key and accumulator references etc.) and building an indirect index on the component head, whose entries are then interpolated into the internal references during a recursive copy.[12]

3.2.6. Reflections on the design

We must emphasise that this architecture was designed for ease of the (complex) debugging anticipated, valuing visibility over performance. Several of the phases could be coalesced, reducing the need for multiple copying of large trees. For example the normalisation and the compiling phases could be combined into a single set of templates for each XSLT element, the body of which both checked

[11]Changing the canonical return to a (map) tuple of *(tree,type)* could be attempted but it would make the use of a large set of element-matching templates completely infeasible.

[12]In XSLT 2.0, all references to components such as variables, named templates, and functions could be statically resolved. This is no longer the case in XSLT 3.0, where components (if not declared private or final) can be overridden in another stylesheet package, necessitating a deferred binding process which in Saxon is carried out dynamically at execution time. The compiler generates a set of binding vectors designed to make the final run-time binding operation highly efficient.

syntax and compiled the result[13]. Similarly the XSLT and XPath compilation phases could be combined, incorporating static type checking in the same operation. And some ot the operations, especially in type representation, may be susceptible to redesign. Some of these will be discussed in the following sections

3.3. Comparing the Two Compilers

At a high level of description, the overall structure of the two compilers is not that different. Internally, the most conspicuous difference is in the internal data structures.

Both compilers work initially with the XDM tree representation of the stylesheet as a collection of XML documents, and then subsequently transform this to an internal representation better suited to operations such as type-checking.

For the XJ compiler, this internal representation is a mutable tree of Java objects (each node in the tree is an object of class `Expression`, and the references to its subexpressions are via objects of class `Operand`). The final SEF output is then a custom serialization of this expression tree. The expression tree is mutable, so there is no problem decorating it with additional properties, or with performing local rewrites that replace selected nodes with alternatives. It's worth noting, however, that the mutability of the tree has been a rich source of bugs over the years. Problems can and do arise through properties becoming stale (not being updated when they should be), through structural errors in rewrite operations (leading for example to nodes having multiple parents), or through failure to keep the structure thread-safe.

For the XX compiler, the internal representation is itself an XDM node tree, augmented with maps used primarily as indexes into the tree. This creates two main challenges. Firstly, the values of elements and attributes are essentially limited to strings; this leads to clumsy representation of node properties such as the inferred type, or the set of in-scope namespaces. As we will see, profiling showed that a lot of time was being spent translating such properties from a string representation into something more suitable for processing (and back). Secondly, the immutability of the tree leads to a lot of subtree copying. To take just one example, there is a process that allocates distinct slot numbers to all the local variable declarations in a template or function. This requires one pass over the subtree to allocate the numbers (creating a modified copy of the subtree as it goes). But worse, on completion we want to record the total number of slots allocated as an attribute on the root node of the subtree; the simplest way of achieving this is to

[13]This is something of an anathema to accepted XSLT wisdom in the general case, where a mutliplicity of pattern-matching templates is encouraged, but in this case the 'processed target', i.e. the XSLT language isn't going to change.

copy the whole subtree again. As we will see, subtree copying contributes a considerable part of the compilation cost.

4. Compiler Performance

The performance of a compiler matters for a number of reasons:

- **Usability and Developer Productivity.** Developers spend most of their time iteratively compiling, discovering their mistakes, and correcting them. Reducing the elapsed time from making a mistake to getting the error message has a critical effect on the development experience. Both the authors of this paper have been around long enough to remember when this time was measured in hours. Today, syntax-directed editors often show you your mistakes before you have finished typing. In an XML-based IDE such as oXygen, the editing framework makes repeated calls on the compiler to get diagnostics behind the scenes, and the time and resource spent doing this has a direct impact on the usability of the development tool.
- **Production efficiency.** In some environments, for example high volume transaction processing, a program is compiled once and then executed billions of times. In that situation, compile cost is amortized over many executions, so the cost of compiling hardly matters. However, there are other production environments, such as a publishing workflow, where it is common practice to compile a stylesheet each time it is used. In some cases, the cost of compiling the stylesheet can exceed the cost of executing it by a factor of 100 or more, so the entire elapsed time of the publishing pipeline is in fact dominated by the XSLT compilation cost.
- **Spin-off benefits.** For this project, we also have a third motivation: if the compiler is written in XSLT, then making the compiler faster means we have to make XSLT run faster, and if we can make XSLT run faster, then the execution time of other (sufficiently similar) stylesheets should all benefit. Note that "making XSLT run faster" here doesn't just mean raw speed: it also means the provision of instrumentation and tooling that helps developers produce good, fast code.

4.1. Methodology

Engineering for performance demands a disciplined approach.

- The first step is to set requirements, which must be objectively measurable, and must be correlated with the business requirements (that is, there must be a good answer to the question, what is the business justification for investing effort to make it run faster?)

Often the requirements will be set relative to the status quo (for example, improve the speed by a factor of 3). This then involves measurement of the status quo to establish a reliable baseline.

- Then it becomes an iterative process. Each iteration proceeds as follows:
  - Measure something, and (important but easily forgotten) keep a record of the measurements.
  - Analyze the measurements and form a theory about why the numbers are coming out the way they are.
  - Make a hypothesis about changes to the product that would cause the numbers to improve.
  - Implement the changes.
  - Repeat the measurements to see what effect the changes had.
  - Decide whether to retain or revert the changes.
  - Have the project requirements now been met? If so, stop. Otherwise, continue to the next iteration.

4.2. Targets

For this project the task we want to measure and improve is the task of compiling the XX compiler. We have chosen this task because the business objective is to improve the speed of XSLT compilation generally, and we think that compiling the XX compiler is likely to be representative of the task of compiling XSLT stylesheets in general; furthermore, because the compiler is written in XSLT, the cost of compiling is also a proxy for the cost of executing arbitrary XSLT code. Therefore, any improvements we make to the cost of compiling the compiler should benefit a wide range of other everyday tasks.

There are several ways we can compile the XX compiler (remembering that XX is just an XSLT stylesheet).

We can describe the tasks we want to measure as follows:

E0: $C_{EEJ}(XX) \rightarrow XX_0$ (240ms ➔ 240ms)

Exercise E0 is to compile the stylesheet XX using the built-in XSLT compiler in Saxon-EE running on the Java platform (denoted here C_{EEJ}) to produce an output SEF file which we will call XX_0. The baseline timing for this task (the status quo cost of XSLT compilation) is 240ms; the target remains at 240ms.

"E1: $T_{EEJ}(XX, XX_0) \rightarrow XX_1$ (2040ms ➔ 720ms)"

Exercise E1 is to apply the compiled stylesheet XX_0 to its own source code, using as the transformation engine Saxon-EE on the Java platform (denoted here T_{EEJ}(source, stylesheet)), to produce an output SEF file which we will call XX_1. Note that XX_0 and XX_1 should be functionally equivalent, but they are not required to be identical (the two compilers can produce different executables, so

long as the two executables do the same thing). The measured baseline cost for this transformation is 2040ms, which means that the XX compiler is 8 or 9 times slower than the existing Saxon-EE/J compiler. We would like to reduce this overhead to a factor of three, giving a target time of 720ms.

"E2: $T_{JSN}(XX, XX_0) \rightarrow XX_2$ (90s ➔ 3s)"

Exercise E2 is identical, except that this time we will use as our transformation engine Saxon-JS running on Node.js. The ratio of the time for this task compared to E1 is a measure of how fast Saxon on Node.js runs relative to Saxon on Java, for one rather specialised task. In our baseline measurements, this task takes 90s – a factor of 45 slower. That's a challenge. Note that this doesn't necessarily mean that every stylesheet will be 45 times slower on Node.js than on Java. Although we've described XX as being written in XSLT, that's a simplification: the compiler delegates XPath parsing to an external module, which is written in Java or Javascript respectively. So the factor of 45 could be due to differences in the two XPath parsers. At the moment, though, we're setting requirements rather than analysing the numbers. We'll set ourselves an ambitious target of getting this task down to three seconds.

"E3: $T_{EEJ}(XX, XX_1) \rightarrow XX_3$ (2450ms ➔ E1 + 25%) "

Exercise E3 is again similar to E1, in that it is compiling the XX compiler by applying a transformation, but this time the executable stylesheet used to perform the transformation is produced using the XX compiler rather than the XJ compiler. The speed of this task, relative to E1, is a measure of how good the code produced by the XX compiler is, compared with the code produced by the XJ compiler. We expected and were prepared to go with it being 25% slower, but found on measurement that we were already exceeding this goal.

There are of course other tasks we could measure; for example we could do the equivalent of E3, but using Saxon-JS rather than Saxon-EE/J. However, it's best to focus on a limited set of objectives. Repeatedly compiling the compiler using itself might be expected to converge, so that after a couple of iterations the output is the same as the input: that is, the process should be idempotent. Although technically idempotence is neither a necessary nor a sufficient condition of correctness, it is easy to assess, so as we try to improve performance, we can use idempotence as a useful check that we have not broken anything. We believe that if we can achieve these numbers, then we have an offering on Node.js that is fit for purpose; 3 seconds for a compilation of significant size will not cause excessive user frustration. Of course, this is a "first release" target and we would hope to make further improvements in subsequent releases.

4.3. Measurement Techniques

In this section we will survey the measurement techniques used in the course of the project. The phase of the project completed to date was, for the most part,

running the compiler using Saxon-EE on the Java platform, and the measurement techniques are therefore oriented to that platform.

We can distinguish two kinds of measurement: bottom-line measurement intended directly to assess whether the compiler is meeting its performance goals; and internal measurements designed to achieve a better understanding of where the costs are being incurred, with a view to making internal changes.

- The bottom-line execution figures were obtained by running the transformation from the command line (within the IntelliJ development environment, for convenience), using the `-t` and `-repeat` options.

  The `-t` option reports the time taken for a transformation, measured using Java's `System.nanoTime()` method call. Saxon breaks the time down into stylesheet compilation time, source document parsing/building time, and transformation execution time.

  The `-repeat` option allows the same transformation to be executed repeatedly, say 20 or 50 times. This delivers results that are more consistent, and more importantly it excludes the significant cost of starting up the Java Virtual Machine. (Of course, users in real life may experience the same inconsistency of results, and they may also experience the JVM start-up costs. But our main aim here is not to predict the performance users will obtain in real life, it is to assess the impact of changes we make to the system.)

  Even with these measures in place, results can vary considerably from one run to another. That's partly because we make no serious attempt to prevent other background work running on the test machine (email traffic, virus checkers, automated backups, IDE indexing), and partly because the operating system and hardware themselves adjust processor speed and process priorities in the light of factors such as the temperature of the CPU and battery charge levels. Some of the changes we have been making might only deliver a 1% improvement in execution speed, and 1% is unfortunately very hard to measure when two consecutive runs, with no changes at all to the software, might vary by 5%. Occasionally we have therefore had to "fly blind", trusting that changes to the code had a positive effect even though the confirmation only comes after making a number of other small changes whose cumulative effect starts to show in the figures.

  Generally we trust a good figure more than we trust a bad figure. There's an element of wishful thinking in this, of course; but it can be justified on the basis that random external factors such as background processes can slow a test run down, but they are very unlikely to speed it up. The best figures we got were usually when we ran a test first thing in the morning on a cold machine.

- *Profiling:* The easiest way to analyse where the costs are going for a Saxon XSLT transformation is to run with the option `-TP:profile.html`. This gener-

ates an HTML report showing the gross and net time spent in each stylesheet tempate or function, together with the number of invocations. This output is very useful to highlight hot-spots.

Like all performance data, however, it needs to be interpreted with care. For example, if a large proportion of the time is spent evaluating one particular match pattern on a template rule, this time will not show up against that template rule, but rather against all the template rules containing an `xsl:apply-templates` instruction that causes the pattern to be evaluated (successfully or otherwise). This can have the effect of spreading the costs thinly out among many other templates.

- *Subtractive measurement:* Sometimes the best way to measure how long something is taking is to see how much time you save by not doing it. For example, this technique proved the best way to determine the cost of executing each phase of the compiler, since the compiler was already structured to allow early termination at the end of any phase. It can also be used in other situations: for example, if there is a validation function testing whether variable names conform to the permitted XPath syntax, you can assess the cost of that operation by omitting the validation. (As it happens, there's a cheap optimization here: test whether names are valid at the time they are declared, and rely on the binding of references to their corresponding declarations to catch any invalid names used as variable or function references).
- A corresponding technique, which we had not encountered before this project, might be termed *additive measurement*. Sometimes you can't cut out a particular task because it is essential to the functionality; but what you can do is to run it more than once. So, for example, if you want to know how much time you are spending on discovering the base URIs of element nodes, one approach is to modify the relevant function so it does the work twice, and see how much this adds to total execution time.
- *Java-level profiling.* There's no shortage of tools that will tell you where your code is spending its time at the Java level. We use JProfiler, and also the basic `runhprof` reports that come as standard with the JDK. There are many pitfalls in interpreting the output of such tools, but they are undoubtedly useful for highlighting problem areas. Of course, the output is only meaningful if you have some knowledge of the source code you are profiling, which might not be the case for the average Saxon XSLT user. Even without this knowledge, however, one can make inspired guesses based on the names of classes and methods; if the profile shows all the time being spent in a class called `DecimalArithmetic`, you can be fairly sure that the stylesheet is doing some heavy computation using `xs:decimal` values.
- *Injected counters.* While timings are always variable from one run to another, counters can be 100% replicable. Counters can be injected into the XSLT code

by calling `xsl:message` with a particular error code, and using the Saxon extension function `saxon:message-count()` to display the count of messages by error code. Internally within Saxon itself, there is a general mechanism allowing counters to be injected: simply add a call on `Instrumentation.count("label")` at a particular point in the source code, and at the end of the run it will tell you the number of executions for each distinct label. The label does not need to be a string literal; it could, for example, be an element name, used to count visits to nodes in the source document by name. This is how we obtained the statistics (mentioned below) on the incidence of different kinds of XPath expression in the stylesheet.

The information from counters is indirect. Making a change that reduces the value of a counter gives you a warm feeling that you have reduced costs, but it doesn't quantify the effect on the bottom line. Nevertheless, we have found that strategically injected counters can be a valuable diagnostic tool.

- *Bytecode monitoring.* Using the option -TB on the Saxon command line gives a report on which parts of the stylesheet have been compiled into Java bytecode, together with data on how often these code fragments were executed. Although it was not orginally intended for the purpose, this gives an indication of where the hotspots in the stylesheet are to be found, at a finer level of granularity than the -TP profiling output.

A general disadvantage of all these techniques is that they give you a worm's-eye view of what's going on. It can be hard to stand back from the knowledge that you're doing millions of string-to-number conversions (say), and translate this into an understanding that you need to fundamentally redesign your data structures or algorithms.

4.4. Speeding up the XX Compiler on the Java Platform

The first task we undertook (and the only one fully completed in time for publication) was to measure and improve the time taken for compiling the XX compiler, running using Saxon-EE on the Java platform. This is task E1 described above, and our target was to improve the execution time from 2040ms to 720ms.

At this stage it's probably best to forget that the program we're talking about is a compiler, or that it it is compiling itself. Think of it simply as an ordinary, somewhat complex, XML transformation. We've got a transformation defined by a stylesheet, and we're using it to transform a set of source XML documents into a result XML document, and we want to improve the transformation time. The fact that the stylesheet is actually an XSLT compiler and that the source document is the stylesheet itself is a complication we don't actually need to worry about.

We started by taking some more measurements, taking more care over the measurement conditions. We discovered that the original figure of 2040ms was obtained with bytecode generation disabled, and that switching this option on

improved the performance to 1934ms. A gain of 5% from bytecode generation for this kind of stylesheet is not at all untypical (significantly larger gains are sometimes seen with stylesheets that do a lot of arithmetic computation, for example).

Analysis of Stylesheet Execution Time

Total time: 4841.351 milliseconds

Time spent in each template, function or global variable:

The table below is ordered by the total net time spent in the template, function or global variable. Gross time means the time including called templates and functions (recursive calls only count from the original entry); net time means time excluding time spent in called templates and functions.

module	line	instruction	count	average time (gross/ms)	total time (gross/ms)	average time (net/ms)	total time (net/ms)
xpath.xsl	659	template *:xpath	2,799	0.502	1,403.761	0.438	1,227.262
component-bindings.xsl	41	template ex:co | ex:accumulator | ex:key	203	1.252	254.205	1.010	204.953
namespaces.xsl	79	function f:recover-namespaces	3,161	0.061	193.844	0.061	193.844
namespaces.xsl	56	function f:encode-namespaces	4,360	0.042	184.961	0.042	184.961
xpath.xsl	400	template process-variables	1,609	1.022	1,645.059	0.099	159.342
static.xsl	627	template staticText	4,348	0.105	456.259	0.036	158.129
type.xsl	170	function t:type	7,244	0.024	177.038	0.022	156.808
normalize.xsl	51	template *	4,341	0.101	439.124	0.035	149.904
xpath.xsl	794	template *:pattern	362	0.436	157.695	0.388	140.300

Figure 1. Example of -TP profile output

Our next step was to profile execution using the `-TP` option. Figure 1 shows part of the displayed results. The profile shows that 25% of the time is spent in a single template, the template rule with `match="*:xpath`. This is therefore a good candidate for further investigation.

4.4.1. XPath Parsing

The `match="*:xpath` template is used to process XPath expressions appearing in the stylesheet. As already mentioned, the XPath parsing is not done in XSLT code, but by a call-out to Java or Javascript (in this case, Java). So the cost of this template includes all the time spent in the Java XPath parser. However, the total time spent in this template exceeds the total cost of running the XJ compiler, which is using the same underlying XPath parser, so we know that it's not simply an inefficiency in the parser.

Closer examination showed that the bulk of the cost was actually in setting up the data structures used to represent the static context of each XPath expression. The static context includes the names and signatures of variables, functions, and types that can be referenced by the XPath expression, and it is being passed from the XSLT code to the Java code as a collection of maps. Of course, the average XPath expression (think `select="."`) doesn't use any of this information, so the whole exercise is wasted effort.

Reducing this cost used a combination of two general techniques:

- *Eager evaluation:* A great deal of the static context is the same for every XPath expression in the stylesheet: every expression has access to the same functions and global variables. We should be able to construct this data structure once, and re-use it.

- *Lazy evaluation:* Other parts of the static context (notably local variables and namespace bindings) do vary from one expression to another, and in this case the trick is to avoid putting effort into preparing the information in cases when it is not needed. One good way to do this would be through callbacks - have the XPath parser ask the caller for the information on demand (through callback functions such as a variable resolver and a namespace resolver, as used in the JAXP XPath API). However, we decided to rule out use of higher-order functions on this project, because they are not available on all Saxon versions. We found an alternative that works just as well: pass the information to the parser in whatever form it happens to be available, and have the parser do the work of digesting and indexing it only if it is needed.

These changes brought execution time down to 1280ms, a good step towards the target of 720ms.

Profiling showed that invocation of the XPath parser still accounted for a large proportion of the total cost, so we subsequently revisited it to make further improvement. One of the changes was to recognize simple path expressions like `.` and `()`. We found that of 5100 path expressions in the stylesheet, 2800 had 5 or fewer tokens; applying the same analysis to the Docbook stylesheets gave similar results. The vast majority of these fall into one of around 25 patterns where the structure of the expression can be recognised simply from the result of tokenization: if the sequence of tokens is (dollar, name) then we can simply look up a function that handles this pattern and converts it into a variable reference, bypassing the recursive-descent XPath parser entirely. Despite a good hit rate, the effect of this change on the bottom line was small (perhaps 50ms, say 4%). However, we decided to retain it as a standard mechanism in the Java XPath parser. The benefit for Java applications using XPath to navigate the DOM (where it is common practice to re-parse an XPath expression on every execution) may be rather greater.

4.4.2. Further investigations

After the initial success improving the interface to the XPath parser, the profile showed a number of things tying for second place as time-wasters: there are about 10 entries accounting for 3% of execution time each, so we decided to spread our efforts more thinly. This proved challenging because although it was easy enough to identify small changes that looked beneficial, measuring the effect was tough, because of the natural variation in bottom-line readings.

Here are some of the changes we made in this next stage of development:

- During the first ("static") phase of processing, instead of recording the full set of in-scope namespace bindings on every element, record it only if the namespace context differs from the parent element. The challenge is that there's no

easy way to ask this question in XPath; we had to introduce a Saxon extension function to get the information (`saxon:has-local-namespaces()`).

- The template rule used to strip processing instructions and comments, merge adjacent text nodes, and strip whitespace, was contributing 95ms to total execution time (say 7%). Changing it to use `xsl:iterate` instead of `xsl:for-each-group` cut this to 70ms.
- There was a very frequently executed function `t:type` used to decode type information held in string form. Our initial approach was to use a memo function to avoid repeated decoding of the same information. Eventually however, we did a more ambitious redesign of the representation of type information (see below).
- The compiler maintains a map-based data structure acting as a "schema" for XSLT to drive structural validation. This is only built once, in the form of a global variable, but when the compiler is only used for one compilation, building the structure is a measurable cost. We changed the code so that instead of building the structure programmatically, it is built by parsing a JSON file.
- We changed the code in the final phase where component bindings are fixed up to handle all the different kinds of component references (function calls, global variable references, call-template, attribute set references, etc) in a single pass, rather than one pass for each kind of component. There were small savings, but these were negated by fixing a bug in the logic that handled duplicated names incorrectly. (This theme came up repeatedly: correctness always comes before performance, which sometimes means the performance numbers get worse rather than better.)
- There's considerable use in the compiler of XSLT 3.0 maps. We realised there were two unnecessary sources of inefficiency in the map implementation. Firstly, the specification allows the keys in a map to be any atomic type (and the actual type of the key must be retained, for example whether it is an `xs:NCName` rather than a plain `xs:string`). Secondly, we're using an "immutable" or "persistent" map implementation (based on a hash trie) that's optimized to support `map:put()` and `map:remove()` calls, when in fact these hardly ever occur: most maps are never modified after initial construction. We added a new map implementation optimized for string keys and no modification, and used a combination of optimizer tweaks and configuration options to invoke it where appropriate.

Most of these changes led to rather small benefits: we were now seeing execution times of around 1120ms. It was becoming clear that something more radical would be needed to reach the 720ms goal.

At this stage it seemed prudent to gather more data, and in particular it occurred to us that we did not really have numbers showing how much time was spent in each processing phase. We tried two approaches to measuring this: one was to output timestamp information at the end of each phase, the other was "subtractive measurement" - stopping processing before each phase in turn, and looking at the effect on the bottom line. There were some interesting discrepancies in the results, but we derived the following "best estimates":

Table 1. Execution times for each phase of processing

Static processing	112ms
Normalisation	139ms
"Compilation" (generating initial SEF tree)	264ms
XPath parsing	613ms
Component binding	55ms

These figures appear to contradict what we had learnt from the `-TP` profile information. It seems that part of the discrepancy was in accounting for the cost of serializing the final result tree: serialization happens on-the-fly, so the cost appears as part of the cost of executing the final phase of the transformation, and this changes when the transformation is terminated early. It's worth noting also that when `-TP` is used, global variables are executed eagerly, so that the evaluation cost can be separated out; the tracing also suppresses some optimizations such as function inlining. Heisenberg rules: measuring things changes what you are measuring.

At this stage we decided to study how much time was being spent copying subtrees, and whether this could be reduced.

4.4.3. Subtree Copying

At XML Prague 2018, one of the authors presented a paper on mechanisms for tree copying in XSLT; in particular, looking at whether the costs of copying could be reduced by using a "tree grafting" approach, where instead of making a physical copy of a subtree, a virtual copy could be created. This allows one physical subtree to be shared between two or more logical trees; it is the responsibility of the tree navigator to know which real tree it is navigating, so that it can do the right thing when retracing its steps using the ancestor axis, or when performing other context-dependent operations such as computing the base URI or the in-scope namespaces of a shared element node.

In actual fact, two mechanisms were implemented in Saxon: one was a fast 'bulk copy" of a subtree from one TinyTree to another (exploiting the fact that both use the same internal data structure to avoid materializing the nodes in a

neutral format while copying), and the other was a virtual tree graft. The code for both was present in the initial Saxon 9.9 release, though the "bulk copy" was disabled. Both gave good performance results in synthetic benchmarks.

On examination, we found that the virtual tree grafting was not being extensively used by the XX compiler, because the preconditions were not always satisfied. We spent some time tweaking the implementation so it was used more often. After these adjustments, we found that of 93,000 deep copy operations, the grafting code was being used for over 82,000 of them.

However, it was not delivering any performance benefits. The reason was quickly clear: the trees used by the XX compiler typically have a dozen or more namespaces in scope, and the saving achieved by making a virtual copy of a subtree was swamped by the cost of coping with two different namespace contexts for the two logical trees sharing physical storage.

In fact, it appeared that the costs of copying subtrees in this application had very little to do with the copying of elements and attributes, and were entirely dominated by the problem of copying namespaces.

We then experimented by using the "bulk copy" implementation instead of the virtual tree grafting. This gave a small but useful performance benefit (around 50ms, say 4%).

We considered various ways to reduce the overhead of namespace copying. One approach is to try and reduce the number of namespaces declared in the stylesheet that we are compiling; but that's cheating, it changes the input of the task we're trying to measure. Unfortunately the semantics of the XSLT language are very demanding in this area. Most of the namespaces declared in a stylesheet are purely for local use (for use in the names of functions and types, or even for marking up inline documentation), but the language specification requires that all these names are retained in the static context of every XPath expression, for use by a few rarely encountered constructs like casting strings to QNames, where the result depends on the namespaces in the source stylesheet. This means that the namespace declarations need to be copied all the way through to the generated SEF file. Using `exclude-result-prefixes` does not help: it removes the namespaces from elements in the result tree, but not from the run-time evaluation context.

We concluded there was little we could do about the cost of copying, other than to try to change the XSLT code to do less of it. We've got ideas about changes to the TinyTree representation of namespaces that might help[14], but that's out of scope for this project.

Recognizing that the vast majority of components (templates, functions, etc) contain no internal namespace declarations, we introduced an early check during

[14]See blog article: http://dev.saxonica.com/blog/mike/2019/02/representing-namespaces-in-xdm-tree-models.html

the static phase so that such components are labelled with an attribute `uniformNS="true"`. When this attribute is present, subsequent copy operations on elements within the component can use `copy-namespaces="false"` to reduce the cost.

Meanwhile, our study of what was going on internally in Saxon for this transformation yielded a few further insights:

- We found an inefficiency in the way tunnel parameters were being passed (this stylesheet uses tunnel parameters very extensively).
- We found some costs could be avoided by removing an `xsl:strip-space` declaration.
- We found that `xsl:try` was incurring a cost invoking `Executors.newFixedThreadPool()`, just in case any multithreading activity started within the scope of the `xsl:try` needed to be subsequently recovered. Solved this by doing it lazily only in the event that multi-threaded activity occurs.
- We found that during a copy operation, if the source tree has line number information, the line numbers are copied to the destination. Copying the line numbers is inexpensive, but the associated module URI is also copied, and this by default walks the ancestor axis to the root of the tree. This copying operation seems to achieve nothing very useful, so we dropped it.

At this stage, we were down to 825ms.

4.4.4. Algorithmic Improvements

In two areas we developed improvements in data representation and associated algorithms that are worth recording.

Firstly, import precedence.

All the components declared in one stylesheet module have the same import precedence. The order of precedence is that a module has higher precedence than its children, and children have higher precedence than their preceding siblings. The precedence order can be simply computed in a post-order traversal of the import tree. The problem is that annotating nodes during a post-order traversal is expensive: attributes appear on the start-tag, so they have to be written before writing the children. The existing code was doing multiple copy operations of entire stylesheet modules to solve this problem, and the number of copy operations increased with stylesheet depth.

The other problem here is that passing information back from called templates (other than the result tree being generated) is tedious. It's possible, using maps, but generally it's best if you can avoid it. So we want to allocate a precedence to an importing module without knowing how many modules it (transitively) imported.

The algorithm we devised is as follows. First, the simple version that ignores `xsl:include` declarations.

- We'll illustrate the algorithm with an alphabet that runs from A to Z. This would limit the number of imports to 26, so we actually use a much larger alphabet, but A to Z makes things clearer for English readers.
- Label the root stylesheet module Z
- Label its `xsl:import` children, in order, ZZ, ZY, ZX, ...
- Similarly, if a module is labelled PPP, then its `xsl:import` children are labelled, PPPZ, PPPY, PPPX, ...
- The alphabetic ordering of labels is now the import precedence order, highest precedence first.

A slight refinement of the algorithm is needed to handle `xsl:include`. Modules are given a label that reflects their position in the hierarchy taking both `xsl:include` and `xsl:import` into account, plus a secondary label that is changed only by an `xsl:include`, not by an `xsl:import`.

Secondly, types.

We devised a compact string representation of the XPath `SequenceType` construct, designed to minimize the cost of parsing, and capture as much information as possible in compact form. This isn't straightforward, because the more complex (and less common) types, such as function types, require a fully recursive syntax. The representation we chose comprises:

- A single character for the occurrence indicator (such as "?", "*", "+"), always present (use "1" for exactly one, "0" for exactly zero)
- A so-called alphacode for the "principal" type, chosen so that if (and only if) T is a subtype of U, the alphacode of U is a prefix of the alphacode of T. The alphacode for `item()` is a zero-length string; then, for example:
  - `N = node()`
  - `NE = element()`
  - `NA = attribute()`
  - `A = xs:anyAtomicType`
  - `AS = xs:string`
  - `AB = xs:boolean`
  - `AD = xs:decimal`
  - `ADI = xs:integer`
  - `ADIP = xs:positiveInteger`
  - `F = function()`
  - `FM = map()`

and so on.

- Additional properties of the type (for example, the key type and value type for a map, or the node name for element and attribute nodes) are represented by a compact keyword/value notation in the rest of the string.

Functions are provided to convert between this string representation and a map-based representation that makes the individual properties directly accessible. The parsing function is a memo function, so that conversion of commonly used types like "1AS" (a single `xs:string`) to the corresponding map are simply lookups in a hash table.

This representation has the advantage that subtype relationships between two types can in most cases be very quickly established using the `starts-with()` function.

It might be noted that both the data representations described in this section use compact string-based representations of complex data structures. If you're going to hold data in XDM attribute nodes, it needs to be expressible as a string, so getting inventive with strings is the name of the game.

4.4.5. Epilogue

With all the above changes, and a few others not mentioned, we got the elapsed time for the transformation down to 725ms, within a whisker of the target.

It then occurred to us that we hadn't yet used any of Saxon's multi-threading capability. We found a critical `xsl:for-each` at the point where we start XPath processing for each stylesheet component, and added the attribute `saxon:threads="8"`, so effectively XPath parsing of multiple expressions happens in parallel. This brings the run-time down to 558ms, a significant over-achievement beyond the target. It's a notable success-story for use of declarative languages that we can get such a boost from parallel processing just by setting one simple switch in the right place.

4.5. So what about Javascript?

The raison-d'etre of this project is to have an XSLT compiler running efficiently on Node.js; the astute reader will have noticed that so far, all our efforts have been on Java. Phase 2 of this project is about getting the execution time on Javascript down, and numerically this is a much bigger challenge.

Having achieved the Java numbers, we decided we should test the "tuned up" version of the compiler more thoroughly before doing any performance work (there's no point in it running fast if it doesn't work correctly). During the performance work, most of the testing was simply to check that the compiler could compile itself. Unfortunately, as already noted, the compiler only uses a fairly small subset of the XSLT language, so when we went back to running the full test

suite, we found that quite a lot was broken, and it took several weeks to repair the damage. Fortunately this did not appear to negate any of the performance improvements.

Both Node.js and the Chrome browser offer excellent profiling tools for Javascript code, and we have used these to obtain initial data helping us to understand the performance of this particular transformation under Saxon-JS. The quantitative results are not very meaningful because they were obtained against a version of the XX compiler that is not delivering correct results, but they are enough to give us a broad feel of where work is needed.

Our first profiling results showed up very clearly that the major bottleneck in running the XX compiler on Saxon-JS was the XPath parser, and we decided on this basis to rewrite this component of the system. The original parser had two main components: the parser itself, generated using Gunther Rademacher's REx toolkit [reference], and a back-end, which took events from the parser and generated the SEF tree representation of the expression tree. One option would have been to replace the back-end only (which was written with very little thought for efficiency, more as a proof of concept), but we decided instead to write a complete new parser from scratch, essentially as a direct transcription of the Java-based XPath parser present in Saxon.

At the time of writing this new parser is passing its first test cases. Early indications are that it is processing a medium sized XPath expression (around 80 characters) in about 0.8ms, compared with 8ms for the old parser. The figures are not directly comparable because the new parser is not yet doing full type checking. Recall however that the XX compiler contains around 5000 XPath expressions, and if the compiler is ever to run in 1s with half the time spent doing XPath parsing, then the budget is for an average compile time of around 0.1ms per path expression. We know that most of the path expressions are much simpler than this example, so we may not be too far out from this target.

Other than XPath parsing, we know from the measurements that we available that there are a number of key areas where Saxon-JS performance needs to be addressed to be competitive with its Java cousin.

- Pattern matching. Saxon-JS finds the template rule for matching a node (selected using `xsl:apply-templates`) by a sequential search of all the template rules in a mode. We have worked to try and make the actual pattern matching logic as efficient as possible, but we need a strategy that avoids matching every node against every pattern. Saxon-HE on Java builds a decision tree in which only those patterns capable of matching a particular node kind and node name are considered; Saxon-EE goes beyond this and looks for commonality across patterns such as a common parent element or a common predicate. We've got many ideas on how to do this, but a first step would be to replicate the Saxon-HE design, which has served us well for many years.

- Tree construction. Saxon-JS currently does all tree construction using the DOM model, which imposes considerable constraints. More particularly, Saxon-JS does all expression evaluation in a classic bottom-up (pull-mode) fashion: the operands of an expression are evaluated first, and then combined to form the result of the parent expression. This is a very expensive way to do tree construction because it means repeated copying of child nodes as they are added to their parents. We have taken some simple steps to mitigate this in Saxon-JS but we know that more needs to be done. One approach is to follow the Saxon/J product and introduce push-mode evaluation for tree construction expressions, in which a parent instructions effectively sends a start element event to the tree builder, then invokes its child instructions to do the same, then follows up with an end element event. Another approach might be to keep bottom-up construction, but using a lightweight tree representation with no parent pointers (and therefore zero-overhead node copying) until a subtree needs to be stored in a variable, and which point it can be translated into a DOM representation.
- Tree navigation. Again, Saxon-JS currently uses the DOM model, which has some serious inefficiencies built in. The worst is that all searching for nodes by name requires full string comparison against both the local name and the namespace URI. Depending on the DOM implementation, determining the namespace URI of a node can itself be a tortuous process. One way forward might be to use something akin to the Domino model recently introduced for Saxon-EE, where we take a third party DOM *as is*, and index it for fast retrieval. But this has a significant memory footprint. Perhaps we should simply implement Saxon's TinyTree model, which has proved very successful.

All of these areas impact on the performance of the compiler just as much as on the performance of user-written XSLT code. That's the dogfood argument for writing a compiler in its own language: the things you need to do to improve runtime performance are the same things you need to do to improve compiler performance.

5. Conclusions

Firstly, We believe we have shown that implementing an XSLT compiler in XSLT is viable.

Secondly, we have tried to illustrate some of the tools and techniques that can be used in an XSLT performance improvement exercise. We have used these techniques to achieve the performance targets that we set ourselves, and we believe that others can do the same.

The exercise has shown that the problem of the copying overhead when executing complex XSLT transformations is real, and we have not found good answers. Our previous attempts to solve this using virtual trees proved ineffec-

tive because of the amount of context carried by XML namespaces. We will attempt to make progress in this area by finding novel ways of representing the namespace context.

Specific to the delivery of a high-performing implementation of Saxon on the Javascript platform, and in particular server-side on Node.js, we have an understanding of the work that needs to be done, and we have every reason to believe that the same techniques we have successfully developed on the Java platform will deliver results for Javascript users.

References

[1] John Lumley, Debbie Lockett, and Michael Kay. February, 2017. XMLPrague. *XPath 3.1 in the Browser*. 2017. http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf

[2] John Lumley, Debbie Lockett, and Michael Kay. August, 2017. Balisage: The Markup Conference. *Compiling XSLT3, in the browser, in itself*. 2017. https://doi.org/10.4242/BalisageVol19.Lumley01

[3] Michael Kay. August, 2007. Extreme Markup. Montreal, Canada. *Writing an XSLT Optimizer in XSLT*. 2007. http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html

[4] Michael Kay. February, 2018. XML Prague. Prague, Czechia. *XML Tree Models for Efficient Copy Operations*. 2018. http://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf