Saxon: Anatomy of an XSLT processor

What is current state of the art in XSLT optimization?

Michael Kay

April 20, 2005
(First published February 01, 2001)

This article describes how an XSLT processor, in this case the author's open-source Saxon, actually works. Although several open-source XSLT implementations exist (see Resources), no one, as far as we know, has published a description of how they work. This article is intended to fill that gap. It describes the internal workings of Saxon, and shows how this processor addresses XSLT optimization. It also shows how much more work remains to be done. This article assumes that you already know what XSLT is and how it works. (If you need a refresher on the basics of XSLT, see Michael Kay's companion article that gives an overview of XSLT.)

I hope this article serves a number of purposes. First, I hope it will give style sheet authors a feel for what kind of optimizations they can expect an XSLT processor to take care of, and by implication, some of the constructs that are not currently being optimized. Of course, the details of such optimizations vary from one processor to another and from one release to another, but I'm hoping that reading this account will give you a much better feel for the work that's going on behind the scenes. Second, it describes what I believe is the current state of the art in XSLT technology (I don't think Saxon is fundamentally more or less advanced than other XSLT processors in this respect), and describes areas where I think there is scope for further development of techniques. I hope this description might stimulate further work in this area by researchers with experience in compiler and database optimization.

Finally (last and also least), this article is intended to be a starting point for anyone who wants to study the Saxon source code. It isn't written as a tour of the code, and it doesn't assume that you want to go into that level of depth. But if you are interested in getting a higher-level overview than you can get by diving into the JavaDoc specs or the source code itself, you'll probably find this useful.

A couple of caveats: the version I describe is Saxon 6.1, and I describe a functional breakdown of the code that doesn't always map cleanly to the package and module structure. For example, this article describes the compiler and interpreter as separate functional components. But in the actual code, the module that handles the <xsl:choose> instruction, for example, contains both compile-time and run-time code to support this instruction. In case you do want to use this article
as a guide to the code, I've included occasional references to package and module names so you know where to look.

First I'll describe the design of the Saxon product. Saxon is an XSLT processor. That is, it is a program that takes an XML document and a style sheet as input and produces a result document as output. (I'm assuming a knowledge of XSLT, though if you're new to it, you might find my [companion article](#) useful as an introduction.)

Saxon includes a copy of the open-source AElfred XML parser originally written by David Megginson, although it can be used with any other XML parser that implements the Java SAX interface. Saxon also includes a serializer that converts the result tree to XML, HTML, or plain text. The serializer is not technically part of the XSLT processor, but it is essential for practical use.

Saxon implements the TrAX (transformation API for XML) interface defined as part of the JAXP 1.1 Java extensions. You don't need to know about this interface to appreciate this article, but understanding the architecture of TrAX would help you to understand the way Saxon is structured.

**Saxon architecture**

The main components of the Saxon software are shown in Figure 1.

**Figure 1. Saxon architecture**

The tree constructor creates a tree representation of a source XML document. It is used to process both the source document and the style sheet. There are two parts to this:

- The **XML parser** (package `com.icl.saxon.aelfred`) reads the source document and notifies events such as the start and end of an element.
- The **tree builder** (module `com.icl.saxon.Builder`) is notified of these events, and uses them to construct an in-memory representation of the XML document.
The interface between the parser and the builder is the Java SAX2 API. Although this SAX API has been only informally standardized, it is implemented by half a dozen freely available XML parsers, allowing Saxon to be used with any of these parsers interchangeably. In between the parser and the tree builder sits a component which I call the **Stripper** (I couldn't resist the name): The Stripper performs the function of removing whitespace text nodes before they are added to the tree, according to the `<xsl:preserve-space>` and `<xsl:strip-space>` directives in the style sheet (module `com.icl.saxon.Stripper`). The Stripper is a good example of a SAX filter, a piece of code that takes a stream of SAX events as input and produces another stream of SAX events as output. At a more macroscopic level, an entire Saxon transformation can also be manipulated as a SAX filter. This approach makes it very easy to split up a complex transformation into a series of simple transformations arranged in a pipeline.

The **tree navigator**, as the name suggests, allows applications to select nodes from the tree by navigating through the hierarchy. The tree representation constructed by the builder component is proprietary to Saxon. This is an area where Saxon differs from some other XSLT processors: some of them use a general-purpose DOM model as their internal tree. The advantage of using the DOM is that the tree can then be produced by third-party software. Trees constructed for a different purpose can be supplied directly as input to a transformation, and equally, the output of a transformation can be used directly by DOM-based applications.

In Saxon I took the view that the interoperability offered by using the DOM comes at too high a cost. First, the DOM tree model differs subtly from the XPath model needed by an XSLT processor, and this difference imposes run-time costs in mapping one model to the other. For example, a DOM tree can contain information that the XPath model does not require, such as entity nodes. Second, DOM trees can be updated in place, whereas the XSLT processing model means that trees are written only sequentially. Designing a tree model that can be written only sequentially allows efficiencies to be achieved. For example, each node can contain a sequence number that makes it easy to sort nodes in their sequential document order, a frequent XSLT requirement. Finally, DOM implementations generally include a lot of synchronization code to make multithreaded access safe. Because the XSLT processing model is "write-once, read-many," the synchronization logic can be much simpler, leading to faster navigation of the tree.

Actually, as you will see, Saxon offers two different tree implementations, each with its own builder and navigation classes (packages `com.icl.saxon.tree` and `com.icl.saxon.tinytree`). The two implementations offer different performance trade-offs.

The **style sheet compiler** analyses the style sheet prior to execution. It does not produce executable code; it produces a decorated-tree representation of the style sheet in which all XPath expressions are validated and parsed, all cross-references are resolved, stack-frame slots are pre-allocated, and so on. The style sheet compiler thereby performs the important function of constructing a decision tree to use at execution time to find the right template rule to process each input node; it would be grossly inefficient to try matching each node against each possible pattern. The decorated tree then comes into play at transformation time to drive the style sheet processing. (The compiler is distributed across the classes in the `com.icl.saxon.style` package, especially the methods `prepareAttributes()`, `preprocess()`, and `validate()`).
At one stage Saxon did actually include a style sheet compiler that produced executable Java code. However, it handled only a subset of the XSLT language, and as the technology developed, the performance gains achieved by full compilation were dwindling. Eventually I abandoned that approach as the development complexity grew while the performance benefits declined. There is currently no full XSLT compiler on the market. Sun has produced an alpha release of a compiler called XSLTC which looks promising (see Related topics), though it is still at an early stage of development.

The decorated tree produced by Saxon's style sheet compiler (rooted at class com.icl.saxon.style.XSLStyleSheet) cannot be saved to disk, because reading the tree back into memory takes longer than recompiling the original (largely because of its increased size). You can reuse the tree so long as it remains in memory. The tree is wrapped in an object called the PreparedStyleSheet, which implements the javax.xml.transform.Templates interface in JAXP 1.1. It is quite common in a server environment to use the same style sheet repeatedly to transform many different source documents. To allow this, the compiled style sheet is strictly read-only at execution time, allowing it to be used in multiple execution threads simultaneously.

The core of the Saxon processor is the style sheet interpreter (class com.icl.saxon.Controller, which implements the javax.xml.transform.Transformer interface in JAXP 1.1). This interpreter uses the decorated style sheet tree to drive processing. Following the processing model of the language, it first locates the template rule for processing the root node of the input tree. Then it evaluates that template rule (or it's "instantiated," in the jargon of the standard).

The style sheet tree uses different Java classes to represent each XSL instruction type. For example, consider the instruction:

```xml
<xsl:if test="parent::section">
  <h3><xsl:value-of select="../@title"></h3>
</xsl:if>
```

The effect of this code fragment is to output an `<h3>` element to the result tree if the current node on the source tree has a parent element of type `<section>`; the text content of the generated `<h3>` node is the value of the title attribute of the parent `<section>`.

This code fragment is represented on the decorated style sheet tree by the structure shown in Figure 2.
Elements in the style sheet map directly to nodes on the tree, as shown in Table 1. All the Java objects that represent elements are subclasses of com.icl.saxon.style.StyleElement, which is a subclass of com.icl.saxon.tree.ElementImpl, the default implementation of an element node in the Saxon tree structure. The two XPath expressions are represented by com.icl.saxon.expr.Expression objects referenced from the nodes of the tree.

### Table 1. Style sheet elements and their corresponding Java classes

<table>
<thead>
<tr>
<th>Element or expression...</th>
<th>...represented by an object in this Java class (subclasses of com.icl.saxon.style.StyleElement)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsl:if&gt;</code></td>
<td>com.icl.saxon.style.XSLIf</td>
</tr>
<tr>
<td><code>&lt;h3&gt;</code> (output, not instruction)</td>
<td>com.icl.saxon.style.LiteralResultElement</td>
</tr>
<tr>
<td><code>&lt;xsl:value-of&gt;</code></td>
<td>com.icl.saxon.style.XSLOf</td>
</tr>
<tr>
<td>XPath expressions</td>
<td>com.icl.saxon.expr.Expression</td>
</tr>
</tbody>
</table>

Executing the `<xsl:if>` instruction causes the process() method of the corresponding XSLIf object to be executed. This method accesses the test Expression object, which has a method, evaluateAsBoolean(). evaluateAsBoolean() is used to evaluate the expression to return a Boolean result. (This is an optimization: it would be possible to use a straightforward evaluate() call and then convert the result to a Boolean, as described in the specification. But knowing that a Boolean is wanted often enables faster evaluation. For example, when the actual value or the expression is a node-set, the final Boolean result is known as soon as a single member of the node-set is found).
If the result of `evaluateAsBoolean()` is true, the `process()` method calls the `process()` method of all the child nodes of the `XSLIf` node on the style sheet tree. If the result not true, it simply exits.

Similarly, the `process()` method for a `LiteralResultElement` copies the element to the result tree and processes the children of the `LiteralResultElement`, while the `process()` method of the `XSLValueOf` object evaluates the select expression as a string, and copies the result as a text node to the result tree.

So the key components of the style sheet interpreter are:

- A class for each style sheet instruction type that contains the logic for that instruction
- A set of supporting classes to handle binding of variables, management of the run-time context, and matching of nodes against template rules
- The XPath expression interpreter to evaluate XPath expressions and return their values

The XPath interpreter (package `com.icl.saxon.expr`) closely follows the `Interpreter` design pattern, one of the 23 classic patterns for object-oriented software described by Gamma, Helm, Johnson, and Vlissides. Each construct in the XPath grammar has a corresponding Java class. For example, the `UnionExpr` construct (written as `A|B`, and representing the union of two node sets) is implemented by the class `com.icl.saxon.expr.UnionExpression`. The XPath expression parser (module `com.icl.saxon.expr.ExpressionParser`), which is normally executed when the style sheet is compiled, generates a data structure that directly reflects the parse tree of the expression. To evaluate the expression, each class in this structure has an `evaluate()` method, which is responsible for returning its value. In the case of the `UnionExpression` class, the `evaluate()` method evaluates the two operands, checks that the result is in both cases a node-set, and then forms the union using a sort-merge strategy.

As in the design pattern described by Gamma et al., the `evaluate()` method takes a `Context` parameter. The `Context` object encapsulates all contextual information needed to evaluate the expression.

This includes:

- Information about the current node and the current node list (needed, for example, to evaluate the XPath functions `position()` and `last()`)
- Access to the `com.icl.saxon.Bindery` object, where values of variables are held
- Access to the list of XML namespaces in scope for the expression, needed when testing the equivalence of names

The XPath interpreter also includes optimization features that extend the basic Interpreter pattern of Gamma et al.

For example:

- Each expression class has a `simplify()` method, to allow expression rewriting. This enables context-independent optimizations to be performed. Sometimes this results in transformation to a different XPath expression (for example `title[2=2]` is rewritten as `title`,...
while \(\text{count}(\$x) = 0\) is rewritten as \(\text{not}(\$x)\). More often the expression rewrite exploits internal classes that represent special cases. For example, the expression `section[@title]` returns all child `<section>`s of the current element that have a title attribute. Because of the context in which the sub-expression `@title` appears, it is possible to rewrite it using a special-purpose class that tests for the presence of a given attribute on the current node and returns a Boolean value.

- **Each expression class has an evaluate() method, and an enumerate() method.** This (in the case of expressions representing node sets) allows the nodes to be retrieved incrementally, rather than all at once. This allows pipelined execution, in the typical manner adopted by relational database systems. Calling `enumerate()` on a union expression, for example, works by merging the enumerations of its two operands. So long as the operands are both already sorted into document order, this avoids the need to allocate memory for the intermediate node sets.

- **Expressions can be progressively reduced, to eliminate their dependencies.** The concept of expression reduction is widely used in functional languages, and is particularly appropriate for optimizing a language such as XPath. Each expression class has a method `getDependencies()` which returns information about the aspects of the context that the method depends on. This already makes certain optimizations possible. For example, if the expression doesn't use the `last()` function then it is not necessary to do look-ahead processing to determine how many elements there are in the context list. Further, each expression has a method `reduceDependencies()`, which returns an equivalent expression in which specified dependencies are eliminated, while others are retained. This is useful where the same expression is used repeatedly. For example, before a sort is carried out, the sort key expression is reduced to eliminate dependencies on variables (because these will be the same for every node in the list) but not on the current node (which will be different for each item in the list).

The XSLT language gives the processor great freedom to evaluate expressions in any order it chooses, because of the absence of side effects. The general policy in Saxon is that scalar values (strings, numbers, Booleans) are evaluated as early as possible, while node-set values are evaluated as late as possible. Evaluating scalar values early enables optimization by doing things only once. Delaying the evaluation of node-set values saves memory, by avoiding holding large lists in memory unnecessarily. It can also save time, if it turns out (as it often does) that the only thing done with the node-set is to test whether it is empty, or to get the value of its first element.

Finally, the **outputter** component (class `com.icl.saxon.output.Outputter`) is used to control the output process. Saxon's result tree is not normally materialized in memory -- because its nodes are always written in document order, they can be serialized as soon as they are output to the result tree. In practice the transformation does not have a single result tree but a changing stack of result trees, because XSL instructions such as `<xsl:message>` and `<xsl:attribute>` effectively redirect the output to an internal destination, while `<xsl:variable>` constructs a result-tree fragment which is actually a separate tree in its own right. The interpreter code for these elements calls the outputter to switch to a new destination and subsequently to revert to the original destination.

External output is written to a file using a serializer. Logically this takes the result tree as input and turns it into a flat file document. In practice, as you have seen, the result tree is not materialized
in memory, so the serializer is handed the nodes of the tree one at a time in document order. This stream of nodes is presented using a SAX2-like interface (com.icl.saxon_EMITTER): it differs from SAX2 in the details of how names and namespaces are represented. As defined in the XSLT Recommendation, there are separate serializers for XML, HTML, and plain text output. Saxon also allows the tree to be supplied to user-written code for further processing, or to be fed as input to another style sheet. This allows you to achieve a multiphase transformation by applying several style sheets in sequence.

Performance

Good performance is necessarily a driving factor in the design of Saxon, second only to conformance with the XSLT specification. This is partly because it is critical to users, but also because in a world where there are several free XSLT processors available, performance will tend to be the main distinguishing feature.

This section discusses some of the factors that affect the performance of an XSLT processor, as well as the strategies Saxon uses to improve speed in each case.

Java language issues

It is often said that Java is slow. There is some justification in this, but the statement needs to be carefully qualified.

Many people imagine Java is slow because it generates interpreted bytecode rather than native code. This used to be true, but not any longer with today’s just-in-time compilers. Raw code execution speed is usually almost as good as -- sometimes better than -- the equivalent code written in a compiled language such as C.

Where Java can have a problem is with memory allocation. Unlike C and C++, Java takes care of memory itself, using a garbage collector to free unwanted objects. This brings great convenience to the programmer, but it is easy to create programs that are profligate with memory: they thrash due to excessive use of virtual memory, or they place great strain on the garbage collector due to the frequency with which objects are allocated and released.

Some coding techniques minimize the memory-allocation problems. For example the use of StringBuffer objects rather than String objects, use of pools of reusable objects, and so on. Diagnostic tools can help the programmer determine when to use those techniques. Getting the code fast does require a lot of tuning, but that is arguably still much easier than using a language such as C+++, in which you must manage all the memory allocation manually.

XSLT processing brings a particular challenge to Java in the implementation of the tree structure. Java imposes considerable red-tape overhead in the size of each object (up to 32 bytes, depending on the Java VM used). This often yields a tree structure in memory many times larger than the source XML file. For example, the empty element <a/> (four bytes in the source file) could expand to an Element object for the node, a Name object for its name, a String object referenced by the Name object, an empty AttributeList node, an empty NamespaceList node, plus numerous 64-bit object references to link these objects with each other and with the parent, sibling, and child
nodes in the tree. A nave implementation could easily generate 200 bytes of tree storage from these four bytes of source. Given that some users are trying to process XML documents whose raw size is 100MB or more, the consequences are predictable and generally fatal.

This is one reason Saxon went down the route of having its own tree structure. By removing the requirement to implement the full DOM interface, I was able to eliminate some data from the tree. Removing the requirement to support update is particularly useful. For example, Saxon uses a different class for elements that have no attributes, knowing that if an element has no attributes to start with, it will never acquire any later. Another technique Saxon uses is to optimize the storage of the common situation of an element that contains a single child text node, for example <b>text</b>.

The XPath tree model, as described in the W3C specification, includes nodes for attributes and namespaces. Because these nodes are rarely accessed in the course of a transformation, Saxon constructs these nodes on demand rather than having them permanently take up space on the tree. (This is the Flyweight design pattern of Gamma et al.)

The latest release of Saxon has gone one step further: using a tree implementation in which the nodes are not represented by Java objects at all. Instead, all the information in the tree is represented as arrays of integers. All nodes are created as transient (or flyweight) objects, constructed on demand as references into these arrays and discarded when they are no longer needed. This tree implementation (package com.icl.saxon.tinytree) takes up far less memory and is quicker to build, at the cost of slightly slower tree navigation. On balance, it appears to perform better than the standard tree, and I therefore provide it as the default.

The standard utility classes such as Hashtable and Vector also affect Java program performance. Developers find it tempting to use these convenient classes liberally throughout an application. However, there is a price to pay. Partly because the classes usually do more than you actually need, they impose more overhead than a class designed for one purpose only. They are also designed to handle a worst-case situation in terms of multithreading. If you know that a data structure will not be accessed by multiple threads simultaneously, you can spare yourself the synchronization costs by designing your own objects rather than using these off-the-shelf classes. Replacing vectors by arrays often pays dividends, the only downside being that you need to handle expansion of the array manually whereas vectors are self managing.

**Location path evaluation**

The most characteristic kind of XPath expression (the one from which XPath gets its name) is the location path. Location paths are used to select nodes in the source tree. A location path essentially consists of an origin and a number of steps. It is similar to a UNIX filename or a URL, except that each step selects a set of nodes rather than a single node. For example ./chapter/section selects all the &lt;section&gt; children of all the &lt;chapter&gt; children of the current node. The origin identifies the start point for navigating the tree: it might be the current node, the root node of the source tree, the root node of a different tree, or a set of nodes located by value using a key. Each step in the location path navigates from one node to a related set of nodes. Each step is defined in terms of a navigation axis (the child axis being the default): For example the ancestor
axis selects all ancestor nodes, the following-sibling axis selects all following siblings of the origin node, the child axis selects its children. As well as specifying an axis, each step may specify the type of node required (such as elements, attributes, or text nodes), the name of the required nodes, and predicates that the nodes must satisfy (for example, child text nodes whose value begins with B).

Devising an execution strategy for a location path is equivalent to the problem of optimizing a relational query, though the theory is currently much less advanced, and most of the techniques used differ little from the nave strategy of doing the navigation exactly the way it is described in the specification. For example, although it is possible in a style sheet to specify keys that must be built to support associative access (rather like the CREATE INDEX statement in SQL), Saxon currently uses these indexes only to support queries that reference them explicitly (by using the key() function) and never to optimize a query that uses straightforward predicates.

The optimization techniques currently used in Saxon for location paths include:

- **Avoiding a sort wherever possible.** Many XSLT instructions require nodes to be processed in document order, so some effort is made to retrieve nodes in document order, and even more effort to detect when the natural order in which nodes are retrieved is either in document order or reverse document order, thus obviating the need for a sort. An example of this is that the expression //item (which is defined to be an abbreviation for /descendant-or-self::node()/item) can be replaced by /descendent::item provided it uses no positional predicates. The latter expression will naturally retrieve nodes in document order, whereas the former might not.

- **Reduction of predicates.** This can sometimes cause predicates to reduce to the constant values true or false, allowing the entire location path to be simplified. More often it simply has the effect of removing a common subexpression. For example in the filter expression $x[count(.|$y)=count($y)] (which is the only convenient way in XSLT 1.0 of doing a set intersection operation), Saxon will evaluate count($y) only once.

- **Early termination with positional predicates.** A predicate such as para[position() <= 3] selects the first three <para> children of the current node. It is not necessary to apply this predicate explicitly to every <para> element to see if it is true, since processing can stop after the third node.

- **Optimization of attribute references.** The XPath model treats attributes in very much the same way as child elements, which greatly simplifies the XPath language. However, because an element may have at most one attribute with a given name, access to attributes can be optimized. This optimization also takes account of the fact that attribute nodes are not actually materialized on the tree unless they are required. This means that while the XPath expression child::title scans all the child elements looking for those whose name is title, the similar expression attribute::title (usually abbreviated to @title) gets the relevant attribute directly.

- **Lazy evaluation of location paths.** Evaluating a location path expression in a particular context does not return an actual list of nodes in memory, rather it returns another expression (referred to as an "intensional node-set," class com.icl.sax.expr.NodeSetIntent) in which all the context dependencies have been removed. It is only when the intensional node-set is
actually used that its members are enumerated: and depending on how it is used, they may not need to be retrieved at all. For example if the node-set is used in a Boolean context, the only processing needed is to test whether it is empty. When an intensional node-set is used for the third time, it is stored extensionally, trading memory for processing time. This is like materializing a view in SQL.

**Style sheet compilation**

I have already described how the first thing Saxon does is to "compile" the style sheet into a decorated tree for efficient execution subsequently. This offers a great opportunity to do things once only rather than doing them each time the relevant instructions are executed.

Some of the tasks done during the style sheet compilation phase are as follows:

- **Validation.** The vast majority of user errors can be detected during the compilation phase. This includes some errors that at first sight would appear to be run-time errors. XPath expressions use dynamic typing (the type of an expression or of a variable is not necessarily known until the expression or variable is evaluated). However, for the vast majority of actual XPath expressions, the type is known at compile time. So, for example, `<xsl:for-each select="$x+2">` can be instantly recognized as an error because the XPath expression `$x+2` can never return a node-set. In many cases it is even possible to detect that `<xsl:for-each select="$x">` is an error, because the absence of assignment statements means that the type of the variable can often be inferred from its declaration.

- **Simplification of expressions.** Some of the techniques used have already been discussed. One important context in which expressions are used is in attribute value templates. For example, the literal result element `<td width="{$x * 2}">` outputs a `<td>` element whose `width` attribute is computed at run time. An important compilation task is to convert attribute value templates into an efficient structure for evaluation at run time.

- **Binding of variables and other names.** Because all variable declarations are visible at compile time, it is possible for the compiler to allocate slots on the stack frame for each called template rule in advance. References to variables within an expression can then be statically bound to a particular slot in either the local stack frame or the list of global variables. Similarly, other references to named objects such as templates and external functions can often be resolved statically. In some cases the XPath syntax allows a name to be generated dynamically (for example, key names or names of decimal formats), but it is still possible to detect the common case where the name is provided as a literal and then bind it statically.

In other cases, doing things at compile time is less feasible, but savings can be made by avoiding repeated execution at run time. An example is the `format-number()` function, which takes as one of its arguments a pattern describing the output format required for a decimal number. Considerable savings are possible by detecting the common case where the format pattern is the same as on the previous execution. The only tricky aspect of such optimizations is to keep the memory of previous executions in a place associated with the current thread: it cannot be kept on the style sheet tree itself, as that needs to be thread safe.
Pattern matching for template rules

The pattern matching operation is potentially expensive, so it is vital to focus the search intelligently. The style sheet compiler therefore constructs a decision tree which is used at run time to decide which template rule to apply to a given node.

I'm using the term decision tree here loosely. This section describes the actual data structures and algorithms in a little more detail. (See modules com.icl.saxon.RuleManager and com.icl.saxon.Mode in the source code.)

When the <xsl:apply-templates/> instruction is applied to a node, a template rule must be selected for that node. If there is no matching rule, Saxon uses a built-in rule. If there is more than one rule, the processor resorts to an algorithm from the XSLT specification for deciding which rule takes precedence. This algorithm is based on user-allocated priorities, system-allocated priorities, the precedence relationships established when one style sheet imports another, and -- if all else fails -- the relative order of rules in the source style sheet.

In a typical style sheet, most template rules match element nodes in the tree. Rules to match other nodes, such as text nodes, attributes, and comments, are comparatively rare. Also, most template rules supply the name of the element they must match in full. Rules for unnamed nodes are allowed but not often used (for example, *[string-length(.) > 10], which matches any element with more than 10 characters of text content).

Saxon's strategy is therefore to separate rules into two kinds: specific rules, where the node type and name are explicitly specified in the pattern, and general rules, where they aren't. The data structure for the decision tree contains a hash table for the specific rules, keyed on the node type and node name, in which each entry is a list of rules sorted by decreasing priority; plus a single list for all the general rules, again in priority order. To find the pattern for a particular node, the processor makes two searches: one for the highest-priority specific rule for the relevant node type and name that matches the node, and one for the highest-priority general rule that matches the node. Whichever of these has highest priority is then chosen.

For a multipart pattern such as chapter/title, the algorithm used is recursive: The match is true if the node being tested matches title and if its parent node matches chapter (module com.icl.saxon.pattern.LocationPathPattern). This simple approach can't be used for patterns that use positional predicates; for example chapter/para[last()], which only matches a para element if it is the last one in a chapter. Matching these positional patterns is potentially very expensive, so it's worth handling the common case of a pattern like para[1] specially.

Numbering

Numbering the nodes on the tree (using the <xsl:number/> instruction) poses a particular optimization challenge. This is because each execution of <xsl:number/> works independently to assign a number to the current node, the number being defined by a complex algorithm using various attributes on the <xsl:number/> instruction. Nothing inherent in the algorithm says that if the last node was numbered 19, the next one will be numbered 20, yet in most common cases
that is indeed the case. It is important to detect those common sequential cases. Otherwise the numbering of a large node-set will have $O(n^2)$ performance, which is what happens if the numbering algorithm as specified in the XSLT Recommendation is applied to each node independently.

Saxon achieves this optimization for a small number of common cases, where most of the attributes to the numbering algorithm are defaulted. Specifically, it remembers the most recent result of an `<xsl:number/>` instruction, and if certain rather complex but frequently satisfied conditions are true, it knows that it can number a node by adding one to this remembered number.

**Finally**

I have tried in this article to give an overview of the internals of the Saxon XSLT processor, and in particular of some of the techniques it uses to improve the speed of transformation. In the 18 months or so since I released the first early versions of Saxon, performance has improved by a factor of 20 (or more, in the case of runs that were thrashing for lack of memory).

It's unlikely that the next 18 months will see a similar improvement. However, there is still plenty of scope, especially for constructs like `<xsl:number/>`. To take another example, Saxon has not even started to explore the possibilities opened by parallel execution, something the language makes a highly attractive option.

Perhaps the biggest research challenge is to write an XSLT processor that can operate without building the source tree in memory. Many people would welcome such a development, but it certainly isn't an easy thing to do.

**Postscript**

**Michael Kay writes in April 2005:** Although this article was written over four years ago, it has stood the test of time. Underneath the surface, the technology has become a lot more sophisticated (for example, much more optimization is now done at compile time), but the high-level architecture of Saxon is still much as described here.
Related topics

- See What kind of language is XSLT? by the same author published as a companion to this article.
- The XSLT 1.0 Recommendation, published by the World Wide Web consortium. The definitive specification of the XSLT language.
- The XPath 1.0 Recommendation, published by the World Wide Web consortium. The definitive specification of the XPath expression syntax used within an XSLT style sheet.
- The Document Object Model (DOM) provides a standard set of objects for representing HTML and XML documents, and a standard interface for accessing and manipulating them.
- The Simple API for XML, a standard interface for processing the contents of an XML document as a stream of parser events.
- The home page for Saxon, the open-source XSLT processor described in this article.
- The SUN XSLTC Compiler, in 2001, was an alpha release. Now, try Apache's Getting Started with XSLTC.
- The Transformation API for XML (TrAX) is defined as part of JAXP 1.1, which itself was published through the Java Community Process in JSR-63.
- Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, and Vlissides. Addison-Wesley, 1998. ISBN 0-201-63361-2. This excellent book isn't available online, but you can buy it at Amazon.com.

© Copyright IBM Corporation 2001, 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)