Copyright © 2008 IEEE. Reprinted from
International Symposium on Cluster Computing and the Grid
(6th : 2006: Singapore)

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Adelaide's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

Compilation of XSLT into Dataflow Graphs for Web Service Composition

Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn
School of Computer Science
University of Adelaide
South Australia 5005, Australia
{pmk,paulc,andrew}@cs.adelaide.edu.au

Abstract

Our current research into programming models for parallel web services composition is targeted at providing mechanisms for obtaining higher throughput for large scale compute and data intensive programs that delegate part of their computation to services, and making it easier to develop such applications. The ability to invoke multiple service calls at one time on different machines enables different portions of the program to be executed concurrently. We are addressing this through an implementation of an existing functional language, XSLT. Our implementation uses a dataflow execution model, and includes a compiler to build dataflow graphs from XSLT source code. This paper describes the execution model used to obtain parallelism and compose web services, as well as the compilation process used to create the dataflow graphs. Our aim with this paper is to present the design of our system and demonstrate that XSLT provides a suitable model for distributed execution and parallel composition of web services.

1 Introduction

The recent popularity of grid computing and web services has seen the emergence of systems for web service *composition*, also known as *orchestration*. This is an approach to writing high-level programs which integrate the functionality provided by different services to develop additional features on top of what any of these services can provide by themselves. It is similar to the idea of shared libraries in traditional programming languages, where developers make use of API calls provided by libraries. Applications that compose different web services together can be arbitrarily complex. However, many languages promoted for this purpose to date provide only limited control constructs, making it hard to include complex application logic in the composition itself.

We are developing a programming environment for creating service composition programs which aims to provide a more powerful programming model. To do this, we are implementing XSLT, a functional language designed for processing XML data. In order to support large scale applications with complex and demanding compute and data requirements, we are targeting our implementation at parallel execution. This is important for many applications which require multiple remote web service operations to be in progress at a time, so that the work of the application can be divided up between machines in a grid to increase throughput. Examples of such applications include those used in the areas of engineering simulations, scientific experiments, and financial analysis, all of which deal with large amounts of data.

As a functional language, XSLT is a desirable choice for developing parallel applications, because the parallelism can be determined implicitly by the compiler. The lack of side effects means that out-of-order evaluation is possible, and it is thus possible for multiple parts of a program to be executing concurrently. The automatic parallelisation of code that we describe in this paper relieves the programmer from having to manually deal with things like threads, critical sections, and message passing. Parallel programs written in the functional style can thus be much simpler than those written in imperative languages as these details are abstracted away by the underlying language implementation.

Our parallel execution model is based on the concept of *dataflow computation* [7], in which a program is represented as a directed graph, with nodes representing *computational entities*, and the edges between them indicating the flow of data. A computational entity can be any type of operation which consumes data, performs some computation on it, and produces a result. In our model, this includes both built-in and user-defined functions, as well as web service calls. A dataflow program, in our model, can consist of operations that are performed by the language implementation itself, such as those used for arithmetic and string manipulation, or operations that are provided by web services on re-

0-7695-2585-7/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

mote machines. The integration between these two types of operations results in a programming and execution environment that, from the programmer's point of view, provides a seamless way of dealing with local and remote processing.

2 Related work

Some approaches have been taken to date towards improving performance of XSLT programs through parallelism. The work described in [2] achieves this by executing multiple copies of a program sequentially; however this approach only benefits cases where there are many instances of a small program being run, not one large program. A commercial parallel XSLT processor is available from Conformative Systems [4], although no technical details on the implementation are available. Work has been done on parallelising XQuery [11], a language that has a lot in common with XSLT.

Applications of the dataflow execution model to web service composition include Triana [12] and Taverna [14]. Both of these use the model to enable multiple web service operations to be executed in parallel. However, they require the programmer to work directly with the dataflow representation, which for large programs can become cumbersome to work with. BPEL (Business Process Execution Language) [5] is an imperative language designed for web service composition which also supports parallelism. However, the parallel sections of the program must be explicitly specified, and if the code is written incorrectly, it is easy to introduce race conditions and non-deterministic behaviour.

A common approach used in dataflow systems is to compile a higher level language into the dataflow representation, which avoids the need for the programmer to explicitly specify the parallelism. In contrast to imperative languages, this translation is more readily achievable for functional languages, which are side-effect free, and thus do not depend on global state. Languages for which dataflow compilers have been developed include APL [13], V [10], ALFL [6], and Lazy ML [1].

3 Overview of XSLT

Extensible Stylesheet Language Transformations (XSLT) [18] is a language specifically designed for processing XML data. All variables, as well as function parameters and return types, are defined in terms of XML Schema [15]. Simple types consist of the common types present in other languages such as integers, floats and strings, while data structures are represented as trees of XML elements. It is a pure functional language, restricted to consuming input data and producing output data, and there is no notion of global state. All functions are side-effect free, and variables are single-assignment only.

The core language concept in XSLT is that of a *sequence constructor*, a group of statements which are evaluated to produce a list, or *sequence*, of values. Each *item* in a sequence may be either an *atomic value* or a *node* in an XML tree. Each statement in the sequence constructor produces zero or more sequence items, and the sequences produced by each statement are concatenated together to form the result of the sequence constructor. Some types of statements such as loops and conditionals may themselves contain other sequence constructors, in a similar manner to the concept of basic blocks found in imperative languages.

Many statements take a parameter specified as an XPath expression [16], which runs a query over a tree of nodes and produces a sequence of items. The result of this expression affects the way in which the statement is evaluated, such as determining which branch of a conditional statement is evaluated, or the sequence over which loop iteration occurs. For example, the path expression `department/employee[@role='Manager']` returns all `employee` elements that have a role attribute of "Manager".

During execution of a program there is the concept of *dynamic context*. This is a set of implicitly defined variables which are related to the current input data being processed. It includes the *context item*, which is a value that refers to the item for which the current block of code, such as a loop body or template, was invoked. The position of the context item and the size of the sequence in which that item resides also forms part of the dynamic context. Certain statements cause the dynamic context to temporarily change - for example, when a loop is evaluated, in which case the context item is set to the current item in the input sequence.

There are two ways of modularising code in an XSLT program: *functions* and *templates*. Functions in XSLT are just like those in other languages; each has a name and a set of parameters, and returns a result. The function body consists of a sequence constructor that can contain any set of statements. All functions are side-effect free; there is no global state that they can modify, and a function can only compute a result based on the values passed in as parameters. This means that a function is guaranteed to return the same result each time it is called with a particular set of parameters. This is a key feature of the language which enables out-of-order evaluation and parallelism.

Templates are similar to functions, except the way they are invoked is different. They effectively act as tree transformation rules, mapping a pattern to a set of statements to be evaluated when that pattern is found. Execution of an XSLT program begins by taking an input XML document, and applying the set of templates to the elements that are encountered. Template rules typically match different portions of the document based on element names, but more complex patterns are also possible. When a pattern matches, the tem-

IEEE COMPUTER SOCIETY

plate is called, and the result of the evaluation is combined with the results of other templates to form the output of the program.

Our implementation of XSLT is targeted at web service composition. Our decision to use XSLT for this purpose was based on two key features of the language: the type system, and the functional semantics. The use of XML schema for defining all types of data handled by a program provides a good match with WSDL, the language used for defining web service interfaces [9]. The functional nature of the language also makes automatic parallelisation easy, which for imperative languages is difficult to achieve [3]. We support web service composition by extending the language with additional mechanisms to associate certain function calls with web service operations; the programmer can then invoke web service operations using the standard function call mechanisms in a transparent manner.

One implication of our model is that we also assume the web services that are composed are stateless; however, for the types of applications we are targeting with our system, this does not impose any serious constraints. We are primarily targeting applications that involve processing of raw data to produce results, such as those used in science and engineering for performing simulations and analysing data collected from experiments.

4 Dataflow model

Our execution model is based on dynamically unfolding dataflow graphs. The program is first compiled into a set of static graphs, each of which corresponds to a block of code within the program, such as a function, template, or loop body. At runtime, a new copy of the graph is instantiated whenever that code block is to be executed. When a graph is instantiated, an *activity* is created for every operation in the graph. Each activity has pointers to the destination activities to which the output values should be sent, in accordance with the structure of the static graph. The activities are placed in a pending set, and when each receives the appropriate number of input tokens, it fires. Upon firing, the operation is performed, the output values are produced, and the activity is removed from the set. The output values get transmitted to the input ports of the activities that are connected to the one that just fired.

A simple block of code with no loops or function calls only requires one graph instantiation. As each activity is fired, subsequent activities become enabled, and fire, producing their output values which flow through the graph. Loops are handled by creating a new copy of the graph corresponding to the loop body for each item in the list of values to be iterated over. Function calls are handled by creating a new copy of the graph built from the function body; this allows recursion to be supported. Parallelism is achieved by executing different loop iterations and function calls on separate machines. A *scheduler* is responsible for deciding which activities should be assigned to which machines, based on CPU load and other information; the details of this are outside the scope of this paper.

The static graphs from which the activities are instantiated are built by the compiler from the XSLT source. Each operation in a graph has one or more input ports, and one or more output ports. We use a *data driven*, or *eager* evaluation strategy; the availability of input data causes output data to be produced.

The compilation process, described in Section 5, parses the source code and builds the static graphs from which the runtime activities are instantiated. A program is passed to the dataflow execution engine in the form of a set of static graphs, with one marked as the *initial* graph. This is like the `main` function in C; it is instantiated first, and from there it makes calls to other functions which get activated as described above.

4.1 Data tokens

Each input port and output port of an operation is assigned a type. The set of types that can be assigned to a port is defined by the XPath data model [17]. These include atomic types, such as integers or strings, as well as complex data structures represented as trees of XML elements. Each type may also have an occurrence indicator associated with it, which indicates the number of values that can be contained within a sequence that matches the type. If present, this is either `*` (zero or more), `+` (one or more), or `?` (zero or one).

A data token may represent either a single item or a sequence of items. Sequences are handled like lists in Lisp; a token can represent a cons-like pair with *left* and *right* pointers to other tokens. Each of these tokens may be a single item, or another pair. Sequences can thus be represented as a tree, with the branches corresponding to pairs, and the leaves corresponding to the actual values in the sequence. Unlike Lisp, the XPath data model does not allow nested lists; a tree of pair tokens is treated as a flat sequence of values.

When a token is transmitted from the output port of one operation to the input port of another operation, this token may represent either a single item or a (possibly empty) sequence. Whether or not it is possible to produce or consume a sequence of items is dependent upon the nature of the operation. Some expect exactly one item to be present in the token, while others allow a different range. This is determined by the occurrence indicator of the type associated with the port. These type associations are checked during compilation where possible, and in other cases at runtime.

Tokens are immutable; once created, they cannot be

Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)
0-7695-2585-7/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

modified. Instead of changing a token, it is necessary to create a new one, which may contain copies of the parts of the old token that don't need to be changed. Due to the way we implement reference sharing among token copies, this is necessary to ensure that the semantics of the execution model allow for deterministic parallel evaluation, which is why each token is treated as a fixed value.

4.2 Operators

Most operations in our dataflow model follow the simple convention that in order to fire, they must receive a token on all of their input ports, and once they have completed, produce a token on a single output port. These operations are implemented internally as C++ functions, which take the set of input values as parameters, and return the value to be sent on the output port. There are, however, a number of special operators which have different semantics. These need to be handled specially by the interpreter. Most of these are standard dataflow operators used for control purposes, with a few that are specific to our implementation. These special operators are as follows:

- **DUP** duplicates an input token. It has one input port on which to receive the token, and two output ports upon which copies of the token are produced.
- **CONSTANT** produces a specific value on its single output port; this value is assigned statically to the operation during compilation. It has a single input port on which it receives a value to trigger firing of the operation; this value is ignored.
- **SPLIT** enables conditional control flow. It has two input ports, one of which accepts a data value of any type, and the other which accepts a boolean value used for control. Upon firing, the data token is produced on only one of the two output ports; the choice of which port to use is determined by the control value.
- **MERGE** takes the first value that arrives on either of its two input ports, and passes it along on its single output port. It is used in combination with SPLIT to handle conditionals.
- **CALL** dynamically instantiates a set of activities for a dataflow graph corresponding to a specific function, the name of which is assigned to the operation during compilation. The CALL operator has one input port for each parameter to the function, plus an additional input for a token representing the dynamic context structure. The single output port from the CALL is connected to another operation in the graph which will receive the result of the function. At runtime, the activity corresponding to the RETURN operation in the function body is connected directly to the destination of the CALL. Both user-defined functions and templates are handled in this manner.
- **MAP** is similar to the CALL operation, except that if the input token is a sequence of items, then the function is instantiated once for each item. At runtime, a set of SEQUENCE activities are dynamically created and connected together such that they take input from the RETURN activity of each function instantiation and produce a list of result values that is in the same order as the input sequence passed to MAP. This result sequence is passed to the operation to which the output port of the MAP node is connected. MAP can be used to invoke subgraphs compiled for loop bodies, path expressions, and filter predicates, which are essentially treated the same as functions from the perspective of the dataflow interpreter.
- **SEQUENCE** takes as input *left* and *right* tokens and produces a pair token pointing to these. A set of SEQUENCE operations connected in series can be used to produce a list of items from individual values.
- **PASS** has one input port and output port, and simply passes the input token along verbatim.
- **SWALLOW** has one input port and no output ports. It just consumes the value passed to it. Both PASS and SWALLOW are generally used to simplify certain steps of the compilation and provide no real useful function at runtime. They can be optimised away in most cases.
- **RETURN** provides identical functionality to the PASS operation, except it is used differently at runtime. When a CALL operation instantiates a function graph, it connects the instantiated RETURN activity to the activity corresponding to the destination operation that was connected to the CALL in the compiled graph.
- **CTXITEM** is used to extract the context item from the dynamic context information. The input token to this operation is expected to be a collection of information about the dynamic context, and the output is the context item component of this collection.
- **WSCALL** invokes a web service operation. The input messages to the service are taken from the tokens received on the input ports. The request is sent asynchronously, and while the operation is being performed on the remote host, other operations can continue to execute locally, as long as they have no data dependencies on the result of the service operation. This allows the possibility of multiple service calls to be in progress at the same time. Upon receiving the response

IEEE COMPUTER SOCIETY

from the service, the result is transmitted on the output port.

4.3 Support for distributed execution

Our dataflow model supports distributed execution of programs in two ways: through activity distribution, and web service invocation. As mentioned above, whenever a graph of activities is instantiated at runtime, it is possible for those activities to be assigned to different machines on a network. Loop bodies, function calls, and independent portions of a block of code can thus be executed independently, as long as there are no data dependencies between them. By running an instance of the execution engine on each machine in a grid, execution of the program can be parallelised.

Web service invocation allows functionality provided by remote services to be incorporated into the program. From the perspective of the dataflow model, a call to a web service is treated as a special type of activity, which gets handled by submitting a SOAP request to the service. The details of how that service is implemented are transparent to the dataflow interpreter; the remote machine executes whatever code is sitting behind the service interface and returns a result. Parallelism is obtained by making multiple web service calls in parallel, each of which may potentially be provided by a different machine. Multiple outstanding web service calls can be in progress at any given point in time during execution. Other parts of the program which do not depend on the results of a given operation can continue execution while the call is in progress, and once the call completes, those activities which do have a data dependency on the web service call operation can be fired.

Our current implementation of this model only supports sequential execution of a graph on a single machine, although parallel web service invocation can occur through the use of asynchronous messaging. We have chosen this model specifically with distributed execution in mind as we intend to implement this in the next stage of our development. The purpose of this paper is to present an overview of the execution environment and compilation process in our system; a detailed evaluation and performance analysis will be provided once we have the distributed implementation completed. We also intend to investigate a variety of scheduling strategies for distributed web service composition, an area which has been given little attention to date in the research community.

5 Compilation process

Initially, the compiler parses the source code for an XSLT program and produces a syntax tree. This tree includes XPath expressions that are specified on attributes of various XSLT statements. Internally, the compiler treats these as one language rather than two, and both XSLT sequence constructors and XPath expressions are represent in a similar manner. The parser also supports an alternative syntax we have developed [8] which represents these constructs in a more consistent manner than the traditional XSLT/XPath combination. The compiler recursively processes this tree, creating a new graph for each function, and then building up portions of the graph for each node in the tree.

Initially, when a function graph is created, there is an input node and a RETURN node. As the tree is processed, additional nodes are added between these two, so that at the end, the output of the last statement in the function points to the RETURN operator.

The input node takes a token representing the dynamic context, which is a structure containing information such as the current item, the length of the input sequence, and the position of the current item within the sequence. Some language constructs, such as the current item expression (`.`), and path expressions, require the current item to be extracted from this structure, which is achieved by adding a CTXITEM operator to the graph. Other operators depend on the dynamic context for other purposes, such as the `position()` function, which returns the position of the current item in the input sequence. The dynamic context is passed as input to these directly without the use of the CTXITEM operation. Whenever a template is invoked, the a new dynamic context is created representing the node that the template applies to.

Because of the use of a data driven evaluation strategy, all nodes must have at least one input, so that it is possible for them to be fired. Thus, even if an operation does not require inputs, such as a constant, or a function with no parameters, it is necessary to create an input port for that node and connect it to another node, in order to control when it is fired. In this case the input value is just consumed and silently ignored; it does not matter what value is passed in as it is essentially just a control dependency.

The following sections describe how each of the main XSLT and XPath language constructs are compiled into graphs. Due to space restrictions we only give simple examples for each. A more complex example, which demonstrates the the parallel composition of web services using this model, is given in [8].

5.1 Constants

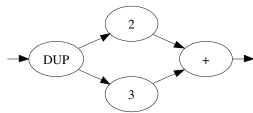
These are the most straightforward to compile. A single node is inserted into the graph representing the CONSTANT operation, and the numerical or string value of the constant is set on the node. An input edge is added to connect this node into the graph so that it will fire upon receipt

IEEE COMPUTER SOCIETY

of a token, however at runtime the value of this input token is ignored and the constant is sent on the output edge.

5.2 Binary operators

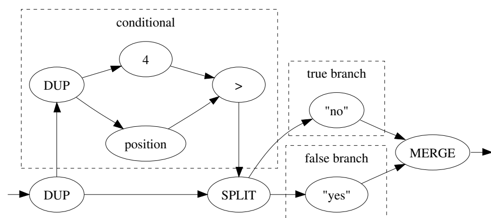
Operators such as +, −, and, or, = and < have subgraphs compiled for both the left and right expressions. A DUP operator is added, which feeds into each of the subgraphs. The output of each subgraph is connected to the binary operator. The graph compiled for the expression `2 + 3` would look like this:



5.3 Conditionals

There are three types of conditionals: XSLT's `if` and `choose` statements, and XPath's `if` expressions. All have the same semantics, with the exception that the XSLT `if` statement does not allow an else branch. Conditionals are handled using a pair of SPLIT and MERGE operations. A DUP is added at the start, one output of which gets fed directly into the first input port of the SPLIT, to get passed on as the input to whichever branch is to be evaluated. The second output of the DUP is fed into the subgraph which is compiled to evaluate the conditional. The subgraphs compiled for the true and false branches each take input from one of the SPLIT's output ports; at runtime, the port on which the SPLIT outputs a token will determine which of these branches gets activated. In the case of multiple branches, false branch is just treated as another conditional and compiled recursively.

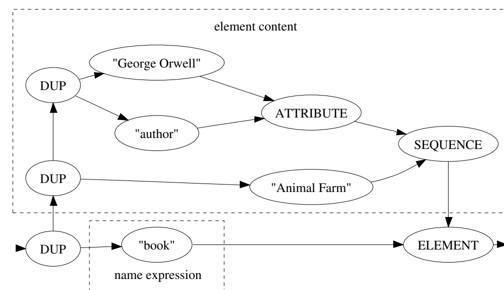
```
<choose>
  <when test="position() > 4">
    yes
  </when>
  <otherwise>
    no
  </otherwise>
</choose>
```



5.4 Element constructors

An element creation statement, specified either by a literal result element, or by using an expression such as `<element name=''...''>`, is created using the ELEMENT operation. This has two input ports; the first receives the name of the element as a string, and the other takes a sequence of items representing the child elements and text nodes, as well as attributes to be added to the element. For each of these two ports, a subgraph is compiled corresponding to child statements of the element constructor and the specified element name expression. In the case of literal elements, the name is just a string constant, however the `<element>` statement allows the name to be computed from an expression. The example below specifies a literal result element with one attribute and containing a single text node. The SEQUENCE operator in the graph constructs a list containing the attribute and text node in a manner similar to Lisp's `cons` operator, and outputs a single data token representing this list.

```
<book author="George Orwell">
  Animal Farm
</book>
```



5.5 Loops

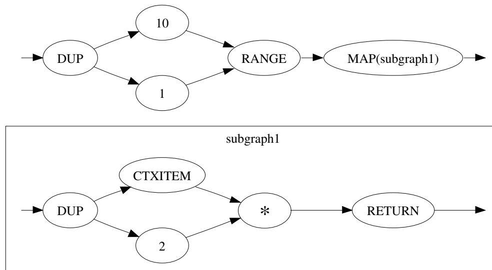
XSLT's `for-each` statement and XPath's `for` construct need to be handled specially. These can cause the contained block of code to be executed an arbitrary number of times. Because our execution model is based on activities that fire once, and parallelism is achieved by having separate activities dynamically instantiated and assigned to different machines, it is necessary to compile the loop body into a separate graph. A MAP operator is then used to take the sequence of values to be processed and instantiate a copy of the graph for each one.

The `select` attribute of the `<for-each>` statement, or its equivalent bracketed expression in an XPath `for` construct, is compiled into a subgraph which produces a sequence of values to be passed into the MAP operator. In this example, the built-in RANGE operation is used to create a sequence of values from 1 to 10, and the subgraph compiled for the loop body doubles each of these. Thus,

0-7695-2585-7/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

the output of the MAP operator is a data token representing the sequence of values from 2 to 20, going up in increments of 2. The context item (`.`) corresponds to the value passed in to the current loop iteration, and must be extracted from the dynamic context value using the CTXITEM operator in the subgraph.

```
<for-each select="1 to 10">
  <sequence select="2 * ."/>
</for-each>
```

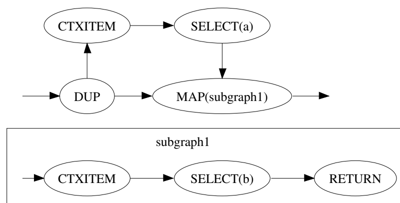


5.6 Path expressions

Expressions of the form `a/b` are compiled using the SELECT operation. The input to this operation is assumed to be a sequence of XML nodes, extracted from the dynamic context using the CTXITEM operation. The SELECT operation takes this sequence, checks each of the nodes against the node test, and outputs a sequence containing the matching nodes.

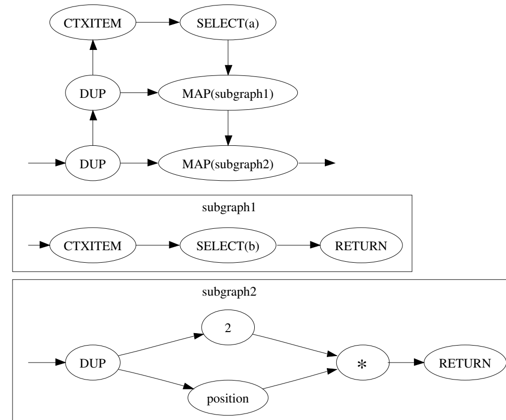
In path expressions with multiple steps, each step is evaluated once for every item in the sequence returned by the previous step. The expression *expr1*/*expr2* would cause *expr2* to be evaluated once for each item returned by *expr1*, in a similar manner to a loop. These types of expressions are compiled by building a separate graph from the latter expression, and using a MAP operation to invoke this graph for each element in the earlier expression. The following example demonstrates this:

```
<sequence select=''a/b''>
```



As a more complex example, consider a path expression where the last step performs some computation. In this case, the position of the context item is extracted and multiplied by two. This results in subgraphs being constructed for the second and third steps. The path evaluation process involves processing each of these steps in turn; this is handled by invoking the appropriate subgraph for each item passed into the two steps.

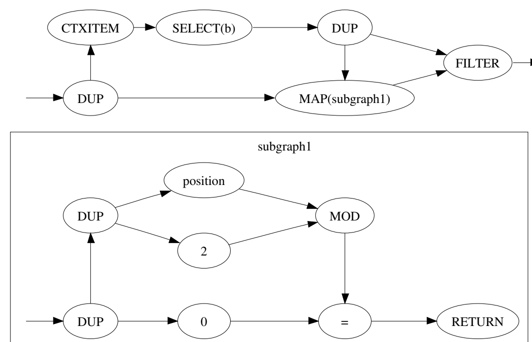
```
<sequence select="a/b/(position()*2)"/>
```



5.7 Filter expressions

Expressions of the form `a[b]` are handled using the FILTER operation. This operates in a similar manner to path expressions; however, with a filter expression, the result of the predicate is used to determine which elements of the sequence produced from the preceding expression are included in the result. As the predicate expression can be arbitrarily complex, and must be executed once for each item in the sequence, it is compiled into a separate graph in the same manner as for path expressions. A FILTER operation is added after the MAP; the output sequence of boolean values from the MAP operation is passed into the first input port of the FILTER, and a copy of the original sequence, obtained by an additional DUP, is passed into the second input port. The result of the FILTER operation is to select those items in the original sequence for which the corresponding predicate evaluates to true.

```
<sequence select="b[position() mod 2=0]"/>
```



Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)
0-7695-2585-7/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

5.8 Template application

The implicit template application that occurs at the start of the program or the explicit application via the `<apply-templates>` statement is treated as an if statement containing multiple branches, each of which tests for a specific template rule. Template rules are tested in priority order, so that those which have have been assigned a higher priority are checked first, and only if they do not match will the lower priority rules be used.

5.9 Function calls

A call is handled by adding a single node to the graph which represents the function call. This is either a CALL or WSCALL operator for user-defined functions and web service operations, as described in Section 4, or an internal operator corresponding to one of the built-in functions defined by the language specification.

A subgraph is compiled for each expression passed as a parameter to the function. For functions that take a single parameter, the output port is connected directly to the input for the subgraph; for multiple parameters, the appropriate number of DUP nodes is created, and the output ports of these are fed in to the parameter subgraphs. The function call node has one input port for each parameter, and the output port of each subgraph is connected to the corresponding input port. For built-in functions which need the dynamic context, this is treated as an additional parameter to the function.

5.10 Type conversion

Both built-in and user-defined functions in XSLT, as well as web service operations defined in WSDL, can associate specific types with their parameters and return values. If a particular type is expected by a function, and a value of a different type is passed in, then the value must be converted to the required type. If a conversion is possible, then it is handled by the language automatically, and the programmer is not required to manually cast. Attempts to pass in a value of a type for which no conversion is possible result in runtime errors.

In order to ensure a function receives input values of the correct types, it is necessary to perform this conversion before the function is called. This is handled by the final stage of the compilation process, which traverses through the graph produced in the first stage and adds type conversion operation nodes where necessary. During the first stage, as the graph is constructed, nodes are annotated with type information indicating the types that they expect on their input ports and produce on their output ports. To decide where type conversion operations are needed, each pair of connected input and output nodes is checked to see if the types match. If the type produced by the output is the same as or a subtype of the type expected on the input port, then no conversion is necessary. Otherwise, a conversion node is added between the two. Because conversions are added only when necessary, connected nodes in the graph that are known to have compatible types do not incur the overhead of performing the checks and conversions at runtime.

6 Future work

The previous sections have explained the work we have done to date which we are using as the basis for our parallel implementation of XSLT. Our current implementation serves as a demonstration of the concepts, and provides a foundation upon which additional improvements will be built. It consists of code for parsing XSLT source code, compiling it into a dataflow graph, and executing it using a sequential interpreter. However, there is not yet support for the parallelism and distribution that we are targeting our implementation at, and this is the major focus of the next stage of our work. Having solved the initial problems of designing an execution model and compilation process, we are now concentrating on six key areas of research:

- *Distributed execution of dataflow graphs*. We will explore the use of a set of interpreters running on separate machines to execute the compiled dataflow graphs.

- *Scheduling of activities to machines*. This ties in with the distributed execution, and will involve an investigation of various dynamic scheduling algorithms.

- *Instruction clustering*. This involves grouping related instructions together in order to reduce the granularity of the graph dealt with by the scheduler.

- *Improved parallelism*. The current model forces all parameters to a function to be computed before the function is called. We will investigate optimisations to the graph structure which relaxes this restriction.

- *Streaming processing of XML data*. By making the flow of data through a program explicit, our model lends itself to the possibility of processing data as it arrives, without having to wait for all input to be available before the first activity is fired.

- *Formal definition of compilation process*. This would provide a more concise and accurate definition of the language transformation, and could possibly be implemented in XSLT itself.

IEEE COMPUTER SOCIETY

7 Conclusion

We are currently developing an implementation of XSLT which enables parallel execution of programs and support for web services. This paper has explained the execution model of our system, and given details of the compilation process. Our work to date has focused on the translation of XSLT programs into dataflow graphs and developing a sequential implementation of the execution engine. In the future we intend to build on this by creating a distributed version of the interpreter which can execute the compiled dataflow graphs in parallel.

As a functional language, XSLT lends itself to automated parallelisation. Many other languages used for web service composition do not provide the same level of support for parallelism, requiring instead that it be specified manually by the programmer. Our approach makes the programmer's task easier by automatically executing sections of code in parallel wherever possible. Because the dataflow model makes data dependencies explicit, it is always possible to determine when it is safe to execute an operation based on whether or not its required data has arrived. This is in contrast to languages with explicit parallelism which require the programmer to carefully manage access to shared variables to avoid non-deterministic behaviour.

It is hoped that our work will contribute to the field of web service composition by providing an effective way to compose web services in parallel. We also see our system being applicable to the problem of large-scale processing of XML data by executing XSLT programs in parallel across a set of machines in a cluster or grid. It is our belief that these two features, combined, will provide a powerful and easy-to-use development environment for creating distributed applications based on modern XML and web services technologies.

Further information about this project is available at `http://gridxslt.sourceforge.net/`.

References

[1] L. Augustsson. A compiler for lazy ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 218–227, New York, NY, USA, 1984. ACM Press.

[2] M. Barreiro, J. L. Freire, V. M. Gulias, and J. J. Sanchez. Exploiting sequential libraries on a cluster of computers. In *Erlang Workshop, in connection with PLI 2001*, Sept. 2001.

[3] P. Briggs. Automatic parallelization. *ACM SIGPLAN Notices*, 31(4):11–15, 1996.

[4] J. Derrick. Cost effective XML processing in the datacenter. In *XML Europe 2004*, Amsterdam, The Netherlands, Apr. 2004.

[5] T. A. et. al. Business Process Execution Language for Web Services version 1.1. http://ifr.sap.com/bpel4ws/, May 2003.

[6] B. Goldberg. Buckwheat: graph reduction on a shared-memory multiprocessor. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 40–51, New York, NY, USA, 1988. ACM Press.

[7] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[8] P. M. Kelly, P. D. Coddington, and A. L. Wendelborn. Distributed, parallel web service orchestration using XSLT. In *1st IEEE International Conference on e-Science and Grid Computing*, Melbourne, Australia, Dec. 2005.

[9] P. M. Kelly, P. D. Coddington, and A. L. Wendelborn. A simplified approach to web service development. In *4th Australasian Symposium on Grid Computing and e-Research (AusGrid 2006)*, Hobart, Australia, Jan. 2006.

[10] S. Kusakabe, T. Nagai, Y. Yamashita, R. Taniguchi, and M. Amamiya. A dataflow language with object-based extension and its implementation on a commercially available parallel machine. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 308–317, New York, NY, USA, 1995. ACM Press.

[11] X. Li, R. Ferreira, and G. Agrawal. Compiler support for efficient processing of XML datasets. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 42–52, New York, NY, USA, 2003. ACM Press.

[12] S. Majithia, I. Taylor, M. Shields, and I. Wang. Triana as a graphical web services composition toolkit. In S. J. Cox, editor, *Proceedings of UK e-Science All Hands Meeting*, pages 494–500. EPSRC, CD-Rom only, Sept. 2003.

[13] A. S. Mazer. A dataflow-based APL for the hypercube. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 505–512, New York, NY, USA, 1988. ACM Press.

[14] T. Oinn, M. Addis, and J. F. et. al. Delivering web service coordination capability to users. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 438–439, New York, NY, USA, 2004. ACM Press. http://taverna.sf.net.

[15] W3C. XML Schema part 0: Primer second edition. W3C Recommendation, Oct. 2004. http://www.w3.org/TR/xmlschema-0/.

[16] W3C. XML path language (XPath) 2.0. W3C Working Draft, Apr. 2005. http://www.w3.org/TR/xpath20/.

[17] W3C. XQuery 1.0 and XPath 2.0 data model. W3C Working Draft, Apr. 2005. http://www.w3.org/TR/xpath-datamodel/.

[18] W3C. XSL transformations (XSLT) version 2.0. W3C Working Draft, Apr. 2005. http://www.w3.org/TR/xslt20/.

0-7695-2585-7/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY