Speed-up SOAP Processing by Data Mapping Template

Wei Jun  
Institute of Software, Chinese Academy of Sciences  
Beijing 100080, China  
+w8 10 62654886  
wj@otcaix.iscas.ac.cn

Hua Lei  
Institute of Software, Chinese Academy of Sciences  
Beijing 100080, China  
+h6 10 62630989  
hualei@otcaix.iscas.ac.cn

Niu Chunlei  
Institute of Software, Chinese Academy of Sciences  
Beijing 100080, China  
+n0 10 62630989  
iuchl04@otcaix.iscas.ac.cn

ABSTRACT
Web Services is gaining popularity in distributed computing due to its loosely-coupled, high-interoperable and platform-independent characteristics. However, web services suffers performance penalty because XML based SOAP is used to specify wire message format, and SOAP processing largely affects the performance of web services. In this paper, we identify that data model mapping between XML data and Java data is the main impact factor on performance, and propose a new paradigm of data model mapping - “Dynamic Early Binding” which enables to improve SOAP processing by avoiding Java reflection operations and proactively generating processing codes. This dynamic early binding is realized by Data Mapping Template (DMT), which is specified by extended context free grammar and implemented by pushdown automaton with output. We illustrate the effectiveness by applying it into a SOAP engine - SOAPExpress, and yielding over 100% speedups compared to Apache Axis 1.2 in our benchmark.

Categories and Subject Descriptors

General Terms
Algorithms, Measurement, Performance, Design.

Keywords
Web Services, SOAP, Data Mapping Template, Dynamic Early Binding, Context Free Grammar.

1. INTRODUCTION
Recently, with the development and standardization of web services protocols such as XML, SOAP [1] and WSDL [2], a new distributed computing paradigm based on web services is gaining momentum. Web services supplies the XML-based service description, registry and invocation mechanisms, and solves the interoperability problems between heterogeneous platforms.

Web services are platform-independent, high interoperable compared to other distributed computing component models such as EJB, CORBA and DCOM. However, web services suffers performance penalty which prevents from widely using in high performance computing. The performance of distributed system is strongly determined by their wire format [3]. The traditional client-server communication paradigms such as RPC offer high performance, but these systems rely on the assumption that communicating parities strictly abide certain protocol which causes highly coupling; the distributed communication paradigm such as Java-RMI adopts serialized object which lessens system coupling, but brings additional marshalling costs; web services uses XML as the message format which realizes high interoperability between heterogeneous platforms. However XML parsing and marshalling dramatically decrease the system performance.

SOAP protocol defines the message format of web services, which serves as the basis of loose-coupling and high interoperability. The core functional component of web services is SOAP Engine, which parses the XML-based SOAP message and carries on the data model mapping between XML data and platform dependent application data, so on. SOAP engine determines the performance of web services. This paper will focus on how to speedup the data mapping between XML data and Java data to improve the performance of SOAP engine.

Through experiments on widely used SOAP engine Apache Axis 1.2 [4], we identify that data model mapping between XML data and Java data is the main impact factor on performance of web services [5]. Based on analysis of the experiments, we propose a new data model mapping paradigm “Dynamic Early Binding”. Dynamic Early Binding will avoid Java reflection by keeping the mapping information and actions in dynamically generated template. This kind of template will be specified based on context-free grammar (CFG) [6], and implemented by pushdown automaton with output actions. We apply the Dynamic Early Binding technique into a SOAP Engine - SOAPExpress we developed. The SOAP processing performance of SOAPExpress is heavily improved compared with Apache Axis 1.2.

This paper is structured as follows: First, we survey related works in section 2. Section 3 proposes the Dynamic Early Binding technique. In section 4, we present the realization of Dynamic Early Binding technique – Data Mapping Template (DMT) in detail. Section 5 introduces application of DMT in SOAPExpress, and illustrates it able to improve the performance of SOAPExpress heavily by experiments. We conclude the paper and discuss our future works in section 6.
2. RELATED WORKS

There have been several studies [7,8,9,10,11] on the performance of the SOAP processing. These studies all agreed that XML based SOAP protocol incurred a substantial performance penalty compared with binary protocols.

Davis conducted an experimental evaluation on the latency performance of various SOAP implementations, comparing with other protocols such as Java RMI and CORBA/IIOP [7]. A conclusion was drawn that two reasons may cause the inefficiency of SOAP. One is about the multiple system calls to realize one logical message sending. Another is about the XML parsing and formatting. The similar conclusion was drawn in [8] by comparison with CORBA. Chiu et al. pointed out that the most critical bottleneck in using SOAP for scientific computing is the conversion between floating point numbers and their ASCII representations [9]. And Kohlhoff indicated that optimizing the SOAP encoding and decoding will improve the performance of business application in the context of web services [9]. Studies in [7,8,9,10] all considered that besides XML parsing, the transformations between XML data and application data are key impact factor on SOAP performance. Ng et al. confirmed this conclusion by undertaking benchmarks on commercial SOAP implementations [11].

Bidirectional data mappings between XML data and Java data are also called deserialization and serialization. They greatly affect the overall performance of SOAP processing. In recent research, various mechanisms are utilized to optimize the deserialization [12] and serialization[13]. In [12], rather than reserializing each message from scratch, a serialized XML message copy is saved in the sender’s stub, changes for the next same type of message will be tracked, and saved copy is reused as a template for the next sending. The serialization usually includes two processes, first getting structured field value of application object, and then mapping field value into XML data. [12] introduces ways to optimize the latter process, but not mention the former. The approach in [13] reuses matching regions from the previously deserialized application objects, and only performs deserialization for a new region that has not been processed before. However, for large SOAP message, especially for SOAP message whose data always changed with different sending, the performance improvement of [13] will be decreased. Also, Java reflection is adopted by [13] as a means to set and get new values, for large object, especially deeply nested object, this will increase performance penalty.

3. DYNAMIC EARLY BINDING

The SOAP processing of web services in client and server side is actually the data model mapping between SOAP message and platform-dependent data. The indispensable elements of data model mapping include XML data definition in XML schema, data definition in specific platform and the mapping rule between them.

Through our experiments [5], the results show that the data model mappings between XML data and Java data heavily impact the performance of SOAP engine in Java platform. Firstly, we will introduce the widely used binding techniques - early binding and late binding in data model mapping, then present the new data binding paradigm “dynamic early binding” which combines the advantages of early binding and late binding. We will use data binding and data model mapping alternatively.

Figure 1. XML data vs. Java data type

SOAP message mainly consists of Body and Head which are wrapped in the Envelope. The Head contains the QoS information such as security and reliability; and the Body consists of the business logic information such as operations, parameters or returned results all in XML format. Figure 1 shows the XML and Java data type included in SOAP request message of a web service. It represents the data model mapping between XML data and Java data.

Table 1. Comparison of data binding techniques

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Late Binding</th>
<th>Static Early Binding</th>
<th>Dynamic Early Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key techniques</td>
<td>Java reflection</td>
<td>code generation</td>
<td>Java reflection and dynamic code generation</td>
</tr>
<tr>
<td>Performance</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Flexibility</td>
<td>high</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Binding-Info</td>
<td>run time</td>
<td>compile time</td>
<td>run time</td>
</tr>
<tr>
<td>Getting</td>
<td>Apache Axis , Castor</td>
<td>XMLBeans</td>
<td>DMT</td>
</tr>
</tbody>
</table>

Here we first explain two pairs of concepts:

- **Late binding vs. Early binding**
  The difference between these two binding paradigms is the time to get binding information, and here the binding information refers to the mapping information between XML data and Java data. Late binding gets the binding information at run time, and uses the information also at run time; early binding gets the binding information at compilation time, and uses the binding information at run time.

- **Dynamic binding vs. Static binding**
  Here the dynamic binding refers to the binding mechanism which can add new mapping XML-Java type pair at run time. In contrast, static binding refers to the mechanism which can only add new mapping pairs at compilation time.

According to the above explanation, the existing data binding implementations can be classified as two categories: **Dynamic Late Binding** and Static Early Binding.

- **Dynamic Late Binding**
  Dynamic Late Binding gets the binding information by Java Reflection at run time, and then uses the binding information to carry on data binding between XML data and Java data, the representations are Apache Axis and Castor. The dynamic late binding can dynamically add the new XML-Java mapping pair, and avoid generating assistant codes by using dynamic feature.
of Java language, both of which could add flexibility though suffer performance penalty.

- In contrast, the Static Early Binding generates the Java template files which record the binding information before running, and then carries on the binding between XML data and Java data at runtime. The static early binding such as XMLBeans improves the performance by avoiding the frequent use of Java reflection; however, it couldn’t add new binding XML-Java pair at runtime, which lessens the flexibility compared to dynamic early binding.

**Figure 2. Dynamic early binding**

As illustrated in Figure 2, **Dynamic Early Binding** will establish the mapping rules between XML schema for some data type and Java class for the same data type at compilation time. At run time, a kind of Java template is generated based on XML schema, Java class and their mapping rules, which we call **Data Mapping Template-DMT**, by dynamic code generation techniques. The DMT will be used to drive the data mapping procedure.

The dynamic early binding avoids the Java reflections so that the performance of data model mapping can be distinctly improved; simultaneously, DMTs can be generated and managed at run time, which makes the Dynamic Early Binding owning similar flexibility as dynamic late binding. The dynamic early binding combines the advantages of static early binding and dynamic late binding, and is the key technique to achieve the goal of high performance SOAP processing.

4. REALIZATION OF DYNAMIC EARLY BINDING

We have analyzed the features of dynamic early binding, and point out that it is the key technique to improve the performance of SOAP engine. In this section, we present our solution for realization of dynamic early binding – Data Mapping Template (DMT), its description and implementation model.

4.1 Specifying DMT by extended Context-Free Grammar

Figure 1 describes the XML data type defined by XML Schema and Java data type defined by Java language. Here we use extended context-free grammar to depict XML data model and Java data model, and use mapping scheme of grammar production to describe the mapping relationship of two data models.

**Definition 1. Data Mapping Template – DMT.** For a data type \( T \), the data mapping template \( DMT = (G_X, G_J) \), \( G_X \) and \( G_J \) are context-free grammars for XML data model and Java data model.

**Definition 2. Context-Free Grammar for DMT**

\[ G = (V, T, P, S, M) \]

- \( V \) is a set of non-terminals, \( \forall A \in V \), \( A \) is called non-terminal. For \( G_X \), \( V \) is the set of data type \( T \) which is defined by XML schema, including simple type, composite type and array type. For \( G_J \), \( V \) is the set of data type \( T \) which is defined by Java language, including primitive types, user-defined Java class, and array types.

- \( T \) is a set of terminals, \( \forall \alpha \in T \), \( \alpha \) is called terminal. For \( G_X \), \( T \) is the set of XML tag names; for \( G_J \), \( T \) is the set of Java field names.

- \( P \) is the set of grammar productions, where each production \( p \) is like \( A \rightarrow \alpha, \alpha \in V, \alpha \in \{ V \cup T \}, |A| \leq |\alpha| \).

- \( S \) is one of non-terminals as start symbol, and the \( S \) represents the data model defined by this grammar.

- \( M \) is a set of mapping schemes, which defines actions in the reduction process. \( \forall m \in M, m \) is called mapping scheme, and consists of a group of atomic operations. The mapping scheme \( m \) and production \( p \) correspond to each other one by one, \( \forall m \in M, \exists p \in P, p \rightarrow m \), and vice versa.

**Table 2. Atomic operation set of XML and Java data model**

<table>
<thead>
<tr>
<th>Operation set ( A^X ) for XML data model</th>
<th>Operation set ( A^J ) for Java data model</th>
</tr>
</thead>
<tbody>
<tr>
<td>createElement(eleName)</td>
<td>creatJavaType(typeName)</td>
</tr>
<tr>
<td>getElementValue(eleName)</td>
<td>setSimpleValue()</td>
</tr>
<tr>
<td>setElementValue(eleValue)</td>
<td>setFieldValue(fieldName, fieldValue)</td>
</tr>
<tr>
<td>addChildElement(ele)</td>
<td>getFieldName()</td>
</tr>
<tr>
<td>getIndexValue(index)</td>
<td>setIndexValue(index, indexValue)</td>
</tr>
<tr>
<td>setIndexValue(index, indexValue)</td>
<td>returnTypeValue()</td>
</tr>
</tbody>
</table>

In the definition of Grammar \( G \), each grammar production is associated with one or a set of basic operations. Table 2 lists the basic operations for the XML data model and Java data model. XML data is organized as a tree structure, and the node value of tree-structured XML data is parsed and approached by XML parser. The basic operations of XML model include creating XML element, setting and getting the child element, setting and getting the element value.

Java is an object-oriented language, the data types of which include primitive, array and user-defined. The primitive types include the primitive data types such as int and the primitive wrapper classes such as Integer. Class is used to specify user-defined data type, the field of which can be accessed directly as a public class variable or indirectly via some accessing methods (setters and getters). The array value can be accessed and assigned at specific index. The basic operations of Java data model include creating and getting value for Java data types, setting and getting the field value for user-defined data type, setting and getting the index value for array types, and setting the primitive types.

Table 3 includes the grammar \( G \) and mapping scheme \( M \) for XML data model and Java data model.
Table 3. Mapping Schemes of $G_X, G_J$

<table>
<thead>
<tr>
<th>$G_X$'s mapping scheme $M_X$</th>
<th>$S \rightarrow tag$ ($T_X.el = S.ele$) $T_X.tag^*$ ($S.value = T_X.value$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_X$ $\rightarrow$ ($T_X'.value = createJavaType(), T_X'.ele = T_X.ele$, $T_X'.value = T_X'.value$)</td>
<td></td>
</tr>
<tr>
<td>$T_X'\rightarrow tag$ ($T_X'.ele = T_X'.getNextChildElement()$) $T_X'$ ($T_X'.setFieldValue(T_X'.value)$) $tag'$ ($T_X'.value = T_X'.value$, $T_X'.ele = T_X'.ele$)</td>
<td></td>
</tr>
<tr>
<td>$T_X^i \rightarrow \xi$, $i = 0, 1, 2, ...$</td>
<td></td>
</tr>
<tr>
<td>$T_X$ $\rightarrow$ ($T_X.value = createJavaType()$, $T_X.value = T_X'.ele$) $T_X'$ ($T_X'.value = T_X'.value$)</td>
<td></td>
</tr>
<tr>
<td>$T_X'$ $\rightarrow$ ($T_X'^2.value = createJavaType()$, $T_X'^2.value = T_X'.ele$) $T_X'^3$ ($T_X'^3.value = T_X'.value$)</td>
<td></td>
</tr>
<tr>
<td>$T_X'^3.S$ $\rightarrow$ ($T_X'^3.setSimpleValue(T_X'^3.value)$) $value$</td>
<td></td>
</tr>
<tr>
<td>$T_X'^2.Tj$ $\rightarrow$ ($T_X'^2.setElementValue(T_X'^2.value)$) $value$</td>
<td></td>
</tr>
<tr>
<td>$T_X'^A$ $\rightarrow$ ($T_X'^A.createElement(T_X'.ele)$) $T_X'^A$ ($T_X'^A.setIndexValue(T_X'.value)$) $tag'$ $T_X'^A$</td>
<td></td>
</tr>
<tr>
<td>$T_X^a$ $\rightarrow$ $\xi$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$G_J$'s mapping scheme $M_J$</th>
<th>$S \rightarrow$ ($S.createElement()$, $T_X.value = S.value$, $T_X.ele = S.ele$) $T_J'A$ ($S.addChildElement(T_X.ele)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_J' \rightarrow$ ($T_J'.value = new Person(), T_J'.ele = T_J'.ele$, $T_J'.value = T_J'.value$)</td>
<td></td>
</tr>
<tr>
<td>$T_J'^a$ $\rightarrow$ name ($T_J'^a.value = T_J'.value$, $T_J'.name$) name' ($T_J'^a.value = T_J'.value$, $T_J'^a.value = T_J'.ele$)</td>
<td></td>
</tr>
<tr>
<td>$T_J'^3 \rightarrow$ ($T_J'^3.value = T_J'.value$) $T_J'^4$</td>
<td></td>
</tr>
<tr>
<td>$T_J'^5 \rightarrow$ ($T_J'^5.setFieldValue(T_J'.value)$) $T_J'^5$</td>
<td></td>
</tr>
<tr>
<td>$T_J'^6$ $\rightarrow$ ($T_J'^6.setSimpleValue(T_J'^6.value)$) $value$</td>
<td></td>
</tr>
<tr>
<td>$T_J'^7$ $\rightarrow$ ($T_J'^7.setElementValue(T_J'^7.value)$) $value$</td>
<td></td>
</tr>
<tr>
<td>$T_J'^8$ $\rightarrow$ ($T_J'^8.createElement(T_J'.ele)$) $T_J'^8$ ($T_J'^8.setIndexValue(T_J'.value)$) $tag'$ $T_J'^8$</td>
<td></td>
</tr>
<tr>
<td>$T_J^a$ $\rightarrow$ $\xi$</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. DMT for the Data Type illustrated in figure 1

$G_X^3$

| $S \rightarrow p$ ($T_X\.ele = S\.ele$) $T_X' p'$ ($S\.value = T_X\.value$) |
| $T_X' \rightarrow$ ($T_X\.value = new Person()$, $T_X\.ele = T_X'.ele$, $T_X\.value = T_X'.value$) |
| $T_X'^a$ $\rightarrow$ name ($T_X'^a\.value = T_X'\.value$.getFirstChildElement()$) $T_X'^a\.name = T_X'\.value$ name' ($T_X'^a\.value = T_X'\.value$, $T_X'^a\.value = T_X'\.ele$) |
| $T_X'^3 \rightarrow$ ($T_X'^3.value = T_X'\.value$) $T_X'^4$ |
| $T_X'^5 \rightarrow$ ($T_X'^5\.setFieldValue(T_X'^5\.value)$) $T_X'^5$ |
| $T_X'^6$ $\rightarrow$ ($T_X'^6\.setSimpleValue(T_X'^6\.value)$) $value$ |
| $T_X'^7$ $\rightarrow$ ($T_X'^7\.setElementValue(T_X'^7\.value)$) $value$ |
| $T_X'^8$ $\rightarrow$ ($T_X'^8\.createElement(T_X'\.ele)$) $T_X'^8$ ($T_X'^8\.setIndexValue(T_X'\.value)$) $tag'$ $T_X'^8$ |
| $T_X^a$ $\rightarrow$ $\xi$ |

$G_J^3$

| $S \rightarrow$ ($S\.createElement("p")$, $T_X\.value = S\.value$, $T_X\.ele = S\.ele$) $T_J'A$ ($S\.addChildElement(T_X\.ele)$) |
| $T_J' \rightarrow$ ($T_J'.value = T_J'\.ele$, $T_J'\.value = T_J'\.value$) $T_J'$ |
| $T_J'^a$ $\rightarrow$ name ($T_J'^a\.value = T_J'\.value$, $T_J'\.name = T_J'\.createElement("name")$) $T_J'^a\.value = T_J'\.value$, $T_J'\.value = T_J'\.ele$) $T_J'^i$ |
| $T_J'^3 \rightarrow$ ($T_J'^3\.value = T_J'\.value$) $T_J'^4$ |
| $T_J'^5$ $\rightarrow$ ($T_J'^5\.setFieldValue(T_J'^5\.value)$) $T_J'^5$ |
| $T_J'^6$ $\rightarrow$ ($T_J'^6\.setSimpleValue(T_J'^6\.value)$) $value$ |
| $T_J'^7$ $\rightarrow$ ($T_J'^7\.setElementValue(T_J'^7\.value)$) $value$ |
| $T_J'^8$ $\rightarrow$ ($T_J'^8\.createElement(T_J'\.ele)$) $T_J'^8$ ($T_J'^8\.setIndexValue(T_J'\.value)$) $tag'$ $T_J'^8$ |
| $T_J^a$ $\rightarrow$ $\xi$ |

$G_X^3$ is used to specify XML data model expressed usually by XML Schema. For the grammar production set $P_X$, $S$ is the start symbol, $tag$ denotes the name of start element, $tag'$ denotes the name of end element; $T_X$ denotes some XML data type, which will be reified as one of XML Schema’s built-in simple type $T_X^S$, array type $T_X^A$, or composite type $T_X^K$. $G_X$ specify the XML data model as a structural tree which is composed of tag names. $T_X$ has two attributes - $T_X^S$ and $T_X^A$. $T_X^S$ represents the XML data, and $T_X^A$ represents the corresponding Java data.
G′ is used to specify Java data model expressed by Java language. For the grammar production set P′, T′ denotes some Java data type, which will be refined as primitive type T′P, array type T′A, or Java class T′C; field is the field name in user-defined Java class. T′ has two attributes - T′.value and T′.ele. T′.value represents the Java data value, and T′.ele represents the corresponding XML data value. From the G′ point of view, a Java data type can be also viewed as a structural-tree, and be depicted in a manner as XML data.

Section 3 mentioned that based on XML data type, Java data type and their mapping rules, the DMT could be generated automatically by algorithm. Table 3 gives the general mapping rules between XML data type and Java data type, and JAX-RPC could be used to build its XML schema for some Java data type. Figure 3 depicts the generation algorithm for DMT, its input of which is a Java data type, and the output is DMT composed of a pair of specific G′X and G′J, by analyzing the hierarchy of Java data type using Java reflection in a depth first traversal. To be simple, the algorithm in Figure 3 omits the recursive analytic logic for user-defined classes and array type.

![Figure 3 Algorithm for generation of DMT](image)

The DMT for the data type illustrated in Figure 1 is specified in table 4, it’s produced automatically by the algorithm.

### 4.2 Implementing DMT by Pushdown Automaton with Output

Last subsection introduced the DMT’s description model by extended Context-Free Grammar, and this subsection will give out the implementation model by pushdown automaton with output actions. For any data type T, the implementation model of DMT is a pair of pushdown automata which is used to recognize a pair of grammars (G′X, G′J), and we call this kind of pushdown automation with output actions as Data Mapping Automaton.

**Definition 3. Data Mapping Automaton** DMA = ( Q, Σ, Γ, Z0, q0, F, O, δ )

- Q is a set of states, ∀ q ∈ Q, q is the state of DMA ;
- Σ is the set of input symbols;
- Γ is a set of stack symbol, ∀ A ∈ Γ is a stack symbol;
- Z0 ∈ Γ is the start stack symbol;
- q0 ∈ Q is the start state of DMA ;
- F is the set of final states, ∀ f ∈ F, f is the final state;
- O is a set of output actions;
- δ is the state transition function and governs the behaviors of the automaton. δ ( q, a, Z ) = { ( p1, y1, o1 ) , ( p2, y2, o2 ), … , ( pm, ym, om ) } , represents that When read input symbol is a, the top stack element is Z and the state is q, DMA can transit the state to p1 and pop the top stack element Z, then push the stack symbol y1 into the stack, and carry on the output action o1, for i = 1,2,3,…,m.

The Data Mapping Automaton DMA = ( Q, Σ, Γ, Z0, q0, F, O, δ ) can be generated by an extended context-free grammar G = ( V, T, P, S, M ). The transformation and equivalence proof between context-free grammar and pushdown automaton can be found in [6].

DMA′X and DMA′J for grammar G′X and G′J are shown in Fig 4, both of which are comprised of input object, input transformer, stack, state controller (SC) and state table (ST). The input transformer will first transfer the input object into what can be recognized by DMA; state table is a two-dimensional array P[i,a], A is indexed by non-terminal and a by terminal, array element P[i,a] records the state transition and output action; state controller controls the state transition and executes the output actions.

![Figure 4 DMA′X, DMA′J for grammar G′X and G′J](image)
the virtual data type tree in a depth-first manner, and returns Java type event to state controller. DMA\(^3\)'s state controllers looks up state table by Java type event and top stack symbol, and makes the state transition and executes the mapping actions such as initializing XML output stream, writing the Java object into the XML output stream in a structural manner, etc. When DMA\(^4\) stops, a XML output stream will be generated and returned.

For the mapping from XML data to Java data, DMA\(^5\) is utilized to traverse input XML stream and construct a Java object by DMA\(^5\)'s output actions. In contrast, DMA\(^1\) is used to visit the fields of Java object in a structural manner and build the XML output stream.

The DMT driven mechanism makes the best of XML Pull Parser to complete the mapping from XML data to Java data in one traversal of XML data. Also DMT uses Java Type Reader to read structural data type information, and completes the generation of XML output stream when DMA\(^1\) traverses the Java object once.

5. EVALUATION OF DMT IN SOAPExpress

SOAPExpress is a high performance SOAP engine we developed by using XML Pull Parsing and dynamic early binding techniques. The dynamic early binding improves the performance of data model mapping in SOAP message processing, and also keeps the flexibility by adding new mapping type at run time. The dynamic early binding is realized by DMT introduced in Section 4.

5.1 Application of DMT in SOAPExpress

![Diagram of SOAP processing in SOAPExpress](image)

SOAPExpress is hosted as a web application by Tomcat. As shown in Fig 5, when SOAPExpress receives a request message, Service-In Handler undertakes to get the name of targeting web service by XML Pull Parser. For RPC-encoded and literal-wrapped style, SOAP body contains the operation information of a service. Through DMT Manager, Service-In Handler can get all the DMTs corresponding to an operation’s input parameters, the DMT instances will take charge of the data model mapping from XML data to Java data. After that, Service Invoker invokes the service and returns the Java object result. Then, Service-Out Handler gets the DMT instances of output parameters from DMT Manager, and the DMT instances drive the mapping from Java object to XML output stream.

There are three components in SOAPExpress to support the DMT driven data mapping. The DMT Manager supplies APIs for search, generation and record of DMTs. DMT Generator creates the DMTs from Java data type definitions. DMT instances are recorded and placed in DMT Registry, which to be used to quickly locate some DMT.

Service-In/Out Handlers look up DMT instances firstly through DMT Manager. The DMT Manager will check the existence of DMT instances. If existed, DMT instances will be returned directly; if not, DMT Manager will call the DMT Generator to generate the DMT instances for corresponding input and output parameters, then return them and put them into DMT Registry.

DMT Registry is in charge of recording and holding DMT instances. DMT Registry is indexed by service name which is bound by DMT instances with its input and output parameters. The main function of DMT Registry is to maintain the mapping relationship from service name to DMT instances.

The DMT Generator composes of Type Analyzer, Model Builder and Byte Code Generator. Firstly, Type Analyzer uses Java reflection to obtain the hierarchy of some Java type, and then Model Builder builds the java object data and XML data from obtained information by the generation algorithm depicted in figure 3, section 4.1. After that, Byte Code Generator generates the bytecode of DMT using Javassist [15] at runtime. The DMT instance for a java type is generated only once at the first time, can be reused as long as the service has no changes.

5.2 Experiments and Results Analysis

We choose the WS Test 1.0 suite [16] to test the time spent on each stage in the SOAP message processing. And several kinds of test cases are designed to measure the performance of various types of web services call, which are described below:

1. **echoVoid**: Send/receive an message with empty body.
2. **echoStruct**: Send/receive an array of size 20, each element is a complex data type composed of three elements - an integer, float and string.
3. **echoList**: Send/receive a linked list of size 20, each element of the list is a Struct defined in echoStruct.

The experimental settings are as follows:

- **CPU**: Pentium-4 2.80 GHz, Memory: 512 MB; OS: Windows XP Professional SP2; JVM: J2SK 1.4.2; Web services runtime: Apache AXIS 1.2; Web container: Tomcat 5.0; XML Parser: Apache Xerces-J 2.6.2.
- The web service client performs 10,000 iterations for each web service call, and the client load is 5 hits per second.

![Performance Comparison](image)
Figure 6 is on the performance comparison between Apache Axis 1.2 and SOAPExpress. For echoStruct and echoList, the XML payload is about 4KB. The measure is on SOAP processing time, which takes count from receiving a HTTP request to returning the HTTP response. The figure shows that about method echoVoid it is very close on time for the two SOAP engines, since method echoVoid has no business logic and just returns the SOAP message with empty body. However, about method echoStruct, the processing time of SOAPExpress is only about 46% of Apache Axis 1.2; about method echoList, the proportion is about 44%. The parameters of method echoList and echoStruct are complex array type and nested data type respectively. The result shows that the performance of SOAPExpress on SOAP processing is much better than that of Apache Axis 1.2.

Figure 7 shows the performance comparison between Apache Axis 1.2 and SOAPExpress with the increase of XML payload. The results illustrate that when XML payload increases, the processing time of SOAPExpress increases much slower than that of Apache Axis 1.2.

![Figure 6. Performance Comparison of Apache Axis and SOAPExpress on XML payload](image)

As explained in [5], frequently using Java reflection causes performance penalty for large size of XML data processing. SOAPExpress, with DMT-driven data mapping, avoids the use of Java reflection so as to have much better performance than that of Apache Axis 1.2.

6. CONCLUSIONS AND FUTURE WORKS

This paper proposes a new data model mapping paradigm “Dynamic Early Binding”, and presents its realization - Data Mapping Template (DMT), which is specified by extended context free grammar and implemented by pushdown automaton with output actions. The DMT-driven data mapping technique realizes quick mappings between XML data and Java data by dynamically generating templates at runtime. The DMT technique has been applied in the implementation of a high performance SOAP engine – SOAPExpress. The evaluation shows that the performance of SOAPExpress is better than that of Apache Axis 1.2. For medium and large size of SOAP message, the performance advantage of SOAPExpress is much more obvious.

The dynamic early binding technique –DMT proposed in the paper doesn’t support the complete XML infoset so far, such as namespace and attribute, which will be considered to add into the DMT definition and implementation. Also, exception handling will be treated in the implementation of DMT - pushdown automaton in our future work.

7. REFERENCES