High Performance SOAP Processing Driven by Data Mapping Template

Wei Jun, Hua Lei, Niu Chunlei, and Zheng Haoran

Technology Center of Software Engineering, Institute of Software
Chinese Academy of Sciences, Beijing 100080, China
wj@otcaix.iscas.ac.cn

Abstract. Web Services, with loosely-coupled, high-interoperable and platform-independent characteristics, is gaining popularity in distributed computing. However, web services suffers performance penalty because its protocol stack is based on XML. SOAP is used to specify wire message format in web services, and SOAP processing largely affects the performance of web services. In this paper, we firstly analyze the performance of web services on Java platform, and identify that data model mapping between XML data and Java data is the main impact factor on performance. Therefore, we propose a new scheme 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 introduce the technique into our developed SOAP engine – SOAPExpress. The effectiveness is illustrated by yielding over 100% speedups compared to Apache Axis 1.2 in our benchmark.

1 Introduction

Recently, with the development and standardization of web services protocols such as XML, SOAP and WSDL, a new distributed computing paradigm based on web services is gaining momentum. Web services supplies the XML-based service description, service registry and service 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 it from widely using in high performance computing. The performance of distributed system is strongly determined by their wire format [1]. 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.

F. Eliassen and A. Montresor (Eds.): DAIS 2006, LNCS 4025, pp. 152 – 168, 2006.

SOAP protocol defines the message format of web services, which serves as the basis of loosely-coupled, high interoperable web services. The core function 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.

We first analyze the performance of web services based on widely used SOAP engine Apache Axis 1.2[2], and identify that data model mapping between XML data and Java data is the main impact factor on performance.

Based on experiments, we propose a new data model mapping paradigm "Dynamic Early Binding". Dynamic Early Binding avoids the use of Java reflection by keeping record of the mapping information and actions in dynamically generated template. The template will be specified based on context-free grammar (CFG), and implemented by pushdown automaton with output actions. We apply the Dynamic Early Binding technique into a high performance SOAP Engine - SOAPExpress we developed. The average 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 analyzes the performance impact points in SOAP message processing, introduces the performance-related techniques, and 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 the application of DMT in SOAPExpress, and illustrates that the DMT improves the performance of SOAPExpress heavily on experiments. We conclude the paper and discuss our future works in section 6.

2 Related Works

There have been several studies on the performance of the SOAP processing [3],[4],[5],[6],[7]. 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 [3]. 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 [4] by comparison with CORBA. Chiu et al. pointed put that the most critical bottleneck in using SOAP for scientific computing is the conversion between floating point numbers and their ASCII representations [5]. And Kohlhoff indicated that optimizing the SOAP encoding and decoding will improve the performance of business application in the context of web services [6]. Studies in [3],[4],[5],[6] 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 [7].

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 [8] and serialization [9]. In [8], rather than re-serializing 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. In [8], several means were introduced to optimize the latter process, but not mention the former. The approach in [9] 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 [9] will be decreased. Also, Java reflection is adopted by [9] as a means to set and get new values, for large object, especially deeply nested object, this will increase performance penalty.

3 Background on Web Services Performance

This section will first analyze the SOAP message processing on the server side, and find the performance bottlenecks. Then we will introduce some basic techniques in SOAP processing, which are the basis of our research work.

3.1 Analysis of Web Services Performance

We use Apache Axis 1.2, one of the most popular web services middleware systems as our testing environment. Apache Axis works as a web application that is located in a web container, so the web container carries on the work of receiving SOAP request message and sending response message through HTTP protocol. Though the HTTP protocol is a possible bottleneck for web services, we will not discuss the point in this paper.

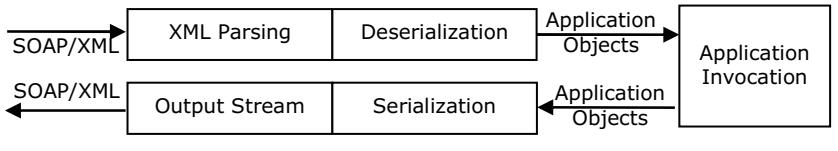

Fig. 1. SOAP processing flow

We divide SOAP message processing into five stages which are shown in Fig. 1.

1. XML Parsing Stage. The XML request message will be parsed by XML parser. In this stage, AXIS1.2 uses the SAX parser as XML parser; it reads the ASCII format data and records the SAX event in the buffer.
2. Deserialization Stage. The parsed XML data will be deserialized to application objects that will be presented to the web services as application object parameters.

At this stage, Axis1.2 replays the recorded SAX event and notifies the deserializers of SAX events to do the deserialization work.

3. Application Invocation Stage. This stage contains the business logic, which calls the application of targeting web service and gets the result of the application. The time spent in this stage is closely related with the complexity of application, no matter whether the business logic is wrapped as a web service or as an EJB.
4. Serialization Stage. This stage is the reverse process of deserialization stage; the application object result will be serialized to XML content. In AXIS1.2 implementation, the XML content is written into a memory buffer.
5. Output Stream Stage. In this stage, the buffered response XML data is written into the output stream. After that, the output stream will be written to the HTTP response object, and the web container will send the response XML data back to client through HTTP connection.

This section surveys the time cost on each stage, and analyzes the bottlenecks of SOAP message processing. As shown in Figure 1, the XML Parsing stage and deserialization stage carry on the mapping from XML data model to Java data model, and serialization stage and output stream stage carry on mapping from Java data model to XML data model.

We choose the WS Test 1.0[10] to test the time spent on different stages in the SOAP message process and to analyze the performance bottleneck. WS Test is a web services test developed by Sun Microsystems. Because we focus on SOAP message process, the web service methods perform no business logic but simply return the parameters that were passed in. It is designed to measure the performance of various types of web services calls, which are described below:

1. echoVoid: Sends and receives an empty message. This tests the performance of the web services infrastructure.
2. echoStruct: Sends and receives an array of size 20, the element of the array is a complex type composed of three elements, each of which is an integer, float and string data type respectively. This method is to test the SOAP engine's ability to process array of complex flat objects
3. echoList: Sends and receives a linked list of size 20, each element of the list is a Struct defined in echoStruct. This method is to test SOAP engine's ability to process deeply nested object.

The experimental environment is set as follows. CPU: Pentinm-4.1 2.80 GHz, Memory: 512 MB, OS: Windows XP Professional SP2, JVM: J2SK 1.4.2, Web services middleware: 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, and the client load is 5 hits per second.

Fig. 2 shows the average time spent on SOAP message process stages, the XML payload is 4 KB, and here the XML payload refers to size of XML data which is to be deserialized to object data. From the experiments, we can see that there are three performance components in the SOAP message process: XML Parsing, deserialization, and serialization. In the XML Parsing stage, the time required for XML parsing of the whole process time is about 80%, 39%, 38% for echoVoid, echoStruct, echoList respectively. In deserialization stage, the time percentages are 33% for

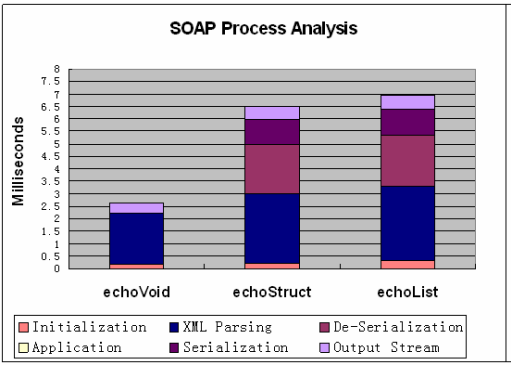

Fig. 2. Processing time on stages

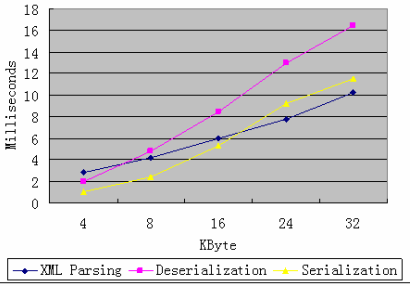

Fig. 3. Processing time on XML payload

echoStruct and 32% for echoList. In serialization stage, the time percentages are 19% for echoStruct and 18% for echoList. In the paper, we call these three stages "Data Model Mapping" which includes the data mapping between XML data and Java data.

In summary, the time spent on these three components is above 90% of the whole time, and deserialization and serialization occupy more than 50%. We increase the XML payload of echoList and record the time spent on these three stages.

Fig. 3 shows the time of method echoList spent on XML parsing, deserialization and serialization stages on different XML payload. As input XML payload increases in size, the time spent on XML parsing and deserialization is also increased, so is the time spent on serialization stage. However, the time spent on deserialization and serialization stages grows dramatically faster than that of XML parsing. As shown by the statistics, the deserialization becomes the biggest part when the size of XML payload exceeds 8KB. So we can conclude that for small XML payload, the XML parsing will be the biggest performance component, but for median and large XML payload, deserialization and serialization will account for the performance latency.

In the deserialization stage and serialization stage of Apache Axis1.2, data mappings between XML data and java object are implemented by java reflection technique. However, java reflection technology is generally considered to be inefficient, according to our experiments, for complex java object, especially nested java object such as linked list in echoList, most of the time is spent on java reflection operations.

3.2 XML Pull Parsing

The SOAP message is based on XML format, so XML parsing is an important component in SOAP message processing. However, XML parsing is normally considered to be time-consuming. In section 3.1, results further point out that XML parsing accounts for more than 35% of the whole SOAP message processing time. So the high performance of XML parsing will lead to the improvement of SOAP message processing. Now the most popular XML parsing paradigms is DOM[11] and

SAX[12]. The DOM builds a complete object representation of the XML document in memory, and then the application visits the built XML model. This can be memory intensive for large documents. SAX parses the whole XML document into a series of SAX event, and informs the application through callbacks. Apache Axis adopts SAX to parse the XML document, compared to DOM, the SAX needn't read the whole document into memory. However, writing the callback methods to deal with XML document adds complexity to application. Meanwhile, both the SAX and DOM require two passes through XML data, firstly, they build the XML representation of the whole XML document; secondly, the application visits the built representation. The extra pass through XML document reduces the SOAP message processing.

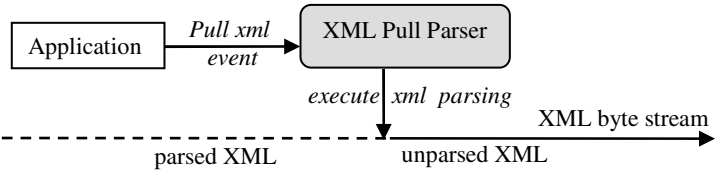

Fig. 4. XML Pull Parsing

As shown in Fig.4, XML Pull Parsing is an application-driven XML parsing paradigm. The application pulls the XML event from XML Pull Parsing, and gets the event of XML elements sequentially. XML Pull Parsing needn't read the whole XML data into memory like DOM, or to write callback method like SAX. Also, it goes through XML data at one pass to avoid extra performance penalty. Because of above advantages, XML Pull Parsing is adopted by us as the XML parsing mechanism, and works as the basis of dynamic template-driven data model mapping technique.

3.3 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.

Section 3.1 shows that the data model mappings between XML data and Java data heavily impact the performance of SOAP engine in Java platform. We will firstly introduce the widely used binding techniques - early binding and late binding in Data Model Mapping, then present a new data binding paradigm "dynamic early binding" which combines the advantages of early binding and late binding. In this subsection, we will use *data binding* and *Data Model Mapping* alternatively.

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. Fig.5 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.

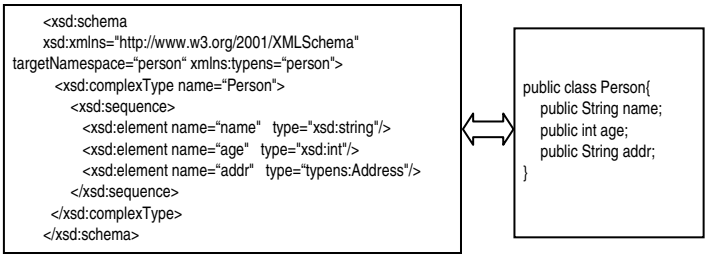

Fig. 5. XML schema vs. Java type

Table 1. Comparison of binding techniques

	Dynamic Late Binding	Static Early Binding	Dynamic Early Binding
Key techniques	Java reflection	code generation	dynamic code generation
Performance	Low	high	high
Flexibility	high	low	high
Binding-Info Getting	run time	compile time	Run time
Representative	Apache Axis, Castor	XMLBeans	DMT

Here we firstly 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 the getting and using of binding information are carried out in parallel; early binding gets the binding information at compilation time, and then uses 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 can only add new mapping pairs at compile time.

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

- The Dynamic Late Binding gets the binding information by Java Reflection technique at run time, and getting and using of binding information are carried in parallel, such as Apache Axis and Castor.
- In contrast, the Static Late Binding generates the Java template files which record the binding information before runtime, and then carries on the binding between XML data and Java data at runtime. The static late 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.

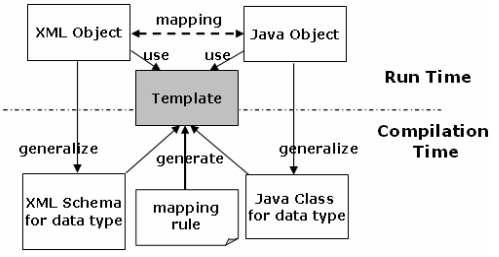

Fig. 6. Dynamic early binding

As illustrated in Fig. 6, **Dynamic Early Binding** generates the Java template class which we call **Data Mapping Template-DMT** at runtime by dynamic code generation techniques, and then the DMT will drives the data mapping process. The dynamic early binding avoids the Java reflections which improve the model mapping performance; meanwhile, the DMT files can be generated and managed at run time which makes dynamic early binding the similarly flexible as dynamic late binding. The dynamic early binding combines the advantages of static early binding and dynamic late binding, and is the key technique of high performance SOAP processing.

4 DMT Driven Data Model Mapping

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

4.1 Specifying DMT by Extended Context-Free Grammar

Figure 5 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, $\forall A \in \mathbb{V}$, 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, $\forall \alpha \in T$, α 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 $A \rightarrow \alpha$, $A \in V$, $\alpha \in V \cup T$, $|A| \leq |\alpha|$.
- S is one of non-terminals as start symbol, and the S represents the data model defined by this grammar.
- M is 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, $\forall m \in M$, $\exists p \in P$, $p \leftrightarrow m$, and vice versa.

In the definition of Grammar G, each grammar production corresponds to one or a group of atomic operations, Table 2 shows the atomic operations for the XML data model and Java data model. XML data is organized as tree-structured format, the element value of which is parsed and approached by XML parser. The atomic operations of XML model include the creating of XML element, setting and getting operations of child element, setting and getting operations of element value.

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

Operation set A^X for XML data model	Operation set A^J for Java data model
creatElement(eleName) getElementValue(eleName) setElementValue(eleValue) addChildElement(ele) getNextChildElement() returnElement()	creatJavaType(typeName) setSimpleValue() setFieldValue(fieldName, fieldValue) getFieldName() getFieldValue(fieldName) setIndexValue(index, indexValue) getIndexValue(index) returnTypeValue()

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

Table 3 shows the grammar G and mapping scheme M for XML data model and Java data model. G^X describes the XML data model defined by XML Schema. In grammar production P^X, **S** is the start symbol, **tag** represents the name of start element, **tag'** represents the name of end element; $\mathbf{T^X}$ represents the XML data type, which can be classified into XML Schema's built-in simple type $\mathbf{T^{XS}}$, array type $\mathbf{T^{XA}}$, and composite type $\mathbf{T^{XC}}$. G^X depicts the XML data model into a structured tree which is composed of tag names. $\mathbf{T^X}$ has two attributes $\mathbf{T^X.ele}$ and $\mathbf{T^X.value}$. $\mathbf{T^X.ele}$ represents the XML data of XML type**,** and $\mathbf{T^X.value}$ represents the corresponding Java data.

Table 3. Mapping Schemes of G^X, G^J

G^X's mapping scheme M^X	– **S → tag** {T^X.ele = S.ele} **T^X tag'** { S.value = T^X.value} – **T^X →** {$T^{XC}{}_0$.value = createJavaType(),T^X.ele = $T^{XC}{}_0$.ele }**$T^{XC}{}_0$**{T^X.value = $T^{XC}{}_0$.value} – **$T^{XC}{}_i$ → tag** {$T^{X'}$.ele = $T^{XC}{}_i$.getNextChildElement() }**$T^{X'}$** {$T^{XC}{}_i$.setFieldValue($T^{X'}$.value)} **tag'** {$T^{XC}{}_{i+1}$.value=$T^{XC}{}_i$.value, $T^{XC}{}_{i+1}$.ele=$T^{XC}{}_i$.ele } **$T^{XC}{}_{i+1}$** – **$T^{XC}{}_{i+1}$ →ε, i = 0,1,2,...** – **T^X →** {T^X.value = createJavaType(), T^{XS}.ele = T^X.ele}**T^{XS}** {T^{XS}.value = T^X.value} – **T^X →** {T^{XA}.value = createJavaType(),T^{XA}.ele = T^X.ele }**T^{XA}** {T^X.value = T^{XA}.value} – **T^{XS}→ε**{T^{XS}.setSimpleValue(T^{XS}.getElementValue())} – **T^{XA} → tag** {T^X.ele = T^{XA}.getNextChildElement()} **T^X** {T^{XA}.setIndexValue(T^X.value)} **tag' T^{XA}** – **T^{XA} → ε**
G^J's mapping scheme M^J	– **S →** {S.creatElement(), T^J.value = S.value, T^J.ele=S.ele} **T^J** { S.addChildElement(T^J.ele)} – **T^J →** {$T^{JC}{}_0$.ele = T^J.ele, $T^{JC}{}_0$.value = T^J.value} **$T^{JC}{}_0$** – **$T^{JC}{}_i$ → field** {$T^{J'}$.value = $T^{JC}{}_i$.getFieldValue(field)}, $T^{J'}$.creatElement()} **$T^{J'}$** {$T^{JC}{}_i$.addChildElement($T^{J'}$.ele), $T^{JC}{}_{i+1}$.value=$T^{JC}{}_i$.value,$T^{JC}{}_{i+1}$.ele=$T^{JC}{}_i$.ele} **$T^{JC}{}_{i+1}$** – **$T^{JC}{}_{i+1}$ →ε, , i = 0,1,2,...** – **T^J →** {T^{JS}.ele = T^J.ele, T^{JS}.value = T^J.value}**T^{JS}** – **T^{JS} →** {T^{JS}.setElementValue(T^{JS}.value)}**ε** – **T^J →** {T^{JA}.ele = T^J.ele , T^{JA}.value = T^J.value}**T^{JA}** – **T^{JA} →** {T^J.creatElement(), T^J.value = T^{JA}.getIndexValue()} **T^J** – {T^{JA}.addChildElement(T^J.ele) } **T^{JA}** – **T^{JA} → ε**

G^J describes the Java data model defined by Java language. In grammar production P^J, **T^J** represents the XML data types, which can be classified into primitive type **T^{JS}**, array type **T^{JA}**, and Java class **T^{JC}**; **field** is the field name in user-defined Java class. **T^J** has two attributes **T^J.value** and **T^J.ele. T^J.value** represents the Java data value of Java type, and **T^J.ele** represents the corresponding XML data value. In G^J, the Java data type is seen as a structured-tree, the primitive type tree has only a root node, the array type tree has the root node with same-structured children nodes, and the children nodes of the class type tree represent its field types. The Java data type can be described in tree-structured manner as XML data in Table 3.

Section 3.3 mentioned that the XML data types defined by XML schema have certain mapping rules to data types of different platforms, and JAX-RPC [13] defines the mapping rules between Java data type and XML data type. So grammar G^J and grammar G^X can be generated by analyzing Java data type and it's mapping rules to XML schema. Fig.7 shows the generation algorithm of G^X and G^J, the input of algorithm is a Java data type, and the output is G^X and G^J by analyzing the hierarchy of Java data type using Java reflection in a depth first traverse. To be simple, the algorithm in Fig 7 omits the recursive analytic logic for user-defined classes and array type. The G^X and G^J of data type in Fig 5 are shown below:

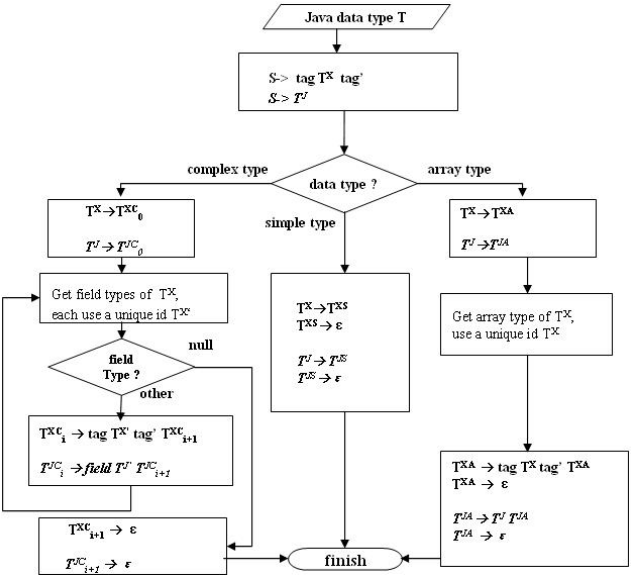

Fig. 7. Algorithm for generation of G^X and G^J

G^X	**S →** **p** {T^x.ele = S.ele} $\mathbf{T^x}$ **p'** {S.value = T^x. value} $\mathbf{T^x}$ **→** {$T^{xc}{}_0$.value = new Person(),$T^{xc}{}_0$.ele = T^x.ele} $\mathbf{T^{xc}{}_0}$ {T^x.value = $T^{xc}{}_0$.value } $\mathbf{T^{xc}{}_0}$ **→ name** {T^{xn}.ele = $T^{xc}{}_0$.getNextChildElement()} $\mathbf{T^{xn}}$ {$T^{xc}{}_0$.value.name = T^{xn}.value} **name'** {$T^{xc}{}_1$.value = $T^{xc}{}_0$.value, $T^{xc}{}_1$.ele = $T^{xc}{}_0$.ele }$\mathbf{T^{xc}{}_1}$ $\mathbf{T^{xn}}$ **→** {N.ele = T^{xn}.ele, N.value = new String()}**N** { T^{xn}.value = N.value} **N →ε**{N.value = N.getElementValue()} $\mathbf{T^{xc}{}_1}$ **→ age** {T^{xa}.ele = $T^{xc}{}_1$.getNextChildElement()}$\mathbf{T^{xa}}$ {$T^{xc}{}_1$.value.age = T^{xa}.value} **age'** {$T^{xc}{}_2$.value = $T^{xc}{}_1$.value, $T^{xc}{}_2$.ele = $T^{xc}{}_1$.ele }$\mathbf{T^{xc}{}_2}$ $\mathbf{T^{xa}}$ **→** {A.ele = T^{xa}.ele, A.value = (int)0} **A** { T^{xa}.value = A.value} **A →ε**{A.value = Integer.parseInt(A.getElementValue())} $\mathbf{T^{xc}{}_2}$ **→ addr** {T^{xad}.ele = $T^{xc}{}_2$.getNextChildElement()} $\mathbf{T^{xad}}$ {$T^{xc}{}_2$.value.addr = T^{xad}.value } **addr'** {$T^{xc}{}_3$.value = $T^{xc}{}_2$.value, $T^{xc}{}_3$.ele = $T^{xc}{}_2$.ele }$\mathbf{T^{xc}{}_3}$ $\mathbf{T^{xc}{}_3}$ **→ ε** $\mathbf{T^{xad}}$ **→** {ADDR.ele = T^{xad}.ele, ADDR.value = new String()} **ADDR** { T^{xad}.value = ADDR.value} **ADDR →ε**{ADDR.value = ADDR.getElementValue()}
G^J	**S →** {S.creatElement("p"), T^j.value = S.value, T^j.ele = S.ele } $\mathbf{T^j}$ {S.addChildElement(T^j.ele)} $\mathbf{T^j}$ **→** {T^{jc}.ele = T.ele, T^{jc}.value = T^j.value}$\mathbf{T^{jc}}$ $\mathbf{T^{jc}{}_0}$ **→ name** { T^{jn}.value = $T^{jc}{}_0$.value.name, T^{jn}.creatElement("name")} $\mathbf{T^{jn}}$ {$T^{jc}{}_0$.addChildElement(T^{jn}.ele), $T^{jc}{}_1$.value = $T^{jc}{}_0$.value, $T^{jc}{}_1$.ele = $T^{jc}{}_0$.ele} $\mathbf{T^{jc}{}_1}$ $\mathbf{T^{jn}}$ **→** {N.ele = T^{xn}.ele, N.value = T^{xn}.value} **N** **N →** {N.setElementValue(N.value)}**ε** $\mathbf{T^{jc}{}_1}$ **→ age** { T^{ja}.value = $T^{jc}{}_1$.value.name, T^{ja}.creatElement("age")} $\mathbf{T^{ja}}$

	{T^{jc}_1.addChildElement(T^{ja}.ele), T^{jc}_2.value = T^{jc}_1.value, T^{jc}_2.ele = T^{jc}_1.ele } $\mathbf{T^{jc}_2}$ $\mathbf{T^{ja}}$ → {A.ele = T^{ja}.ele, A.value = T^{ja}.value } **A** **A** → { A.setElementValue(A.value)}**ε** $\mathbf{T^{jc}_2}$→ **address** {T^{jad}.value = T^{jc}_2.value.addr, T^{jad}.creatElement("address")} $\mathbf{T^{jad}}$ {T^{jc}_2.addChildElement(T^{jad}.ele), T^{jc}_3.value = T^{jc}_2.value, T^{jc}_3.ele = T^{jc}_2.ele } $\mathbf{T^{jc}_3}$ $\mathbf{T^{jc}_3}$ → **ε** $\mathbf{T^{jad}}$ → { ADDR.ele = T^{jad}.ele, ADDR.value = T^{jad}.value} **ADDR** **ADDR** → { ADDR.setElementValue(ADDR.value)}**ε**

4.2 Implementing DMT by Pushdown Automaton with Output

Last Section introduced the DMT's conceptual model by extended Context-Free Grammar, and this section will give out the implementation model by pushdown automaton with output. For any data type T, the implementation model of DMT is a pair of pushdown automata which is used to recognize grammar (G^X, G^J), and we call this kind of pushdown automation with output **Data Mapping Automaton**. The execution of data mapping automaton is the execution of grammar's mapping schema, and also the mapping process from one data model to another.

Definition 3. Data Mapping Automaton DMA= $(Q, \sum, \Gamma, Z_0, q_0, F, O, \delta)$

- Q is a set of states, $\forall q \in Q$, q is the state of DMA;
- $\sum$ is the set of input symbols;
- $\Gamma$is a set of stack symbol, $\forall A \in \Gamma$ is a stack symbol;
- $Z_0 \in \Gamma$is the start stack symbol;
- $q_0 \in Q$ is the start state of DMA ;
- F is the set of final states, $\forall f \in F$, 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. $\delta(q, a, Z) = \{ (p_1, \gamma_1, o_1), (p_2, \gamma_2, o_2), \ldots, (p_m, \gamma_m, o_m) \}$, 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 p_i and pop the top stack element *Z*, then push the stack symbol γ_i into the stack, and carry on the output action o_i, for $i = 1, 2, 3, \ldots, m$.

The Data Mapping Automaton DMA= $(Q, \sum, \Gamma, Z_0, q_0, F, O, \delta)$ 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 [14].

Fig 8 shows the data mapping automata DMA^X and DMA^J for grammar G^X and G^J, 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[*A*, *a*], *A* is indexed by non-terminal and *a* by terminal, array element P[*A*,*a*] records the state transition and output action; state controller controls the state transition and executes the output actions.

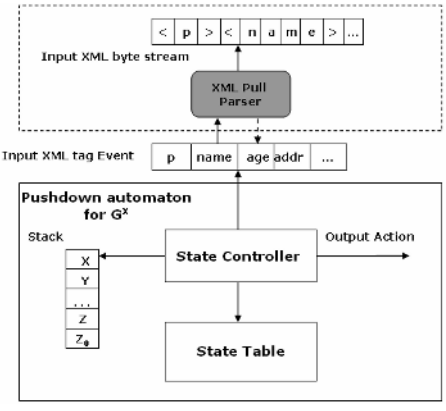

Fig. 8(a). DMA^X for grammar G^X

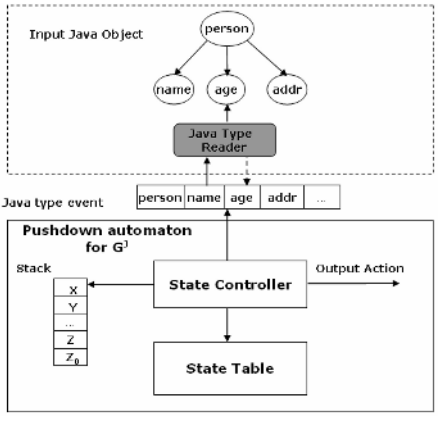

Fig. 8(b). DMA^J for grammar G^J

The input object of DMA^X is XML byte stream, and the input transformer is XML Pull Parser. When DMA^X starts, its state controller drives XML Pull Parser to read the XML byte stream and get the XML tag events. State controller looks up state table by the XML tag event and the top stack symbol, and then makes the state transition and executes the mapping actions in the state table such as creating new Java object, setting value of Java object, etc. When DMA^X stops, a Java object will be constructed and returned.

The input object of DMA^J is Java object, and the input transformer is Java Type reader. The Java data types can be viewed as a structural-tree. For a specified Java data type, a virtual data type tree can be generated and kept in memory. The Java type reader traverses the virtual data type tree in a depth-first manner, and returns Java type event to state controller. DMA^J '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^J stops, a XML output stream will be generated and returned.

For the mapping from XML data to Java data, DMA^X is utilized to traverse input XML stream and construct a Java object by DMA^X's output actions. In contrast, DMA^J 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^J traverses the Java object once.

5 Application and Evaluation of DMT in SOAPExpress

SOAPExpress is the high performance SOAP engine 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-driven data model mapping introduced in Section 4.

5.1 Application of DMT in SOAPExpress

SOAPExpress is hosted as a web application by Tomcat. As shown in Fig 9, 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.

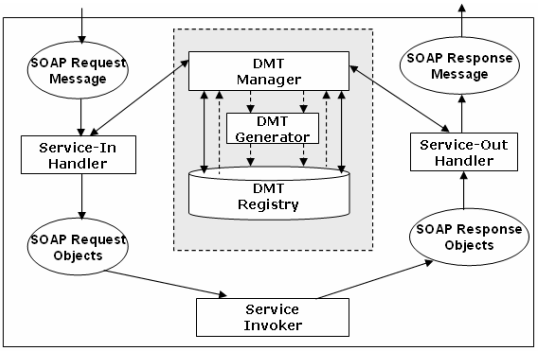

Fig. 9. SOAP processing in SOAPExpress

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 7. 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

This subsection will test the performance of SOAP message processing by SOAPExpress, and compare the result with Apache Axis 1.2. The test suite and environment are the same as the setting in section 3.1.

Figure 10 illustrates the performance comparison between Apache Axis 1.2 and SOAPExpress for WS Test 1.0, the XML payload is 4KB for *echoStruct* and *echoList*. The measure is on SOAP processing time, which begins from the point SOAPExpress receives a HTTP request and ends at the point SOAPExpress returns the HTTP response. The statistics shows that times about method *echoVoid* are very close for these two SOAP engines, since method *echoVoid* has no business logic and just returns empty SOAP message. 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 echoList and echoStruct are complex array type and nested data type. The result shows that the performance of SOAPExpress is higher than Apache Axis 1.2.

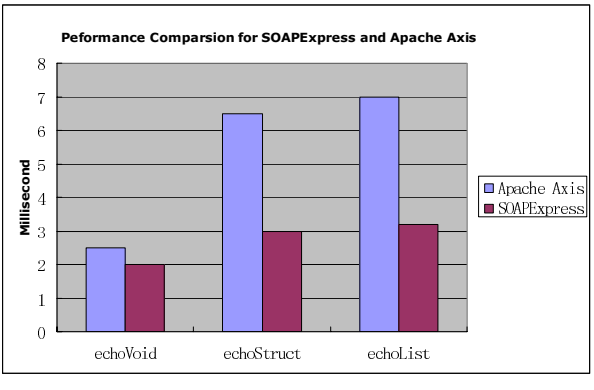

Fig. 10. Performance Comparison of Apache Axis 1.2 and SOAPExpress

We increase the XML payload, and analyze the impact of XML payload on SOAP message processing performance. Fig 11 shows that when the XML payload increases, the time of SOAPExpress increases much slower than that of Apache Axis 1.2. The Section 3.2 points out the frequent use of Java reflections cause performance penalty for large XML data. The DMT-driven data model mapping avoids the use of Java reflection, and keeps the processing time increases slowly with XML payload.

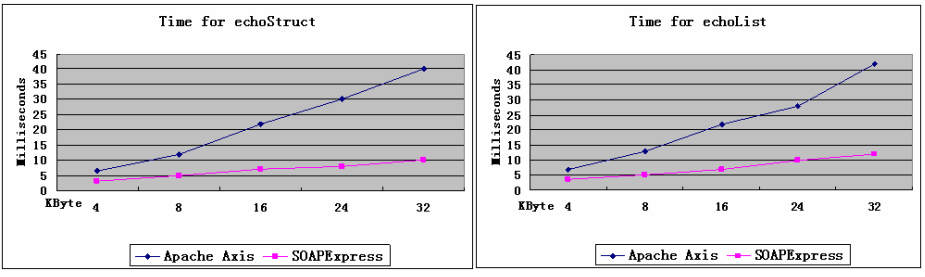

Fig. 11. Comparison of Apache Axis 1.2 and SOAPExpress on XML payload

6 Conclusion and Future Work

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 utilized in the implementation of a high performance SOAP engine – SOAPExpress. The testing results show that the performance of SOAPExpress is better than 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 yet. Such as *namespace* and *attribute*, they 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.

References

1. Bustamante, F. E., Eisenhauer, G., Schwan, K., and Widener, P.: Efficient wire formats for high performance computing. In Proceedings of Supercomputing 2000(SC 2000), IEEE CS Press(2000) 64-64.
2. The Apache Software Foundation, Apache Axis 1.2. http://ws.apache.org/axis/.
3. Davis, D. and Parashar, M.: Latency performance of SOAP implementations. In Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE CS Press(2000) 407-412.
4. Elfwing,R., Paulsson, U., Lundberg, L.: Performance of SOAP in Web Service Environment Compared to CORBA. Proceedings of the Ninth Asia-Pacific Software Engineering Conference (APSEC'02). IEEE CS Press(2002) 84-96.
5. Chiu, K., Govindaraju, M., Bramley, R.: Investigating the limits of SOAP performance for scientific computing. In Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), IEEE CS Press(2002) 246-254.
6. Kohlhoff, C. and Steele, R.: Evaluating SOAP for high performance business applications: Real-time trading systems. In Alternate Proceedings of the Twelfth International World. Wide Web Conference, (2003) 262-270.

7. Ng, A., Chen, S. P. and Greenfield, P.: An Evaluation of Contemporary Commercial SOAP Implementations. In Proceedings of the 5th Australasian Workshop on Software and System Architecture, Adelaide, Australia (2003) 64-71.
8. Abu-Ghazaleh, N. Lewis, M. J., Govindaraju, M.: Differential Serialization for Optimized SOAP Performance. In Proceedings of the 13th IEEE International Symposium on High Performance. Distributed Computing (HPDC-13), IEEE CS Press(2004) 55-64.
9. Suzumura, T., Takase, T. and Tatsubori, M.: Optimizing Web Services Performance by Differential Deserialization. In Proceedings of IEEE/ACM International Conference on Web Services, IEEE CS Press(2005) 185-192.
10. WS Test 1.0, http://java.sun.com/performance/reference/whitepapers/WS_Test-1_0.pdf.
11. Document Object Model, http://www.w3.org/DOM/.
12. Simple API for XML, David Brownell, SAX2, O'Reilly & Associates, Inc.(2002.)
13. Java API for XML-Based RPC, http://java.sun.com/webservices/jaxrpc/docs.html.
14. Linz, P.: An Introduction to Formal Languages and Automata, third edition, Jones & Bartlett Publishers (2001).
15. The JBoss Community, Javassist, http://www.jboss.com/products/javassist.