Toward Remote Object Coherence with Compiled Object Serialization for Distributed Computing with XML Web Services*

Robert van Engelen[1], Wei Zhang[1], and Madhusudhan Govindaraju[2]

[1] Dept. of Computer Science, Florida State University
[2] Dept. of Computer Science, State University of New York (SUNY) at Binghamton
engelen@cs.fsu.edu
wzhang@cs.fsu.edu
mgovinda@binghamton.edu

Abstract. Cross-platform object-level coherence in Web services-based distributed systems and grids requires lossless serialization to ensure programming-language specific objects are safely transmitted, manipulated, and stored. However, Web services development tools often suffer from lossy forms of XML serialization, which diminishes the usefulness of XML Web services as a competitive approach to binary protocols. The difficulty mainly originates from the impedance mismatch between programming language data types and XML schema types. To overcome this obstacle, we propose hybrid static/dynamic algorithms to support lossless serialization of programming-language specific binary-encoded object graphs to text-based XML trees, while staying within the limits imposed by XML schema validation and the XSD type system. This paper presents a compiler-based approach to automatically emit serialization routines for C and C++ data types to XML. Experimental results show that the presented compiler-based serialization is efficient and performance is comparable to systems that use binary protocols.

1 Introduction

XML Web services architectures support the service-oriented computing (SOA) paradigm, which is loosely defined as a services-based distributed computing approach to achieve interoperability between distributed applications deployed by disparate organizations across the Internet. Web services in essence provide platform-neutral distributed computing environments by using W3C-approved open XML standards. However, the technology has received limited success in certain application areas that require strong object-level coherence due to the impedance mismatch between programming language types and XML schema types (XSD types) [18]. Current XML document-centric Web services implementations avoid this issue by supporting loosely-coupled data exchanges in semi-structured XML documents. This tends to work well for business-oriented

* This work is supported in part by the US Department of Energy DEFG02-02ER25543

hierarchical data structures, but is far too simplistic for science and engineering applications deployed on computational grids. Application-centric Web service implementations must use carefully crafted bijective mappings to serialize internal application data to XML and vice versa using structurally precise and semantically safe translations. In practice this has proven to be difficult given that serialization must take place within the limits imposed by XML schema standards and the XSD type system. This is especially hard to achieve with XML Web services in heterogeneous distributed systems with platform-specific nodes that may adopt different and non-standard XML serialization methods. To avoid these issues, current Web services implementations of computational grids often advocate the use of a single programming language with a select choice of Web services toolkits. This severely limits the applicability of the approach to heterogeneous systems and negates the benefits of XML Web services in general.

To address these shortcomings we developed compiler-based techniques to generate serialization algorithms to safely translate C and C++ data structures to XML and vice versa. Because standard C and C++ runtime environments do not implicitly carry runtime type information on data structures and object instantiations needed to perform the translation to XML, we used a hybrid form of static and dynamic type analysis. Static analysis is used to build a plausible data model at compile time for representing the possible instances of object graphs by tracking down object relationships. This analysis is comparable to static shape analysis [5] and related to points-to analysis [11]. We then use the model to generate type-specific serialization algorithms. The generated serialization algorithms analyze the actual runtime object graph instances using compile-time hints to effectively serialize them in XML, and vice versa, using a mapping that guarantees object-level coherence. We implemented the approach in the gSOAP [13, 14] toolkit for C and C++ and tested the approach against other toolkits such as Apache Axis for Java and .NET. Performance results are shown for a gSOAP benchmark application on a variety of machines.

The remainder of this paper is organized as follows. Section 2 presents a brief overview of some of the most widely used systems and protocols for data exchange in distributed applications. The mapping of types to XML schema is discussed in Section 3 and applied to C and C++. XML serialization for object-level coherence is introduced in Section 4 followed by a presentation of the serialization algorithms in Section 5. Section 6 presents performance results to verify the efficiency of the approach on various platforms. The paper summarizes the conclusions in Section 7.

2 Motivation and Related Work

While object serialization in binary protocols such as the Java RMI object serialization protocol, XDR for Sun RPC, CORBA's IIOP, and Microsoft's DCOM have been around for years, serializing objects in XML is relatively new. XML serialization is gaining traction in Web services applications to achieve interoperability across programming language domains and disparate organizations. An advantage is that XML schemas are platform-neutral in contrast to RMI and DCOM, more expressive compared to CORBA's IDL, and enables a wider use of tools and systems for XML processing, storage, and retrieval.

Large-scale distributed systems require strong object coherence guarantees [2] to ensure that objects moved, cached, and copied across a set of nodes in a distributed system preserve their structure and state. Platform-specific approaches achieve this goal through, mostly proprietary, binary serialization protocols. Modern programming languages such as Java and C# are intrinsically equipped with object serialization capabilities to support remote object invocation, persistent object storage, and message passing in distributed systems. The programming languages support an explicit form of object-level coherence in which separately compiled applications must meet minimum requirements for consistency by sharing object definitions (e.g. class files). Implicit object-level coherence can be found in programming languages for distributed systems, e.g. Orca [3].

Several systems and protocols have been proposed and developed since the early 1980s for inter- and intra-application data exchange. This section briefly reviews some of the most widely used systems and protocols. Because the security mechanisms of these systems is poor or at least require additional transport-level security, they operate mostly on LANs behind firewalls. In contrast, XML Web services consist of a set of firewall-friendly open standards for (mostly synchronous) data exchange across the Internet, message-level security and authentication, message routing, resource management, peer notification, etc.

Sun Microsystems' RPC (Remote Procedure Call) compiler generates stub and skeleton code for marshaling simple data structures between client and server applications. The marshaling process convert application data into XDR (External Data Representation) [7] for transmission. XDR is an IETF (Internet Engineering Task Force) standard [7] for the description and encoding of data. XDR supports a subset of C types and cannot be used to serialize pointer-based data structures.

CORBA is a platform-independent architecture ORB (Object Request Brokerage) architecture [10]. CORBA's IIOP (Internet Inter ORB Protocol) is used to transmit objects between CORBA applications. IIOP supports a wide variety of data types that can be specified in IDL (Interface Description Language). CORBA is a proprietary heavy-weight product.

Microsoft's DCOM protocol is similar to IIOP and enables COM objects on different Windows-based systems to communicate. Although DCOM is a platform-independent protocol, it is mainly used within Windows environments.

Sun Microsystems' Java RMI (Remote Method Invocation) [12] serializes objects between Java applications. There is no limit on the type of data objects that can be exchanged. Entire object graphs can be serialized. Associated class bytecodes are loaded on demand.

The Message Passing Interface (MPI) library [6] is a platform-independent lower-level message passing architecture for efficient communication of numerical data among communicating groups of nodes in a cluster or SMP machine. The Parallel Virtual Machine (PVM) library [4] is similar to MPI.

Several Web services toolkits for SOAP/XML [15] are available for various programming languages, such as Apache Axis for Java and C++ [1], SOAP Lite for Perl [8], and gSOAP for C and C++ [13]. The Microsoft .NET framework [9] provides a platform-dependent Web services framework for C#. The .NET framework supports serialization of data objects managed by the CLR (Common Language Runtime). The .NET framework includes the IIS (Internet Information Services) Web server to deploy .NET applications as Web services.

3 Mapping C and C++ Types to XML Schema

The XML Web services standard supports two XML encoding styles: *SOAP-RPC encoding* style and *document literal* style [15]. The choice of encoding style is fixed in the WSDL (Web Services Definition Language) [16] interface definition of a service. However, the two styles differ significantly in the expressiveness of the serialized XML representation of application data, and consequently the algorithms for mapping application data to XML.

3.1 RPC Encoding Style

The SOAP-RPC (Remote Procedure Calling) encoding style is a standard SOAP 1.1/1.2 [15] serialization format that can be viewed as the greatest common denominator of types among programming-language type systems. The encoding supports types that have equal counterparts in many programming languages, which greatly simplifies interoperability. To this end, SOAP-RPC encoding uses a subset of the XSD type system by limiting the choice of XML schema components to an orthogonal subset of structures to represent primitive types, records, and arrays. In addition, a mechanism for multi-referenced objects is available to support the serialization of object graphs. However, there are two problems with RPC encoding. The first is that the multiref serialization with `href` and `id` attributes violates XML schema validation constraints, because these attributes are not part of the schema of a typical data structure. The second problem is that the serialization of nil references, multi-referenced objects, and (sparse) multi-dimensional arrays is not precisely defined which leads to interoperability problems that are often related to the use of `id` and `href` references. For example, *every* object in the graph is serialized with `id` and `href` by Apache Axis [1] rather than the multi-referenced objects alone, making it difficult to achieve object-level coherence across programming language domains.

Table 1 shows the mapping of primitive and compound C/C++ types to XSD types and XML schema components for SOAP-RPC encoding with gSOAP. Mappings for Java, C#, and other mainstream languages are similar. Note that the full set of primitive XSD types is not shown in Table 1. Additional XSD types, such as `xsd:decimal`, can be represented by other types, e.g. strings. The encoding is consequently controlled at the application layer. With gSOAP, users can bind these XSD types to C/C++ types using a `typedef`, for example:

```
typedef char *xsd__decimal;
```

	C/C++ Type *T*	**Target XML Schema Type**
primitive	bool	`xsd:boolean`
	char	`xsd:byte`
	short	`xsd:short`
	int32_t	`xsd:int`
	int64_t	`xsd:long`
	float	`xsd:float`
	double	`xsd:double`
	size_t	`xsd:unsignedLong`
	time_t	`xsd:dateTime`
	char*	`xsd:string`
	wchar_t*	`xsd:string`
	std::string	`xsd:string`
	enum	`xs:simpleType/restriction/enumeration`
	typedef *T*	`xs:simpleType/extension`
compound	struct	`xs:complexType/sequence`
	class	`xs:complexType/complexContent/extension`
	typedef *T*	`xs:complexType/complexContent/extension`
	T [*nnn*]	*SOAP-encoded array of T*
	T *	*the schema type of T*

Table 1. Mapping C/C++ Types to Schema Types for SOAP-RPC Encoding

Each struct or class data member is mapped to a local `xs:element` of the `xs:complexType` for the struct or class. See Figure 1 for an example. SOAP-RPC encoding requires arrays to be encoded as "SOAP encoded arrays" [15], where each SOAP array is a type restriction of the generic SOAP array schema. Another disadvantage of mapping C arrays to XML is the absence of a true array type in C (arrays in C are pointers). Arrays are either declared as fixed-size arrays or have to be declared as a struct with a pointer __ptr and __size field to store the runtime array size, for example:

```
struct floatarray { float *__ptr; int __size; };
```

Languages that support arrays as first-class citizens, such as Java and C#, can map arrays to SOAP arrays without forcing users to adopt mapping structures.

The XML schema standard adopted by the Web services architecture requires support for XML namespaces. XML namespaces bind user-defined types to one or more type spaces, similar to C++ namespaces. However, C does not support namespaces. Therefore, an alternative mechanism is used by optionally qualifying type names with a namespace prefix:

```
enum prefix__name { ... };
struct prefix__name { ... };
class prefix__name { ... };
typedef T prefix__name;
```

C Source Declarations	**Target XML Schema**
	`<simpleType name="State">`
	`  <restriction base="xsd:string">`
typedef char *xsd__decimal;	`    <enumeration value="OFF"/>`
	`    <enumeration value="ON"/>`
enum State {OFF, ON};	`  </restriction>`
	`</simpleType>`
struct Example	`<complexType name="Example">`
{	`  <sequence>`
char *name;	`    <element name="name" type="xsd:string"/>`
xsd__decimal value;	`    <element name="value" type="xsd:decimal"/>`
enum State state;	`    <element name="state" type="tns:State"/>`
struct Example *list;	`    <element name="list" type="tns:Example"`
};	`              minOccurs="0" nillable="true"/>`
	`  </sequence>`
	`</complexType>`

Fig. 1. Example Mapping of C Type Declarations to XML Schema

Suppose for example two distinct List data structures are used by two different services. One service uses the 'x' namespace while the other uses 'y', bound to namespace URIs http://www.x.org and http://www.y.org, respectively:

```
//gsoap x schema namespace: http://www.x.org
struct x__List { char *key, *val; struct x__List *next; };
//gsoap y schema namespace: http://www.y.org
typedef xsd__NMTOKENS y__List;
```

where the latter list is a space-separated list of tokens. Namespace bindings are used in RPC encoding and document literal styles.

3.2 Document Literal Style

A unique feature of the gSOAP toolkit is the full support for document literal style for XML serialization, which covers the entire XML schema component definition space. Document literal style encoding is a significant departure from RPC encoding by promoting expressiveness as opposed to the simplicity of an orthogonal type system. On the one hand, the expressiveness allows variant records (unions) to be serialized and arrays can be serialized in-line instead of separately using the SOAP array encoding format. On the other hand, the absence of a standard out-of-band mechanism for object referencing, such as the SOAP-RPC multi-ref encoding with `href` and `id` attributes, is a concern. This poses additional challenges for object-level coherent serialization guarantees.

The liberation from the SOAP-RPC encoding constraints mainly affects the mapping of structs and classes to the `xs:complexType` schema component. Without loss of generalization, the differences can be summarized as follows:

– Enables the use of XML attribute definitions within a `xs:complexType`, where XML attributes can be instances of primitive XSD types and instances of `xs:simpleType`;
– Allows repetitions of an `xs:element` in a `xs:complexType` sequence, i.e. elements that may have multiple occurrences indicated by `xs:maxOccurs`>1;
– Allows the use of `xs:choice` and `xs:any`;
– Allows the use of `xs:group` and `xs:attributeGroup`. However, these macro structures have no effect on the mapping since they can be expanded within the schema and only serve as syntactic conveniences.

These content definitions are enabled in gSOAP using the declarations of struct and class members as is shown in Table 2. In addition, the document literal style supports `xs:complexType` extensions, i.e. a simple form of inheritance that is accomplished with C++ class inheritance.

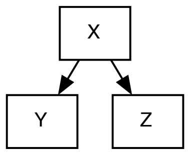
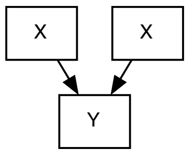
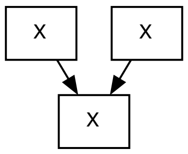
Attributes are declared with a special qualifier '`@`', STL containers such as vectors are mapped to (potentially) unbounded sequences of elements, members that point to dynamic arrays must be preceded by a int __size field that holds the runtime array size information. The use of STL containers and pointers to arrays as shown in Table 2 is preferred over SOAP-RPC encoded arrays, see also WS-I Basic Profile [19]. A union must be preceded by a variant record discriminator int __union that holds the index to the union field to be serialized, and void pointers must be preceded by int __type field that holds the runtime type tag value of the object pointed to. Note that the source-code level changes are all ANSI C compliant, except for the use of '`@`' to annotate data members for attribute serialization.

4 XML Serialization for Object-Level Coherence

To ensure object-level coherence of serialized object graphs in XML, a referencing mechanism in XML must be used to represent graph edges. Any explicit representation of edges in XML will preserve the logical structure of object graphs,

Struct/Class Member m	**Change**	**Target XML Schema Type**
$T\ m$;	`@` $T\ m$;	`xs:attribute/@type="`T`"`
std::vector⟨T⟩ m;	*none*	`xs:element/@maxOccurs="unbounded"` `xs:element/@type="`T`"`
T *m; *(points to multiple elements)*	int __size; T *m;	`xs:element/@maxOccurs="unbounded"` `xs:element/@type="`T`"`
union $U\ m$;	int __union; union $U\ m$;	`xs:choice`
void *m;	int __type; void *m;	`xs:element/@type="xs:anyType"`

Table 2. Data Member Changes to Support Document Literal Style Serialization

	Tree	**DAG**	**DAG**
Shape			

Schema	`<complexType name="X">`
	`  <sequence>`
	`    <element name="y"`
	`        type="tns:Y"/>`
	`    <element name="z"`
	`        type="tns:Z"/>`
	`    ...`
	`  </sequence>`
	`</complexType>`
		`<complexType name="X">`
		`  <sequence> ... </sequence>`
		`  <attribute name="ref"`
		`        type="REF"/>`
		`</complexType>`
		`<complexType name="Y">`
		`  <sequence>...</sequence>`
		`  <attribute name="id"`
		`        type="ID"/>`
		`</complexType>`
			`<complexType name="X">`
			`  <sequence>...</sequence>`
			`  <attribute name="id"`
			`        type="ID"/>`
			`  <attribute name="ref"`
			`        type="REF"/>`
			`</complexType>`

XML	`<x>`
	`  <y>...</y>`
	`  <z>...</z>`
	`  ...`
	`</x>`
		`<x ref="123">...</x>`
		`...`
		`<x ref="123">...</x>`
		`...`
		`<y id="123">...</y>`
			`<x ref="456">...</x>`
			`...`
			`<x ref="456">...</x>`
			`...`
			`<x id="456">...</x>`

Fig. 2. Three Different Object Referencing Graph Examples

such as DAGs and cyclic graphs. While SOAP-RPC implicitly relies on multi-reference encoding with `id` and `href` attributes to represent edges, document literal does not support this mechanism and the schemas must, in principle, explicitly define `xsd:ID` and `xsd:REF` attributes for each XML component that resembles an application data type that can be referenced.

Consider the three different object referencing graphs shown in Figure 2. Regular tree-based XML document configurations do not require an explicit referencing mechanism, because graph nodes are simply nested in XML. DAGs and cyclic graphs must be serialized using explicit references to preserve their logical structure, e.g. using `id` and `ref` attributes. This requires additional declarations of these attributes to the schema components of the objects, assuming document literal style is used.

Suppose that organization A defines a schema for an object and organization B wants to define object graphs where A's object can be multi-referenced, e.g. in a DAG. Then, A's schema must be changed to include `id` and `ref` attributes, assuming document literal encoding style is used. This is problematic, because after A's schema is published it is usually detrimental to interoperability to change it. Note that SOAP-RPC style with implicit referencing can only be used if A's schema conforms to the RPC style restrictions.

It is quite common that object referencing is either completely enabled or disabled depending on the application's requirements. Therefore, explicitly adding referencing attributes to each `xs:complexType` is cumbersome and introduces an unnecessary layer of complexity. The approach is rarely (if ever) implemented. A meta-level binding mechanism to classify pointers as reference (non-nil pointer), unique (can be nil and is the single reference to an object), or full (to shared and aliased objects) as in DCE IDL is only partially possible (i.e. reference or full using the `xs:nillable` attribute). A default referencing mechanism with well-defined semantics for serialization is essential to simplify cooperation between organizations and to ultimately improve interoperability. Web services standards have to consider implicit referencing mechanisms to support object-level coherence. A first step in this direction is the publication of XML ID [17].

Another problem is the loss of floating point precision by representing numerical data in XML. We can avoid this by using hexadecimal or base64-encoded IEEE 754 floating point values. Encoding and decoding must be performed by a pre-processor prior to XML validation to avoid flagging the values as invalid.

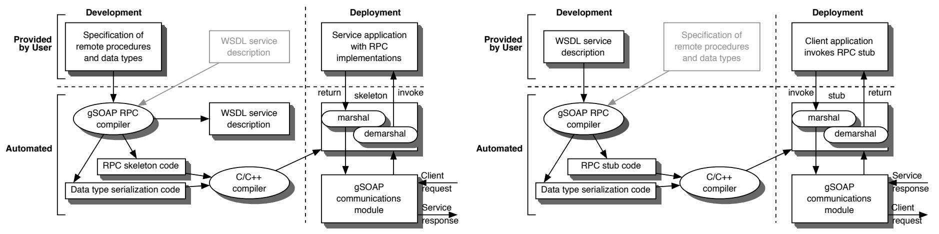
We consider the following issues critical for achieving object-level coherence:

– For document literal encoding the biggest problem is the absence of a default referencing mechanism in XML. We suggest using `id` (or `xml:id` from the XML ID [17]) and `ref` similar to SOAP 1.2 RPC encoding instead of explicitly adding `xsd:ID` and `xsd:REF` attribute declarations to schemas;
– To prevent loss of precision, avoid serializing floats and doubles as XSD types. We suggest using IEEE 754 floats encoded in hexadecimal or base64 by defining a simpleType restriction of `xsd:hexBinary` or `xsd:base64Binary`, respectively.

Furthermore, SOAP/XML processors must be aware of the referencing mechanism and obey the serialization rules to preserve the logical structure of an object graph.

5 A Static-Dynamic Approach to XML Serialization

Based on the requirements for object-level coherence, we implemented serialization techniques that preserve the logical structure and state of C/C++ data structures serialized in XML. The serialization of object graphs in gSOAP is automatic and relies on a static-dynamic approach. The gSOAP *soapcpp2* compiler generates stubs and skeletons for remote procedure calling. The server application development and deployment is shown in Figure **3a**. A client application



a) Server Configuration **b) Client Configuration**

Fig. 3. Developing and Deploying Server and Client Applications with gSOAP

can be build from a WSDL (using the gSOAP *wsdl2h* preprocessor) or directly from the service specification, as is shown in Figure 3**b** (shaded box). In both server and client application development, C/C++ type definitions are parsed by the *soapcpp2* gSOAP RPC compiler to generate the serialization routines for each type.

5.1 Static Analysis

The *soapcpp2* compiler builds a plausible data model at compile time using static analysis by tracking down object relationships. The model represents the possible instances of object graphs. The model is then used to generate type-specific serialization algorithms that analyze the actual runtime object graph instances to effectively serialize them in XML, and vice versa, using a mapping that guarantees object-level coherence. In terms of runtime storage overhead of the approach, every runtime pointer in a graph is stored in a hash table for comparison to detect multi-referenced objects and graph cycles. The serialization approach does not require the augmentation of application types with meta information such as tags or run-time type references.

Consider for example the type declarations shown in Figure 4 and the data model derived from it. The model shows the pointer edges necessitating the placement of pointer-checking code in the serializer and deserializer routines. For example, when serializing a `Node` instance, two addresses have to be checked for pointer references: the `Node` instance address and the `val` member address. Since these could be identical, type disambiguation is necessary to distinguish the references by keeping track of pointer's target types at run time. Also all integer nodes must be checked in this case for inbound pointer edges, e.g. the `val` member and the `SSN` nodes. However, no pointer checks are required for floats, e.g. the `num` member. Note that `val` is embedded within `Node`, which means that it must be annotated with an XML `id` attribute in the serialized stream.

The *soapcpp2* compiler generates optimized serialization code based on the model. These runtime serialization algorithms use two phases. The first phase determines which data components belong to the data structure and which pointers are used to reference them. This phase is only relevant for pointer-based data

Source Code	**Data Model**

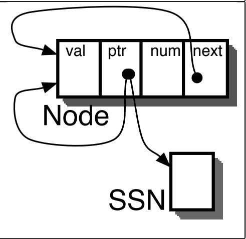
typedef int SSN;
struct Node
{
  int val;
  int *ptr;
  float num;
  struct Node *next;
};
	

Fig. 4. Data Structure Declarations and Plausible Data Model

structures. The second phase emits the XML encoded form of the entire data structure, with all sub-components of the structure serialized recursively.

5.2 Serialization Phase 1

Phase 1 traverses the runtime data structure graph by visiting each node and by following the pointer references to all sub-nodes recursively. For each pointer that was followed to a sub-node, it stores the pointer's target address in a hash table together with an identification of the data type referenced by the pointer. The hash table key is a triple of $\langle$*PtrLoc*,*PtrType*,*PtrSize*$\rangle$, where *PtrLoc* is the pointer's target address, *PtrType* is the type of data referenced by the pointer, and *PtrSize* is the size of the object in bytes, which is statically determined with sizeof. The *PtrSize* of dynamic arrays is computed from the array size and element size, where dynamic arrays are defined with a struct/class containing a pointer field and size field. Node pointers are only followed through to visit sub-nodes when the key $\langle$*PtrLoc*,*PtrType*,*PtrSize*$\rangle$ is not already contained in the hash table. When the key is already contained in the hash table, then the hash table entry is marked *RefType*=multi to indicate that a multi-referenced sub-node has been found. Entries in the hash table are marked *RefType*=embedded when a data element that is pointed to is embedded in larger structure, such as a field of a struct or class, or an element of an array. It is noteworthy to mention that a hash table entry is created at run time only for each pointer in a pointer-based data structure. No additional space is required to serialize non-pointer-based structures.

5.3 Serialization Phase 2

Phase 2 emits the data in XML by visiting each node in the data structure graph and by following the pointer references to the sub-nodes for recursive serialization. Multi-referenced nodes (those whose hash table entry is marked *RefType*=multi) are serialized as multi-referenced objects referenced with `id` and `ref` in the XML stream. The serialization settings can be set to SOAP 1.1/1.2 RPC encoding. Special care is taken to serialize data elements that are embedded within structs or arrays (that is, the hash table entry for these elements are marked *RefType*=embedded) to preserve object graph coherence. The embedded property of data elements affects the placement of `id` and `ref` attributes in the encoded form of XML.

The two-phase serialization is illustrated with the example data structure shown in Figure 5 based on the data type declarations shown in Figure 4. The structure consists of three nodes, two structs A and B, and a node C that contains a single integer value. The serialization starts at the root struct stored at address A. The first phase consists of a pass over the entire data structure to collect the properties of the pointers used in the data structure graph and to store these in the runtime points-to table shown in Table 3.

Each entry has a unique index *ID*, a hash table key $\langle$*PtrLoc*,*PtrType*,*PtrSize*$\rangle$ consisting of a target pointer address *PtrLoc*, pointer type *PtrType*, and size

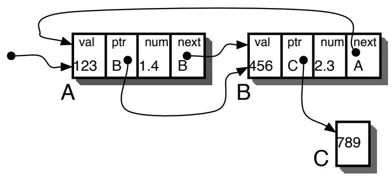
Source Code	**Runtime Object Graph**	**Serialized XML**
struct Node A, B;	 val ptr num next 123 B 1.4 B A val ptr num next 456 C 2.3 A B 789 C	`<Node id="_1">`
SSN C;		`  <val>123</val>`
C = 789;		`  <ptr ref="#_2"/>`
A.val = 123;		`  <num>1.4</num>`
A.ptr = &B.val;		`  <next>`
A.num = 1.4;		`    <val id="_2">456</val>`
A.next = &B;		`    <ptr>789</ptr>`
B.val = 456;		`    <num>2.3</ptr>`
B.ptr = &C;		`    <next ref="#_1"/>`
B.num = 2.3;		`  </next>`
B.next = &A;		`</Node>`

Fig. 5. Source Code and Resulting Object Graph in XML

PtrSize of the object pointed to, an indication of the number of references *Ptr-Count* made to this target address and the type of the reference *RefType*, which is either “single”, “multi”, or “embedded”.

The serialized XML output is shown in Figure 5. The root node is serialized with `id="_1"`, because it is multi-referenced. The second struct at location B is serialized in XML as a nested element of the first node struct, because it has only a single reference. Note that the ptr field in the first struct points to the val field in the second struct, which is stored at B. Because the val field is embedded within a struct, the ptr is serialized with a *forward* pointing `ref="#_2"` attribute. This ensures that the receiving side can decode the XML and backpatch the `ptr` pointer field to point to the val field *after* the contents of the second struct are decoded. The ptr field in the second struct points to a single-referenced integer located at C. The XML serialized value is placed directly in an `ptr` element without an `ref` attribute, because it is a single reference.

The gSOAP *soapcpp2*-generated deserialization routines decode the contents to reconstruct the original data structure graph. The parser takes special care in handling the `id` and `ref` attributes that resemble pointers. When the data structure is reconstructed, temporarily unresolved references are kept in a hash table. When the target objects of the references have been parsed and the data is allocated in memory, the unresolved references are replaced by pointers. In

ID	*PtrLoc*	*PtrType*	*PtrSize*	*PtrCount*	*RefType*
1	A	Node	16	2	multi
2	B	int	4	1	embedded
3	B	Node	16	1	single
4	C	int	4	1	single

Table 3. Runtime Table for Points-To Analysis

effect, the unresolved pointers in the `Node` structures are back-patched with pointer values to link the separate parts of the (cyclic) graph structure together.

5.4 Handling Polymorphism

Pointers to polymorphic objects, i.e. instances of derived classes, are serialized using C++ dynamic binding of the objects pointed to. The gSOAP *soapcpp2* compiler augments classes with virtual serializers to enable pointer-based polymorphism based on single inheritance. The static analysis determines whether pointers to instances need to be treated differently. If so, serialization and deserialization routines are generated for runtime encoding and decoding of object graphs that have pointers to derived instances, which requires the XML serialization of these instances with schema-compliant `xsi:type` attributes to hold type information so that the decoders can accurately reconstruct the object graphs.

6 Results

The use of XML as a data transport protocol comes at a price and the serialization of internal application data to XML can be subject to performance degradation. However, our compiled approach exploits the well-formed properties of XML to construct schema-specific predictive XML parsers to decode XML directly into application data without the need for an additional XML conversion layer. This technique effectively compresses the Web services communication stack, as shown in Figure 6 by combining XML parsing, XML validation, and data conversion into one stage. This is made possible by the fact that XML schemas describe context-free languages on tokens that are XML elements, attributes, and character data content.

The performance in the number of roundtrip messages achieved per second of a gSOAP benchmark client and server application is shown in Figure 7. The performance of a round-trip message containing a struct of size 1.2K is shown over HTTP 1.1 without using keep-alive connection persistence. The serialization

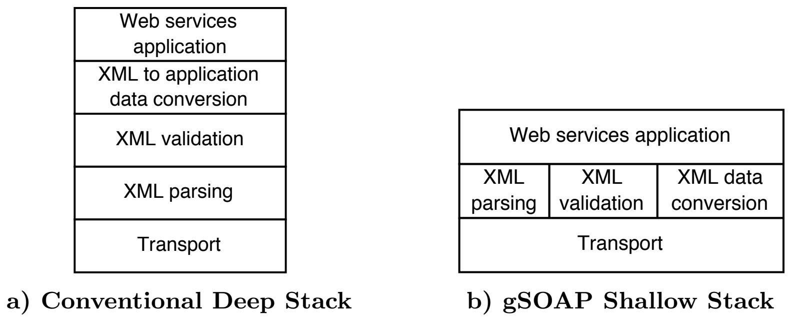
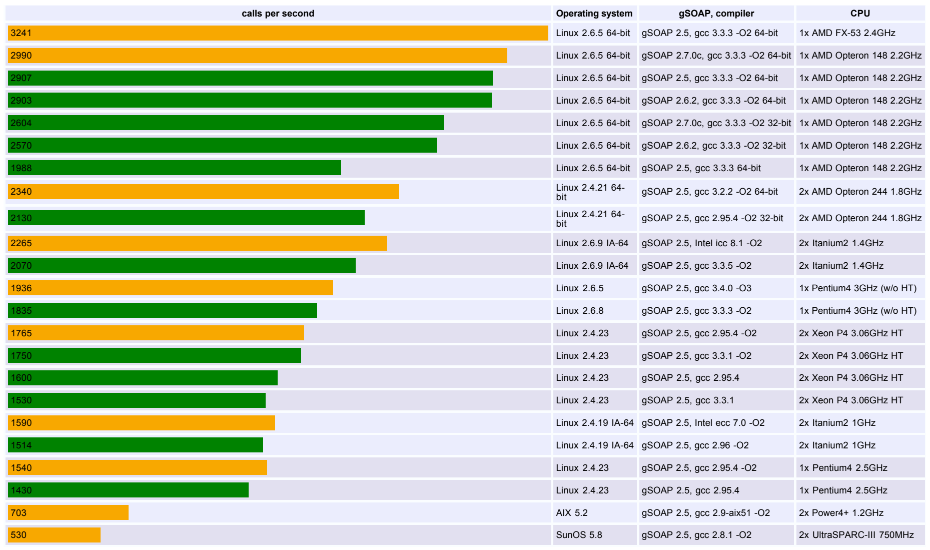


Fig. 6. Compressing the Web Services Stack



calls per second	Operating system	gSOAP, compiler	CPU
3241	Linux 2.6.5 64-bit	gSOAP 2.5, gcc 3.3.3 -O2 64-bit	1x AMD FX-53 2.4GHz
2990	Linux 2.6.5 64-bit	gSOAP 2.7.0c, gcc 3.3.3 -O2 64-bit	1x AMD Opteron 148 2.2GHz
2907	Linux 2.6.5 64-bit	gSOAP 2.5, gcc 3.3.3 -O2 64-bit	1x AMD Opteron 148 2.2GHz
2903	Linux 2.6.5 64-bit	gSOAP 2.6.2, gcc 3.3.3 -O2 64-bit	1x AMD Opteron 148 2.2GHz
2604	Linux 2.6.5 64-bit	gSOAP 2.7.0c, gcc 3.3.3 -O2 32-bit	1x AMD Opteron 148 2.2GHz
2570	Linux 2.6.5 64-bit	gSOAP 2.6.2, gcc 3.3.3 -O2 32-bit	1x AMD Opteron 148 2.2GHz
1988	Linux 2.6.5 64-bit	gSOAP 2.5, gcc 3.3.3 64-bit	1x AMD Opteron 148 2.2GHz
2340	Linux 2.4.21 64-bit	gSOAP 2.5, gcc 3.2.2 -O2 64-bit	2x AMD Opteron 244 1.8GHz
2130	Linux 2.4.21 64-bit	gSOAP 2.5, gcc 2.95.4 -O2 32-bit	2x AMD Opteron 244 1.8GHz
2265	Linux 2.6.9 IA-64	gSOAP 2.5, Intel icc 8.1 -O2	2x Itanium2 1.4GHz
2070	Linux 2.6.9 IA-64	gSOAP 2.5, gcc 3.3.5 -O2	2x Itanium2 1.4GHz
1936	Linux 2.6.5	gSOAP 2.5, gcc 3.4.0 -O3	1x Pentium4 3GHz (w/o HT)
1835	Linux 2.6.8	gSOAP 2.5, gcc 3.3.3 -O2	1x Pentium4 3GHz (w/o HT)
1765	Linux 2.4.23	gSOAP 2.5, gcc 2.95.4 -O2	2x Xeon P4 3.06GHz HT
1750	Linux 2.4.23	gSOAP 2.5, gcc 3.3.1 -O2	2x Xeon P4 3.06GHz HT
1600	Linux 2.4.23	gSOAP 2.5, gcc 2.95.4	2x Xeon P4 3.06GHz HT
1530	Linux 2.4.23	gSOAP 2.5, gcc 3.3.1	2x Xeon P4 3.06GHz HT
1590	Linux 2.4.19 IA-64	gSOAP 2.5, Intel ecc 7.0 -O2	2x Itanium2 1GHz
1514	Linux 2.4.19 IA-64	gSOAP 2.5, gcc 2.96 -O2	2x Itanium2 1GHz
1540	Linux 2.4.23	gSOAP 2.5, gcc 2.95.4 -O2	1x Pentium4 2.5GHz
1430	Linux 2.4.23	gSOAP 2.5, gcc 2.95.4	1x Pentium4 2.5GHz
703	AIX 5.2	gSOAP 2.5, gcc 2.9-aix51 -O2	2x Power4+ 1.2GHz
530	SunOS 5.8	gSOAP 2.5, gcc 2.8.1 -O2	2x UltraSPARC-III 750MHz

Fig. 7. Performance Results in Number of Roundtrip Messages per Second of a Benchmark Client/Server Application on Various Platforms

algorithms are efficient, because runtime pointer checks for serializing object graphs are restricted to pointer-based objects only as determined from the static data model.

7 Conclusions

Object-level coherence can be achieved with specialized serialization algorithms to ensure data structures and object graphs preserve their structure when passed from one XML Web service application to another or when stored and retrieved in XML format. This paper discussed the mapping issues and presented algorithms for serializing C and C++ data structures in XML, while providing object-level coherence and performance guarantees. Object graphs are serialized using carefully placed `id` and `ref` attributes. This approach works well for SOAP RPC encoding but is less suitable for the more expressive document literal style. Unless the Web services community converges on the XML ID standard and a universal reference mechanism, coherence problems cannot be adequately addressed.

8 Acknowledgments

The authors would like to thank Martin Kuba from the Supercomputing Center, Institute of Computer Science, Masaryk University, Brno, Czechoslovakia, for testing the performance of gSOAP on a wide range of machines.

References

1. Apache Foundation. Apache axis project. Available from http://ws.apache.org/axis.
2. A. Bakker, M. van Steen, and A. S. Tanenbaum. From remote objects to physically distributed objects. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, 1999.
3. H. E. Bal and M. F. Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 28, pages 162–177, New York, NY, 1993. ACM Press.
4. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. PVM3 users's guide and referential manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Sept. 1994.
5. R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages*, pages 1–15, 1996.
6. W. Gropp, R. Lusk, and A. Skjellum. *Using MPI*, 1994.
7. IETF. XDR specification. www.ietf.org/rfc/rfc1014.txt.
8. P. Kulchenko. SOAP::Lite for Perl. Available from http://www.soaplite.com.
9. Microsoft. .NET framework. Available from http://www.microsoft.com/net.
10. OMG. CORBA component model. http://www.omg.org.
11. B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.
12. Sun Microsystems. Java programming language. Available from http://java.sun.com.
13. R. van Engelen and K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid*, pages 128–135, Berlin, Germany, May 2002.
14. R. van Engelen, G. Gupta, and S. Pant. Developing web services for C and C++. *IEEE Internet Computing*, pages 53–61, March 2003.
15. W3C. SOAP 1.1 and 1.2 specifications. Available from www.w3c.org.
16. W3C. WSDL web services description language specification. Available from www.w3c.org.
17. W3C. XML ID 1.0 specification. Available from www.w3c.org.
18. W3C. XML schema specification. Available from www.w3c.org.
19. WS-I. Basic profile bp1.0a. Available from www.ws-i.org.