

# Differential Deserialization for Optimized SOAP Performance

Nayef Abu-Ghazaleh and Michael J. Lewis  
Grid Computing Research Laboratory  
Department of Computer Science  
State University of New York (SUNY)  
Binghamton, NY 13902  
{nayef, mlewis}@cs.binghamton.edu

## ABSTRACT

SOAP, a simple, robust, and extensible protocol for the exchange of messages, is the most widely used communication protocol in the Web services model. SOAP's XML-based message format hinders its performance, thus making it unsuitable in high-performance scientific applications. The deserialization of SOAP messages, which includes processing of XML data and conversion of strings to in-memory data types, is the major performance bottleneck in a SOAP message exchange. This paper presents and evaluates a new optimization technique for removing this bottleneck. This technique, called differential deserialization (DDS), exploits the similarities between incoming messages to reduce deserialization time. Differential deserialization is fully SOAP-compliant and requires no changes to a SOAP client. A performance study demonstrates that DDS can result in a significant performance improvement for some Web services<sup>1</sup>.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*Parsing, Optimization*; H.3.5 [Information Storage and Retrieval]: Online Information Services—*Web-based services*; C.2.2 [Computer Systems Organization]: Computer-Communication Networks—*Network Protocols*

## General Terms

Algorithms, Languages, Performance

## Keywords

Deserialization, Performance, SOAP, Web Services, XML Parsing

<sup>1</sup>This research is supported by NSF Career Award ACI-0133838 and DOE Grant DE-FG02-02ER25526, and NSF Award CNS-0454298.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC/05 November 12-18, 2005, Seattle, Washington, USA  
Copyright 2005 ACM 1-59593-061-2/05/11 ...\$5.00.

## 1. INTRODUCTION

Web services provide standard protocols and data formats for representing and calling services in a wide area environment. The Web Services Description Language (WSDL) specifies and describes services, and the XML-based SOAP message format specifies a means for invoking them over some underlying protocol, usually HTTP. SOAP is simple, expressive, extensible, and language independent, thereby enabling interoperability between clients and servers implemented with different languages and toolkits. These attractive characteristics led SOAP and other Web services protocols to be adopted as Grid computing standards [10].

Unfortunately, because SOAP is an ASCII and XML based format, converting between SOAP and in-memory application data representations can be expensive. Data serialization and deserialization can quickly become the communication—and indeed the application—bottleneck, especially for high performance scientific Grid applications [5].

In previous work [4, 3, 2], we addressed this problem on the sender's side, by avoiding serializing entire messages. The sender side of our SOAP implementation, called bSOAP, saves copies of outgoing message buffers, and tracks when the client code makes changes to the data sent in the messages. Only the changes are reconverted and rewritten into the outgoing message buffer template. The rest of the template remains unchanged from the previous send, avoiding serialization for that portion of the message. Our performance study indicates that this technique, called *differential serialization* (DS), can result in significant performance improvement when clients send sequences of similar messages with similar structure and content.

In this paper, we describe the design and implementation of differential serialization's analogue on the server side, which we call *differential deserialization* (DDS). The idea is to avoid fully deserializing each message in an incoming stream of similar messages. Differential deserialization gets its name because the server-side parser deserializes the differences between an incoming message and a previous one. DS and DDS are completely separate and independent ideas and implementations; neither depends on the other for any portion of the performance enhancements we report; the two techniques represent very different realizations of the same high level idea; DS for sending SOAP data, and DDS for receiving it.

We believe that in general, DDS is an even more promising

```

POST / HTTP/1.1
Host: api.google.com
User-Agent: gSOAP/2.7
Content-Type: text/xml; charset=utf-8
Content-Length: 664
Connection: close
SOAPAction: "urn:GoogleSearchAction"

```

```

<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xm
lns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xm
lns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xm
lns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:x
sd="http://www.w3.org/2001/XMLSchema" xmlns:api="urn:Google
Search"><SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schem
as.xmlsoap.org/soap/encoding/"><api:doGoogleSearch><key>XXX
XX</key><q>Binghamton Grid Computing</q><start>0</start><ma
xResults>10</maxResults><filter>true</filter><restrict></re
strict><safeSearch>false</safeSearch><lr></lr><ie>latin1</i
e><oe>latin1</oe></api:doGoogleSearch></SOAP-ENV:Body></SOA
P-ENV:Envelope>

```

(a) A search for “Binghamton Grid Computing”

```

POST / HTTP/1.1
Host: api.google.com
User-Agent: gSOAP/2.7
Content-Type: text/xml; charset=utf-8
Content-Length: 667
Connection: close
SOAPAction: "urn:GoogleSearchAction"

```

```

<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xm
lns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xm
lns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xm
lns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:x
sd="http://www.w3.org/2001/XMLSchema" xmlns:api="urn:Google
Search"><SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schem
as.xmlsoap.org/soap/encoding/"><api:doGoogleSearch><key>XXX
XX</key><q>Differential Deserialization</q><start>0</start>
<maxResults>10</maxResults><filter>true</filter><restrict><
/restrict><safeSearch>false</safeSearch><lr></lr><ie>latin1
</ie><oe>latin1</oe></api:doGoogleSearch></SOAP-ENV:Body></
SOAP-ENV:Envelope>

```

(b) A search for “Differential Deserialization”

Figure 1: Two similar gSOAP-generated SOAP messages to the Google doGoogleSearch Web Service.

implementation optimization technique than DS, because we feel it will be more applicable. DS only works if the same client sends a stream of similar messages. DDS, on the other hand, can avoid deserialization of similar messages sent by multiple different clients. Because the client-server model implies interaction between many clients and the same server, and because the speed of the server is more often the determining factor for performance, DDS has the potential to be more effective.

To illustrate the opportunity for performance enhancement, consider the two sample SOAP messages depicted in Figure 1. Each message arrives at a Google Web services container to invoke a simple search. The message in Figure 1(a) searches for “Binghamton Grid Computing,” whereas the one in Figure 1(b) searches for “Differential Deserialization.” Aside from the search string (embedded five lines from the bottom of the message), the only other difference in SOAP contents is the value of the “Content Length” field in the header (664 in Figure 1(a), 667 in Figure 1(b)). If these two messages arrived in succession, the SOAP server side would ordinarily have to parse and deserialize both of them completely and independently of one another. Ideally, the server would be able to save time by recognizing that the messages are largely similar, and that only the search string needs to be deserialized and reconverted to the server’s memory.

Clearly, the Google Web service can expect to receive many such similar messages, that differ only in their search string. Google’s SOAP messages are likely to resemble one another because they are primarily generated from a client Web interface, and they all invoke the Google *doGoogleSearch* service. One factor that may limit the opportunity for performance enhancement is that different SOAP toolkits will generate equally valid SOAP messages that differ cosmetically (due to spacing, tag names, etc.). Therefore, it might be necessary for a DDS-enabled server to differentiate between messages coming in from different toolkits; to consider all gSOAP-generated messages separately from Apache SOAP messages, for example. The relatively small

number of SOAP toolkits in prominent use, along with the common practice of most SOAP toolkits identifying themselves in message headers, makes this possible.

DDS works by periodically checkpointing the state of the SOAP deserializer that reads and deserializes incoming SOAP messages, and computing checksums of SOAP message portions. The checksums can be compared against those of the corresponding message portion in the previous messages, to identify with high probability that the message portions match one another exactly. When message portions match, the deserializer can avoid duplicating the work of parsing and converting the SOAP message contents in that region. To do so, the deserializer runs in one of two different modes, which we call *regular* mode and *fast* mode, and can switch back and forth between the two, as appropriate. In regular mode, the deserializer reads and processes all SOAP tags and message contents, as a normal SOAP deserializer would, creating checkpoints and corresponding message portion checksums along the way. In fast mode, the deserializer considers the sequence of checksums (each corresponding to disjoint portions of the message), and compares them against the sequence of checksums associated with the most recently received message for the same service.

If all of the checksums match (the best, but unrealistic case where the next message is identical to the previous one), then the normal cost of deserializing is replaced by the cost of computing and comparing checksums, which is generally significantly faster. When a checksum mismatch occurs, signalling a difference in the incoming message from the corresponding portion of the previous message, bSOAP switches from fast mode to regular parsing, and reads and converts that message portion’s contents as it would otherwise have had to (without the DDS optimization). bSOAP can switch back to processing the message in fast mode if and when it recognizes that the parser state is the same as one that has been saved in a checkpoint; this is described in Section 3.3 below. If the next message is different from the previous message in *all* message portions (the worst case scenario), then a DDS-enabled deserializer runs slower than

a normal deserializer, because it does the same work, plus the added work of calculating checksums and creating parser checkpoints.

Thus, the effectiveness of the DDS optimization depends on the the following factors:

- The amount of similarity between consecutive messages in an incoming message stream, which determines the percentage of time the deserializer spends in fast mode
- How much faster it is to traverse an incoming message in fast mode than in regular parsing mode with full parsing and deserialization
- How quickly the parser can recognize (i) the need to switch from fast mode to regular mode (which depends primarily on checksum calculation and comparison times), and (ii) the opportunity to switch back to fast mode from regular mode
- The overhead of creating checkpoints, and calculating and comparing checksums

Creating many frequent checkpoints can increase the percentage of time spent in fast mode, at the expense of checkpoint creation time, checkpoint memory consumption, and checksum calculation and comparison times. The performance study of Section 4 characterizes these tradeoffs for sample message streams. The study reveals that the opportunity for performance improvement through differential deserialization is significant. In particular, when sequences of identical messages are transmitted, deserialization speedup is at least 3.8, and can reach 41. For the more realistic case where some parts of messages change, DDS still decreases deserialization time when less than half of the message changes. Specific results depend on where the changes occur in the message, and the parameters of bSOAP's DDS configuration, as detailed in Section 4.

This paper presents initial results that demonstrate that differential deserialization has the potential to significantly increase performance of some Web services. Section 2 motivates the work, provides some background information on SOAP and XML parsing, and identifies related work. Section 3 describes in more detail the design and implementation of bSOAP's fully-functional server-side parser and deserializer, discussing fast and regular modes, the creation of checkpoints, switching back and forth between modes, and memory and error-handling issues. Section 4 presents the results of a performance study that characterizes DDS overhead and identifies the circumstances under which it represents an effective optimization technique. Section 5 summarizes the contributions of this paper, and describes several areas for future improvements.

## 2. BACKGROUND AND RELATED WORK

In this section, we describe other XML and SOAP processing techniques for optimized parsing and faster deserialization. The goals of this section are twofold. First, we characterize our deserializer with respect to the most popular approaches, and describe our regular parsing mode. This provides the necessary groundwork for understanding the DDS performance enhancements described in Section 3. Second, we compare our optimization technique with other similar

approaches, and with other approaches that share our goal of speeding up end-to-end SOAP performance.

### 2.1 XML parsing and SOAP deserialization

Deserialization converts XML streams in wire-format to objects in memory. This process consists of four steps; reading the XML stream from the socket layer into a buffer, parsing the XML stream into tokens, converting data types in ASCII to binary formats, and populating data structures with the received values. The widely used paradigms for parsing XML documents include Document Object Model (DOM) [17], Simple API for XML (SAX) [7], and XML Pull Parser (XPP) [13]. Other recent research on high performance parsers investigates the advantages of using schema specific parsers [6].

The DOM model maps the XML document into a tree representation in memory. This allows the document to be easily traversed and modified. However, for large documents, DOM parsing can be memory intensive. In contrast, SAX parsing never stores the entire XML document in memory. Instead, it emits events for all the document's elements and tags. For large static documents, SAX is preferable to DOM. SAX is also often used when only a few specific elements of an XML document need to be extracted and processed. *Pull parsing*, employed by the XPP [13] parser, is specialized for parsing payloads in which elements are processed in succession, and no element needs to be revisited. XPP provides the added feature of incrementally building a partial XML Infoset tree in memory. As a result, applications can process the document as it is being parsed, without waiting for the entire document to be parsed.

DOM, SAX, and XPP require two passes through the XML document; the parser tokenizes the document in the first pass, and the application processes the content in the second. An STL `map` is typically used to compare each tag retrieved by the parser with the one that is expected. Results [5] show that a `trie` data structure, which has  $O(1)$  lookups as opposed to  $O(\lg n)$  for STL `map`, can provide significant performance improvement for matching tags that appear repeatedly.

bSOAP contains a pull parser that incorporates many optimizations found in other parsers, including avoiding unnecessary string copies. One unique feature of bSOAP's XML parser is that it contains parsing routines that are optimized for the case where the expected content of the message is known a priori (e.g., for SOAP messages). In this case, the parser requires only a single pass over the document content.

### 2.2 Overview of bSOAP's deserializer

bSOAP's deserializer uses a schema to drive the deserialization process. A *schema* in bSOAP is a collection of data structures and supporting deserialization routines that define the acceptable format of a SOAP message, as well as the rules for converting a SOAP message to application-specific objects. Specifically, schemas are represented using instances of four *schema classes*, each of which has an associated deserialization routine. These classes are *SimpleElement*, *ComplexElement*, *ArrayElement*, and *MethodElement*, respectively. A planned fifth class, *UnionElement*, will support the XML schema's union types. This representation of schemas is primarily based on the SOAP data model<sup>2</sup>; *SimpleElements* correspond to SOAP's simple types, *Com-*

<sup>2</sup>bSOAP currently supports SOAP 1.1. Support for other

*plexElements* correspond to SOAP’s compound types, and *ArrayElements* correspond to SOAP arrays.

The information stored in a schema class object generally depends on the type and instance of the application object(s) to which the schema object maps. For example, an *ArrayElement* object contains a reference to the schema class object corresponding to the type of the array elements. Similarly, a *ComplexElement* object contains references to schema class objects corresponding to the subtypes constituting the compound type. In addition, a schema object of *ArrayElement*, *ComplexElement*, or *SimpleElement* may contain the expected name of the corresponding XML element in the SOAP message.

To make this more concrete, consider, for example, how bSOAP processes a C++ doubles array. An *ArrayElement* object contains a reference to a *SimpleElement* object. When a message containing an array of ten doubles is deserialized, the *ArrayElement*’s deserializer calls XML parsing routines<sup>3</sup> to verify that the relevant part of the XML message is valid per the SOAP data model, and to extract information about the array (including its size). After that, it calls the *SimpleElement*’s deserializer once for each of the ten doubles, which, in turn, does its own validation checks and handles the text-to-double conversions.

The application must construct a bSOAP schema and pass it to bSOAP. It can do this at run-time (e.g., from a WSDL [9] document) or at compile-time. bSOAP calls an application-defined callback routine, passing it the name of the back-end method. The application returns the corresponding bSOAP schema (if possible) to bSOAP, in the form of a *MethodElement* object. A *MethodElement* object contains all the information bSOAP needs to deserialize the message. This includes the schema class objects corresponding to the method’s parameters and any dynamically-allocated memory from previous deserializations (i.e., memory containing the application’s objects and checkpoint data).

Everything described to this point is independent of the DDS optimization, as it describes bSOAP’s operation in regular parsing mode (without creating checkpoints). It is described here to provide context for the description of DDS, which follows in Section 3.

### 2.3 Other optimization techniques

In our HPDC 2004 paper [4], we introduced the term “differential deserialization,” as future work. The same term has since been used to describe the most closely related work in this area [14]. The authors build on their own work of efficiently processing XML documents by comparing byte sequences with those from previously processed documents [15], which are maintained in a DFA. Their version of differential deserialization optimizes processing time by recycling in-memory objects on the receiver side of a SOAP communication pair, thereby eliminating the expensive step of creating these objects in memory. The technique achieves performance improvement of up to 288%. Our work employs a different technique toward the same goal, based on checkpoints and full parser state checkpointing.

data models can be achieved by defining the appropriate schema classes and associated deserialization routines.

<sup>3</sup>bSOAP has a built-in XML parser, with parsing routines optimized for the case where the valid XML content is known before parsing.

For other SOAP and XML optimization techniques, our own work on differential serialization [4, 3, 2], as mentioned in Section 1, represents a sender-side optimization that shares with DDS the approach of optimizing SOAP end-to-end performance by avoiding (de)serialization for streams of similar messages. Devaram et al. [8] describe “parameterized client-side caching” of messages in files. Entire messages can be sent as is, and partial caching allows the client to reuse cached messages and change a few of the parameters for subsequent sends. This work is more similar to DS than DDS in that it is a client-side optimization.

Chiu et al. [5] address SOAP performance bottlenecks by using trie data to reduce the number of comparisons for XML tags. Another optimization is chunking and streaming of messages. gSOAP [16] also provides this feature, in addition to compression, routing, and the use of optimized XML data representations using XML schema extensibility. These techniques are complementary to differential deserialization.

The SOAP specification allows the use of “multi-ref accessors”—identifiers that refer to previously serialized instances of specific elements of the SOAP call. Multi-ref accessors can be included within messages that bSOAP deserializes, without undermining the DDS optimization.

The SOAP community has suggested several different specifications that would standardize SOAP binary formats, including base64 encoding, DIME [12] and BEEP [1]. While these techniques do achieve performance gains, they reduce the simplicity and universality of SOAP, the characteristic that makes it interoperable and attractive.

## 3. DESIGN AND IMPLEMENTATION

This section describes bSOAP’s DDS design and implementation. Section 3.1 begins with a description of how checkpoints are created. Section 3.2 then describes how they are used, by discussing bSOAP’s *fast* parsing mode. Section 3.3 describes the process used to identify when bSOAP’s deserializer can switch into fast mode, which requires identifying when a saved parser checkpoint matches the parser’s current state, while it is processing an incoming SOAP message. This is accomplished by a process we call *progressive matching*, using a *matching stack* data structure. When the deserializer detects differences in arriving messages, saved checkpoints may become unusable, because they do not reflect a state that could possibly be arrived at for the current message. We call these *stale checkpoints*, and describe the process of removing them, and replacing them with valid ones, in Section 3.4. Finally, Sections 3.5 and 3.6 describe memory management and error handling issues, respectively.

### 3.1 Creating checkpoints

Checkpointing is enabled while bSOAP is operating in regular mode. That is, bSOAP may create checkpoints the first time a message to a particular service is seen, and whenever it must switch from fast mode to regular mode.

bSOAP creates the first checkpoint just after parsing the start tag of the XML element carrying the name of the back-end method. This allows for switching to fast mode immediately after getting a *MethodElement* object from the application. The initial few bytes before the start tag carrying the name of the back-end method are not checkpointed; this allows DDS to be independent of header extensions, which are likely to change from message to message.

Field	Description
Program stack	The portion of the program stack used by the deserializer.
Matching stack	Describes the application object being deserialized.
Strings stack	Contains the strings referenced by the deserializer.
Namespace aliases stack	Contains the namespace alias mappings hash table's nodes.
Memory blocks stack	Contains pointers to active memory blocks.
Memory blocks list	A linked list of any memory blocks created while deserializing the checkpoint's message portion.
Pointers	A couple of pointers that may be used during parsing.
Size	The number of bytes in the checkpoint's message portion.
Checksum	Checksum for the checkpoint's message portion.

**Table 1: Description of state saved in a checkpoint**

Thereafter, bSOAP uses the number of bytes processed to determine the point in the message at which to create a checkpoint. bSOAP can be configured to set the frequency with which checkpoints are created. Requesting more checkpoints increases overhead in regular parsing mode, but can lead to a larger percentage of time spent in fast deserialization mode, a tradeoff that is explored in the performance study of Section 4. Checkpoints are arranged in a linked list, according to the order of their creation.

Table 1 briefly describes a checkpoint's contents, including the state of the deserializer/parser<sup>4</sup>. For all but the first checkpoint, we define a checkpoint's *message portion* to be the portion of the message beginning just after the previous checkpoint was created, and just before the checkpoint is created. Thus, checkpoints are created on message portion boundaries.

bSOAP disables checkpointing just before parsing the end tag of the XML element carrying the name of the back-end method. The state of the parser after deserializing the final message portion—the portion of the message processed just after the last checkpoint is created—is never checkpointed (because there would be no need to revert back to the state corresponding to a message that has been completely parsed); bSOAP just computes its checksum.

### 3.2 Fast mode

In fast mode, bSOAP goes over the checkpoints in order, starting from the checkpoint following the one at which the deserializer switched to fast mode. For each checkpoint, the deserializer computes the checksum of the corresponding portion in the message and compares it to the one stored in the checkpoint. If they match, the deserializer can avoid deserializing this portion of the message, since that work was

<sup>4</sup>Our checkpointing mechanism is generic in the sense that it simply saves and restores parser state. Therefore, this approach will not affect features such as processing Unicode characters, or SOAP's multi-reference values.

already done for the previously arriving message (the contents of the two messages are the same in this region). The deserializer repeats this process for subsequent checkpoints and for the final message portion when the checkpoints list is exhausted.

If a checksum for a message portion mismatches, or bSOAP gets to the end of the message prematurely, it switches to regular mode by restoring the state saved in the checkpoint for the preceding message portion. bSOAP does not restore state if the checksum for the first checked message portion mismatches, as its state would be valid. State restoration involves restoring the contents of the various stacks from the checkpoint and restoring the values of several pointers used during deserialization. Pointers pointing to data in the parsing buffer may also need to have their values adjusted.

### 3.3 Switching to fast mode

An opportunity for switching to fast mode exists when the state of the parser matches that saved in a checkpoint. Two aspects of the state must match for the deserializer to safely switch to fast mode: (1) the application objects that are being deserialized, and (2) the namespace alias mappings. Because the namespace alias mappings can change without a corresponding change in the data structures being deserialized, they must be checked separately.

Consequently, bSOAP goes through a two-step process for detecting when its state matches that in a checkpoint. In the first step, it detects when it gets to a point in the structure of the XML message where a checkpoint has been created. In the second step, it ensures that the namespace alias mappings are the same as those stored in the checkpoint.

A *matching stack* contains all the necessary information for detecting a structural match (step one). Each deserialization routine pushes its own frame onto the matching stack. The first item a deserialization routine pushes on the stack is a reference to the schema object for which it has been invoked. This is used to make the matching process more robust, as each schema object has a unique address in memory. The schema object reference is followed by any necessary information that would allow the deserialization routine to detect (1) whether a particular checkpoint, given the matching stack stored in that checkpoint, could have been created while the deserializer was deserializing the same application objects, and if so, (2) how far it has proceeded into the deserialization of those objects.

For example, a matching stack frame for an *ArrayElement* object might include the number of array elements that have been deserialized. This would allow its deserialization routine to determine whether a particular checkpoint was created while the deserializer was processing an array element it had not yet deserialized (if the number of elements in the checkpoint is larger) or not (if the number of elements in the checkpoint is smaller). Similarly, a matching stack frame for a *ComplexElement* object includes a pointer to a schema class object (corresponding to one of the compound type's subtypes) that is currently being deserialized, allowing its deserialization routine to use similar logic to determine its status with respect to a particular checkpoint.

When the matching stack of the deserializer completely matches that stored in a particular checkpoint, this indicates that the checkpoint was created while the deserializer was deserializing the same application objects that it is cur-

rently deserializing. We refer to the process of determining when the matching stack of a checkpoint and the deserializer match as *progressive matching*, since this process takes place as the deserializer is progressing in the deserialization of the message.

Through the process of progressive matching, the deserializer keeps track of the first checkpoint that is likely to match its state. When it detects that it can no longer have its state match that checkpoint, it picks the very next checkpoint whose state could match, as its new candidate. This checkpoint would be the very first checkpoint that was created after the current application object, which triggered the mismatch, was deserialized. More precisely, this checkpoint would be the first checkpoint that its matching stack does not contain the most-recently pushed stack frame. The previous checkpoint, as well as all checkpoints that fall between it and the new candidate become *stale checkpoints*, since the information they contain is no longer valid with respect to the message currently being deserialized.

### 3.3.1 Switching points

bSOAP can switch to fast mode at predefined points within the structure of the message. The last piece of information pushed on the matching stack describes which of the four *switching points* was encountered before the checkpoint was created.

These four switching points are (1) just before processing a start tag, (2) just before processing an end tag, (3) just after processing a start/end tag, and (4) just before processing character data. We chose these as switching points mainly because it is easy to predict state change, should a checkpoint be created after any of them. For example, if a checkpoint is created while in the middle of processing an attributes list containing new namespace alias definitions, the new mappings can be easily excluded from the check performed afterward, since they can only grow the namespace alias mappings stack.

When a checkpoint is not created at a switching point, a checksum of the bytes between the last encountered switching point and the point at which the checkpoint was created is computed. This checksum is checked before the namespace-alias mappings are checked. For example, if a checkpoint gets created just before processing the character “e” in the start tag `<item>`, a checksum of `<it` would be stored in the checkpoint and checked before switching to fast mode. We refer to the bytes between the point a checkpoint is created and the point the last switching point is encountered as *partially processed content*.

To summarize, bSOAP switches to fast mode when three conditions hold: (1) The matching stacks of both the deserializer and the checkpoint completely match, (2) the checksum of the partially processed content (if any) matches the checksum of the corresponding content in the message, and (3) namespace alias-mappings are the same in both the deserializer and the checkpoint.

## 3.4 Updating checkpoints

Due to changes in the message, some checkpoints may become stale when they no longer contain correct state information. Stale checkpoints must be removed for correct operation of the deserializer. In addition, newly-created checkpoints must be merged into the checkpoints list.

The deserializer removes stale checkpoints immediately af-

ter it detects them while performing progressive matching. On the other hand, the deserializer merges newly-created checkpoints into the checkpoints list only when it leaves regular mode. This can happen due to a switch to fast mode, or the arrival of the end of the message while in regular mode. In the former case, the deserializer inserts any newly-created checkpoints immediately after the checkpoint from which its state was restored in the last switch from fast mode to regular mode<sup>5</sup>. It also updates the checksum and message portion size corresponding to the checkpoint at which it switches to fast mode. When the end of the message arrives while in regular mode, newly-created checkpoints are added to the end of the checkpoint list (from which had all stale checkpoints were removed).

## 3.5 Memory management

During deserialization, memory may have to be dynamically allocated. Each checkpoint has a linked list of memory blocks that were created while deserializing its message portion, arranged according to their creation order. A similar list exists for memory blocks created in the final message portion.

Currently, all memory blocks correspond to arrays. The only exception is the first memory block, which holds the binary values of the parameters. Any memory blocks that may still be used for holding binary values are designated *active*. The first memory block is always active, while other memory blocks remain active as long as bSOAP has not finished deserializing the elements of the corresponding arrays. bSOAP uses a stack to track active memory blocks. Whenever it allocates a new memory block, it pushes an indirect reference to it on the stack. Whenever a memory block is no longer active, it pops its reference off the stack.

When a checkpoint becomes stale and bSOAP destroys it, it also destroys any associated memory blocks. However, to support switching to fast mode even when an array size changes in a new message, the deserializer may reallocate some blocks. Specifically, when an array is encountered in the message, the information on the deserializer’s matching stack and the matching stack of the checkpoint may indicate that the checkpoint was created while the deserializer was deserializing the same array. This indicates that the deserializer can potentially switch to fast mode at that checkpoint; it then reallocates the memory block of the array (which it can get from the memory block’s stack in the checkpoint). The deserializer then also destroys any memory blocks that occur before the reallocated block in the checkpoint’s memory blocks list, since it would not need to reallocate them.

## 3.6 Handling errors

A special flag in a checkpoint is used to force a switch from fast mode to regular mode. When an error occurs in the incoming SOAP message, this flag is set for the checkpoint corresponding to the message portion preceding the one at which the error occurs. This would be the last newly-created checkpoint after the last switch to regular mode or, if no new checkpoints were created, the checkpoint from which the deserializer would have its state restored in the last switch to regular mode. This flag simply makes handling errors more robust. bSOAP could instead create a Null checksum

<sup>5</sup>This checkpoint always exists, since bSOAP attempts to switch to fast mode immediately after it determines the set of checkpoints associated with the back-end method.

for the following checkpoint, to force a checksum mismatch to occur, and therefore, a switch from fast mode to regular mode.

Using checksums to match message portions of consecutive messages is not perfect, of course; it is possible for some changes to go undetected by some checksumming algorithms. However, the probability that the corresponding message portions of two consecutive messages produce the same checksum, without actually matching, will be extremely low (depending on the algorithm). Should it happen, however, bSOAP is likely to report a spurious error as soon it switches to regular mode, since its internal view of the expected format of the message is very likely to be inconsistent with the actual message format. However, should a change in values, rather than structure, go undetected, the back-end method would get passed incorrect values (whatever those values were after the previous deserialization)<sup>6</sup>.

## 4. PERFORMANCE STUDY

This section describes a performance study that characterizes the overhead of DDS checkpointing, and investigates the situations in which it results in performance improvement. We begin by comparing bSOAP’s parsing and deserializing routines to those of gSOAP [16], a widely used efficient SOAP implementation. Section 4.2 presents the results of these tests, performed with bSOAP checkpointing and checksum calculation turned off. Section 4.3 then describes a set of tests designed to characterize the overhead of DDS. We measure the performance of bSOAP with checkpointing and checksum calculation turned on, but operating in regular parsing mode. Thus, the bSOAP deserializer is doing the work required to support DDS, but is not benefiting from the optimization, since it never switches to fast mode.

Section 4.4 demonstrates the opportunity for improvement by studying a “near-best case scenario,” wherein the bSOAP deserializer operates exclusively in fast mode, as opposed to regular mode. This section serves to provide an upper bound on the opportunity for DDS performance gains. The more realistic scenario requires the bSOAP deserializer to switch back and forth between regular and fast parsing modes. Presumably, the performance of the deserializer in this case would fall somewhere between that reported Sections 4.3 and 4.4, because it operates in fast mode part of the time, but also includes the overhead of identifying when to move between the two modes. These results are included in Section 4.5.

### 4.1 Experimental setup

We ran all performance tests on a single Pentium 4 2.4 GHz machine with 512 MB of RAM, and a 60GB 7200 RPM IDE Drive. The tests started a timer, read a sequence of “incoming” SOAP messages from a file, deserialized them, then stopped the timer just before calling the backend method. We averaged 100 iterations of the full test to amortize the initial read from the file—the full messages for all but the first message were typically read directly from cache, allowing us to isolate the deserialization time. We set the parsing

<sup>6</sup>We are assuming that a SOAP client is well-behaving and does not intentionally send invalid SOAP messages. One way to protect against malicious clients is to use a random initial checksum value for each different method or even for each message portion.

buffer size to 32K for both bSOAP and gSOAP, used gSOAP version 2.7.3 (the most recent version at this paper’s submission time), and compiled both gSOAP and bSOAP with gcc 3.4.4 and optimization flags set to “-O3”. gSOAP was also compiled with both WITH\_NOIDREF and WITH\_FAST defined, which disable support for multi-reference values and potentially improve performance, respectively. Unless otherwise noted, for all DDS tests, we used a software-based CRC32 algorithm for computing checksums, which has relatively poor performance; a better performing checksum algorithm could achieve even better performance gains, as shown in Table 2 below.

### 4.2 Baseline deserialization study

This section characterizes bSOAP’s deserialization performance without DDS support turned on, comparing performance to the popular and efficient gSOAP implementation. Both toolkits deserialized messages containing arrays of three different data types: “easy-to-deserialize” doubles, “difficult-to-deserialize” doubles, and integers. We studied three different data types to vary the percentage of time spent converting SOAP values, as opposed to parsing tags and SOAP message structure. Integers represent the fastest conversion routines; integer values in the test messages corresponded to the array index (1, 2, 3, etc.). We define easy-to-deserialize doubles (labelled “Easy Doubles” in Figure 2) to also be integers corresponding to the array indices. We define difficult-to-deserialize doubles (labelled “Hard Doubles”) to be randomly generated doubles, which required an average of 23 characters to encode. Thus, messages containing integers and easy doubles are equivalent in size, while messages containing hard doubles are larger<sup>7</sup>.

Figure 2 shows that bSOAP’s performance, without considering DDS, is comparable to that of gSOAP, for all three types of values we studied, and for messages containing arrays ranging from 1K to 100K elements. Therefore, having shown that we’re applying the DDS optimization to a relatively fast baseline deserialization implementation, we do not include results for gSOAP tests for subsequent parts of this performance study.

### 4.3 DDS overhead

Section 4.2 above reports the results of bSOAP with DDS turned completely off. In particular, the bSOAP deserializer is not even performing the necessary work of creating checkpoints and calculating message portion checksums. This section reports results for bSOAP with DDS checkpointing and checksum creation turned on. We study the same message sizes as in Figure 2 (1K, 10K, 25K, 50K, 75K and 100K array elements). The tests report results for various message portion sizes (which directly influence the number of checkpoints created). In particular, in separate tests, the deserializer was configured to create checkpoints every  $N$  bytes, for  $N$  in {32, 256, 512, 1K, 4K}.

Figure 3 shows that the overhead of DDS checkpointing is small, except when the message portion size is extremely small (in Figure 3, 32 bytes). For message portion sizes of 256 bytes up to 4K bytes, the increase in overhead is

<sup>7</sup>Therefore, the difference in hard doubles and the other two types is due to conversion difficulty as well as the parser touching more bytes. However, the important comparison for the purposes of this study is between gSOAP and bSOAP, not between the different data types.

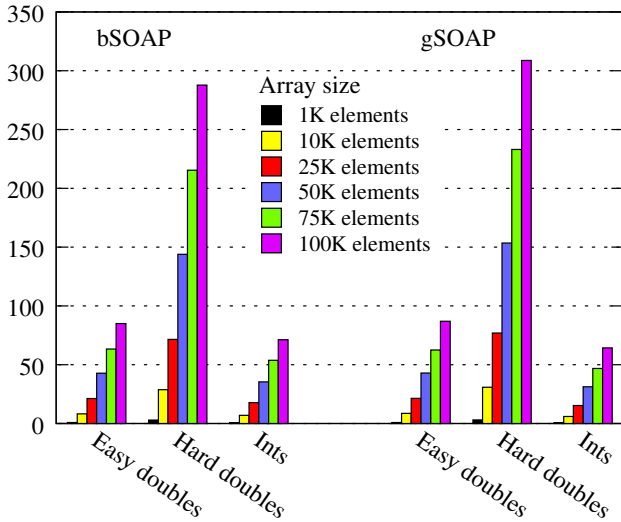


Figure 2: Deserialization time, in milliseconds, for bSOAP and gSOAP, for arrays of integers, and easy and hard doubles of various sizes.

relatively small, especially compared to the opportunity for performance improvement identified in the following subsections.

#### 4.4 Near-best case DDS performance

Whereas the previous subsection characterizes the overhead of DDS, this section identifies the opportunity for performance enhancement. DDS will be most effective when consecutive messages are most similar to one another. Therefore, this test uses sequences of identical messages, with the message portion size kept constant at 4K bytes. This test, then, compares bSOAP’s fast deserialization mode, under various checksum algorithms and when storing and comparing message portions instead of computing their checksums, with bSOAP’s regular deserializer (without DDS support) as well as a regular bSOAP deserializer that uses dummy conversion routines. The dummy conversion routines do not perform string to binary conversions. Thus, the performance of the dummy deserializer serves as an upper bound on the performance of bSOAP’s regular deserializer.

Table 2 shows an extremely significant opportunity to improve performance, especially for large messages. Fast mode deserialization for all cases is much faster than the corresponding regular deserialization. For example, the time to deserialize a SOAP message containing 100K hard-to-deserialize doubles is about 7 ms in fast mode (using the XOR algorithm), and 288 ms in regular mode (a speedup of over 41). For 100K integers, regular mode requires just over 71 ms, and fast mode (XOR algorithm) requires less than 3 ms (a speedup of nearly 25). For this test (containing sequences of identical messages), a *worst-case* speedup corresponds to the dummy deserializer vs. fast-mode deserializer using the slowest checksumming algorithm (CRC32). Even this speedup is  $63.9 / 16.7 \approx 3.8$  for a 100K array of hard doubles. Again, this test simply demonstrates the opportunity and upper bound for performance improvement, not the expected speedup of a realistic message stream.

#### 4.5 DDS dual-mode performance

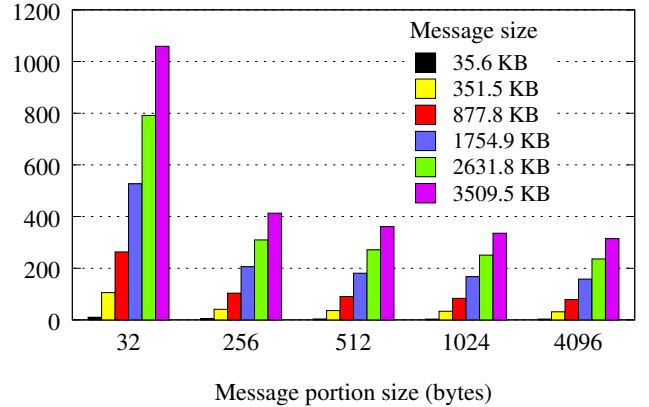


Figure 3: Deserialization time, in milliseconds, for bSOAP when operating completely in regular mode, and for various message portion sizes.

		Number of array elements ( $\times 1000$ )					
		1	10	25	50	75	100
Dummy	Ints	0.57	5.67	13.6	28.2	42.0	56.0
	Easy Doubles	0.56	5.61	14.1	28.6	41.6	55.4
	Hard Doubles	0.70	6.38	16.0	31.5	47.6	63.9
No DDS	Ints	0.67	6.93	17.7	35.4	53.7	71.3
	Easy Doubles	0.83	8.24	21.2	42.7	63.3	85.0
	Hard Doubles	2.89	28.7	71.5	144	215	288
XOR	Ints	0.04	0.21	0.60	1.27	1.84	2.91
	Easy Doubles	0.04	0.29	0.78	1.58	2.37	3.04
	Hard Doubles	0.08	0.73	1.85	3.13	5.37	7.01
Adler32	Ints	0.05	0.26	0.75	1.54	2.26	3.45
	Easy Doubles	0.04	0.34	0.91	1.85	2.77	3.55
	Hard Doubles	0.09	0.84	2.11	3.65	6.12	8.02
CRC32	Ints	0.09	0.67	1.80	3.69	5.45	7.72
	Easy Doubles	0.08	0.75	1.97	3.97	5.97	7.84
	Hard Doubles	0.18	1.72	4.29	8.03	12.9	16.7
Cmp	Ints	0.06	0.51	1.32	2.63	3.89	5.63
	Easy Doubles	0.06	0.57	1.48	2.95	4.42	5.75
	Hard Doubles	0.13	1.32	3.23	5.89	9.47	12.5

Table 2: Fast mode deserialization times, in milliseconds, for various checksumming algorithms and for arrays of ints, easy doubles, and hard doubles of various sizes. Fast mode deserialization times when message portions are fully stored and compared instead of checksums (rows labelled “Cmp”), as well as deserialization times for bSOAP when compiled with no DDS support (rows labelled “No DDS”) and when compiled with no DDS support and dummy conversion routines were used (rows labelled “Dummy”) are also shown.

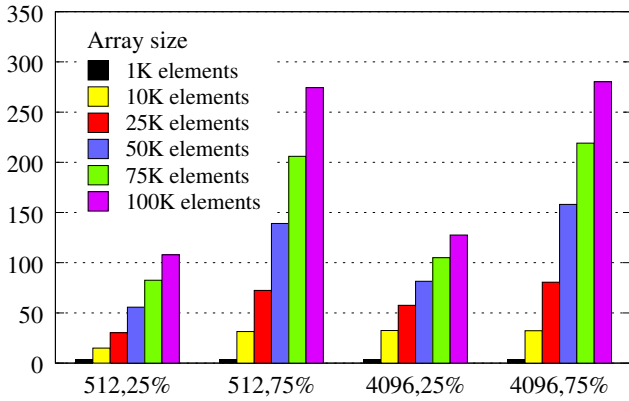


Figure 4: Deserialization time, in milliseconds, for messages divided into 100 partitions ( $T = 100$ ), two different percentages of the values around the center of each partition are changed ( $P = 25$  and  $75$ ), and for two message portion sizes ( $N = 512$  and  $4096$ ). Each  $x$ -axis label denotes the message portion size and the percentage of values changed ( $N, P$ ).

Clearly, sequences of identical messages do not represent a realistic scenario. The test reported in this section varies some of the values in arrays of hard doubles, requiring the deserializer to switch between fast and regular parsing modes. In particular, this test is designed to vary the number of bytes that changed, as well as the location of those changes. The results of the test, therefore, include the cost of identifying (i) the need to switch from fast mode to regular mode (because of a checksum mismatch), and (ii) the opportunity to switch back to fast mode (due to a state match). Results also depend on the percentage of time that can be spent in fast mode, compared to regular mode.

For this test, we divided the arrays into  $T$  partitions for  $T$  in  $\{1, 100, 200, 500\}$  and changed  $P$  percent of each partition for  $P$  in  $\{25, 50, 75, 100\}$ . The changed values were consecutive values clustered around the center of each partition. We used message portion sizes of  $N$  bytes for  $N$  in  $\{32, 512, 1K, 4K\}$ . Thus, when the deserializer is in fast mode, it switches to regular mode upon detecting the first message portion with a changed value, spends some time (generally longer for larger  $P$  and  $N$ ) in regular parsing mode, then possibly switches back to fast mode.

Table 3, located in the Appendix, contains all the deserialization times for hard doubles, for various values of  $T$  (number of partitions),  $N$  (message portion sizes), and  $P$  (percentage of values changed from one message to the next). For clarity of presentation, Figure 4 and Figure 5 depict subsets of Table 3’s data in bar graph form. In particular, Figure 4 plots Table 3’s bold values, and Figure 5 plots values in italics.

As the number of array partitions  $T$  increases, the performance of DDS worsens (especially for larger  $N$ ), as shown by the relative difference between the corresponding bars in Figure 4 and Figure 5. As  $T$  increases, the number of switches from fast mode to regular mode increases. With larger  $N$ , the percentage of time spent in regular mode, potentially re-deserializing unchanged values, increases. For example, when  $P$  is 25,  $N$  is 4096, and the array size is 100K,

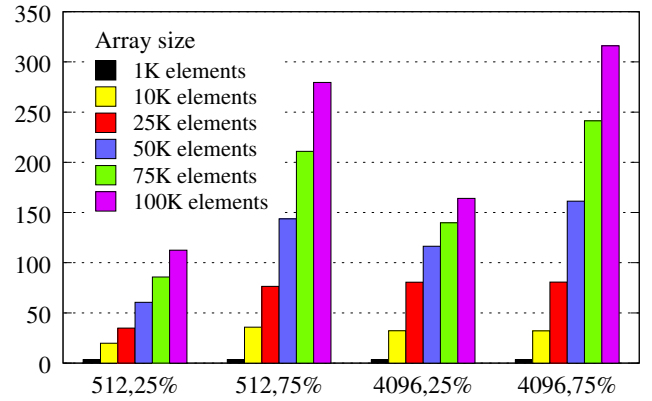


Figure 5: Deserialization time, in milliseconds, for messages divided into 200 partitions ( $T = 200$ ), two different percentages of the values around the center of each partition are changed ( $P = 25$  and  $75$ ), and for two message portion sizes ( $N = 512$  and  $4096$ ). Each  $x$ -axis label denotes the message portion size and the percentage of values changed ( $N, P$ ).

the number of switches from fast mode to regular mode goes up from about 243 (for  $T = 100$ ) to 661 (for  $T = 500$ ). As a result, the percentage of bytes processed in regular mode goes up from about 36.34% to about 81.75%. As an example of how the message portion size  $N$  affects performance, the percentage of bytes processed in regular mode in the case where  $T = 100$ ,  $P = 25$ , and a 100K-element array goes up from about 27.7% ( $N = 1024$ ) to about 36.34% ( $N = 4096$ ).

Checkpointing overhead becomes a major performance bottleneck for smaller values of  $N$  (particularly 32). For example, despite the fact that for all cases where  $P = 25$ , bSOAP processed the least amount of bytes in regular mode when  $N$  is 32 (about 24.44% for  $T = 1$  and a 100K-element array), this message portion size still exhibits the worst performance (however, still better than a regular deserializer).

As  $P$  becomes larger, the effect of both  $T$  and  $N$  becomes smaller. This is because  $T$  and  $N$  affect the number of values that are unnecessarily re-deserialized. Larger  $P$  leads to fewer unchanged values, and DDS becomes less effective.

Progressive matching (in particular, the steps that take place after the matching stacks are determined to match) has insignificant effect on DDS performance. This is because as the number of checkpoints increases (due to the decrease in the message portion size), the number of attempts to switch to fast mode increases as well. However, when  $P = 100$  (in which case, none of the attempts to switch to fast mode proved successful), the performance is comparable for all values of  $N$  other than 32. When  $N$  is 32, the difference in time is mostly due to the overhead of creating checkpoints, and not to progressive matching (otherwise we would see a significant difference for larger values of  $N$ ).

Comparing the portion of Figure 2 that plots hard doubles with the values in Table 3 demonstrates the cases in which DDS improves performance. In particular, for 100K element arrays, regular deserialization time is 288ms. Values less than 288 ms in the 100K array size table rows represent improvements, whereas values above 288 ms represent cases where the DDS overhead is not recovered. In general, when the message portion size is larger 32 bytes, and when only

25% or 50% (and even 75% in many cases) of array values change, DDS reduces deserialization time.

These tests demonstrate that one DDS performance bottleneck is checkpointing overhead. We are currently addressing this issue in subsequent optimizations, which show considerable promise. We plan to further improve DDS performance as we describe below.

## 5. SUMMARY AND FUTURE WORK

Differential deserialization (DDS) is a server-side optimization technique that takes advantage of similarities between messages in an incoming message stream to a Web service. The bSOAP parser and deserializer checkpoints its state and calculates checksums of portions of incoming messages. When the corresponding portions of the next message match, bSOAP can avoid the expensive step of completely parsing and deserializing the incoming message; instead, it can use the results of the previously parsed message. We describe the mechanism for checkpointing, switching back and forth between fast and regular deserialization modes, and memory management and error handling issues in our bSOAP implementation of DDS. A performance study demonstrates that the optimization technique shows potential for Web services that receive sequences of similar invocation request messages. DDS is the server-side analogue to differential serialization (DS) [4], but DDS is a completely different technique that does not depend in any way on DS. In fact, DDS does not require any changes to the SOAP client that invokes the service.

We plan to improve and enhance our DDS implementation in several ways. First, we will take advantage of the fact that a major portion of each checkpoint stack remains unchanged between two checkpoints. Currently, each checkpoint completely defines the parser state, leading to a considerable amount of redundant information stored in consecutive checkpoints. We could instead store only the changes in the contents of the stacks, from one checkpoint to the next. This improvement will decrease DDS memory requirements, checkpoint creation overhead, restoration time, and checking of namespace-alias mapping changes.

Another approach we plan to investigate to reduce redundant information in checkpoints and to improve overall performance of bSOAP, is to take advantage of the ability to predict state change. For example, the amount of state change just before an array element is about to be deserialized and just before the next array element is about to be deserialized is small, and can easily be predicted.

bSOAP could also detect message changes, and therefore the need to switch from fast to regular mode, more quickly by comparing a few bytes at the beginning, middle, and/or end of message portions. Only if these bytes match would the complete message portion checksum be calculated. This could help for the cases in which the checksumming algorithm itself is a significant portion of DDS overhead. Finally, using heuristics to dynamically tune message portion sizes might help in improving DDS performance, by reducing time unnecessarily spent in regular mode.

**Acknowledgement:** We would like to thank Madhu Govindaraju and our other co-authors of [11] for much of the material upon which Sections 2.1 and 2.3 are based.

## 6. REFERENCES

- [1] The Blocks Extensible Exchange Protocol Core (BEEP), March 2001. <http://www.ietf.org/rfc/rfc3080.txt>.
- [2] N. Abu-Ghazaleh, M. Govindaraju, and M. J. Lewis. Optimizing Performance of Web Services with Chunk-Overlaying and Pipelined-Send. *Proceedings of the International Conference on Internet Computing (ICIC)*, pages 482–485, June 2004.
- [3] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Performance of Dynamic Resizing of Message Fields for Differential Serialization of SOAP Messages. *Proceedings of the International Symposium on Web Services and Applications*, pages 783–789, June 2004.
- [4] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential Serialization for Optimized SOAP Performance. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, pages 55–64, June 2004, Honolulu, Hawaii.
- [5] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 246–254, Edinburgh, Scotland, July 23–26, 2002.
- [6] K. Chiu and W. Lu. A Compiler-Based Approach to Schema-Specific Parsing. In *First International Workshop on High Performance XML Processing*, 2004.
- [7] David Megginson et al. SAX 2.0.1: The Simple API for XML. <http://www.saxproject.org>.
- [8] K. Devaram and D. Andresen. SOAP Optimization via Parameterized Client-Side Caching. In *Proceedings of PDCS 2003*, pages 785–790, November 3–5, 2003.
- [9] E. Christensen et. al. Web Services Description Language (WSDL) 1.1, March 2001. <http://www.w3.org/TR/wsdl>.
- [10] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6)*, 2002.
- [11] M. R. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis. A Benchmark Suite for SOAP-based Communication in Grid Web Services. *SC—05: Supercomputing '05*, page to appear, Seattle WA, November 2005.
- [12] IBM and Microsoft Corporation. Direct Internet Message Encapsulation (DIME). <http://www-106.ibm.com/developerworks/library/ws-dime/>.
- [13] Indiana University, Extreme! Computing Lab. Grid Web Services. <http://www.extreme.indiana.edu/xgws/>.
- [14] T. Suzumura, T. Takase, and M. Tatsubori. Optimizing Web Services Performance by Differential Deserialization. *IEEE/ACM International Conference on Web Services*, pages 185–192, Orlando, FL, July 12–15, 2005.
- [15] T. S. T. Takase, H. Miyashita and M. Tatsubori. An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization.

- [16] R. A. van Engelen and K. Gallivan. The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In *The Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 128-135, May 21-24, 2002, Berlin, Germany.
- [17] World Wide Web consortium. Document object model. <http://www.w3c.org/DOM>.

## APPENDIX

### A. DUAL-MODE DDS PERFORMANCE

		Number of partitions ( $T$ )															
		1				100				200				500			
$P\%$	Arr. Size	Message portion size ( $N$ )				Message portion size ( $N$ )				Message portion size ( $N$ )				Message portion size ( $N$ )			
		32	512	1024	4096	32	512	1024	4096	32	512	1024	4096	32	512	1024	4096
25%	1000	2.39	1.04	1.01	1.24	2.97	<b>3.61</b>	3.35	<b>3.44</b>	3.16	<i>3.46</i>	3.34	<i>3.23</i>	3.91	3.46	3.34	3.26
	10000	23.1	10.2	9.98	9.72	23.9	<b>15.0</b>	18.7	<b>32.4</b>	24.4	<i>19.8</i>	27.6	<i>32.3</i>	26.3	33.4	33.8	32.3
	25000	57.5	26.0	24.4	23.7	58.8	<b>30.3</b>	33.1	<b>57.6</b>	59.2	<i>34.9</i>	42.3	<i>80.6</i>	61.0	49.5	69.0	80.6
	50000	116	51.0	48.6	47.4	118	<b>55.7</b>	57.5	<b>81.6</b>	119	<i>60.5</i>	66.7	<i>116</i>	120	75.1	94.6	162
	75000	173	77.0	72.8	70.4	176	<b>82.6</b>	81.8	<b>105</b>	177	<i>85.8</i>	91.1	<i>140</i>	178	100	119	242
	100000	238	111	97.7	94.3	235	<b>108</b>	106	<b>128</b>	236	<i>112</i>	115	<i>164</i>	237	126	142	269
50%	1000	4.67	1.86	1.88	1.91	4.68	3.48	3.38	3.26	5.20	3.47	3.34	3.23	5.70	3.43	3.32	3.24
	10000	41.1	18.5	17.7	17.7	41.9	23.3	26.4	32.7	42.1	27.7	33.5	32.2	44.0	35.1	33.8	32.3
	25000	103	46.4	44.1	42.6	103	55.8	52.9	77.7	104	55.5	61.8	80.5	105	70.1	84.1	80.5
	50000	205	93.0	88.5	85.2	207	97.4	97.4	120	207	102	106	155	209	116	134	161
	75000	308	139	132	128	311	144	143	163	312	149	150	196	313	163	178	242
	100000	413	186	177	170	414	192	186	205	415	195	194	237	417	209	223	322
75%	1000	5.92	2.71	2.61	2.62	6.42	<b>3.46</b>	3.33	<b>3.26</b>	6.66	<i>3.46</i>	3.32	<i>3.23</i>	7.21	3.46	3.35	3.25
	10000	59.0	26.9	26.0	25.0	59.3	<b>31.4</b>	33.6	<b>32.2</b>	60.0	<i>35.8</i>	33.5	<i>32.1</i>	61.8	35.5	33.5	32.3
	25000	147	67.4	63.9	61.5	148	<b>72.4</b>	73.2	<b>80.5</b>	148	<i>76.4</i>	81.2	<i>80.7</i>	150	87.7	83.6	80.6
	50000	295	135	128	123	326	<b>139</b>	137	<b>158</b>	296	<i>144</i>	146	<i>161</i>	299	158	168	161
	75000	445	202	192	185	446	<b>206</b>	200	<b>219</b>	447	<i>211</i>	209	<i>241</i>	448	224	237	242
	100000	594	270	256	246	598	<b>274</b>	267	<b>280</b>	598	<i>280</i>	274	<i>316</i>	597	292	301	322
100%	1000	7.72	3.46	3.32	3.22	7.65	3.46	3.33	3.22	7.70	3.45	3.31	3.23	7.66	3.46	3.32	3.22
	10000	77.2	35.1	33.5	32.2	76.4	35.1	33.4	32.2	76.6	35.8	33.5	32.5	76.9	35.4	33.5	32.2
	25000	191	87.8	85.2	80.3	191	87.8	83.6	80.5	192	87.8	84.0	80.3	191	87.8	84.2	80.4
	50000	391	177	168	161	387	176	168	161	386	176	168	161	385	176	167	161
	75000	581	265	251	242	590	265	252	242	582	264	251	241	583	264	252	242
	100000	780	354	336	322	779	353	335	322	779	353	335	322	779	353	335	322

**Table 3: Dual-mode deserialization times for hard doubles arrays of various sizes. Different tests contain results for all combinations of messages divided into 1, 100, 200, and 500 partitions ( $T$ ); DDS message portion sizes ( $N$ ) of 32, 512, 1024, and 4096 bytes; arrays of size 1K, 10K, 25K, 50K, 75K, and 100K; and the percentage of values that differ from one message to the next ( $P$ ) set at 25%, 50%, 75%, and 100%.**