

Investigating the Limits of SOAP for Scientific Computing

**Kenneth Chiu
Madhusudhan Govindaraju
Randall Bramley**

Indiana University

SOAP 1.2 Specification

- SOAP is a XML protocol for message-based distributed communication. Offers interoperability, human-readability, loose-coupling.
- SOAP is a layered specification:
 - Messaging framework (Part 1).
Only required part.
 - Data model/encoding (Part 2).
Data structures/values modelled as a DAG. DAG must then be encoded.
 - Type system.
Can be ad hoc. XML Schema most widely accepted.

Example of Array

```
<array enc:arraySize="2">  
  <item>1.23456234</item>  
  <item>3.14159267</item>  
</array>
```

Motivation

SOAP is commonly perceived to be slow.

- Is this due to immature implementations, or more fundamental limits?
- Where should performance efforts focus?
- Any good predictors of maximum SOAP performance?

We sought to answer these questions for SOAP with the XML Schema datatypes. Other type systems valid, but wanted to investigate this first.

We mainly investigated arrays of doubles, since we believe this are the predominate data type for scientific computing.

Related Work

- Shirasuna et al. (2002).
Showed benefits of chunking and base64 encoding.
- Davis and Parashar (2002).
Tested latency of strings and arrays of integers of several existing implementations. Showed disabling Nagle's algorithm important.
- Govindaraju et al.(2000).
Examined the size of SOAP messages. Suggested a multiprotocol approach.

Optimizations

Profiling and examining existing implementations suggested some optimizations which we investigated with an experimental SOAP implementation:

- Single-pass XML parsing.
 - Tries.
 - Streamed content model.
- Schema-specific parsing.
 - Array parser.

Single-Pass Parsing

- Common parsing interfaces (DOM, SAX) require two passes over the XML data.
 - First pass by parser to discern XML structures such as tags and content.
 - Structures presented to application which then makes a second pass.
- Single-pass parsing accomplished with:
 - Tag-specific callbacks with tries.
 - Streamed content interface between the parser and application.

Trie

- For given tag, applications must determine the right piece of code to execute.
- Straightforward tag-matching techniques require multiple passes over data.
 - Balanced, binary trees.
 - Hashing. Two strings, same hash.
 - Perfect hashing. Not in table, valid hash.
- Tries are table-driven representations of restricted DFAs.
 - PATRICIA tries. Not in trie, valid hash.

Trie Results

- Deserialization tests conducted with an array of mesh interface objects:

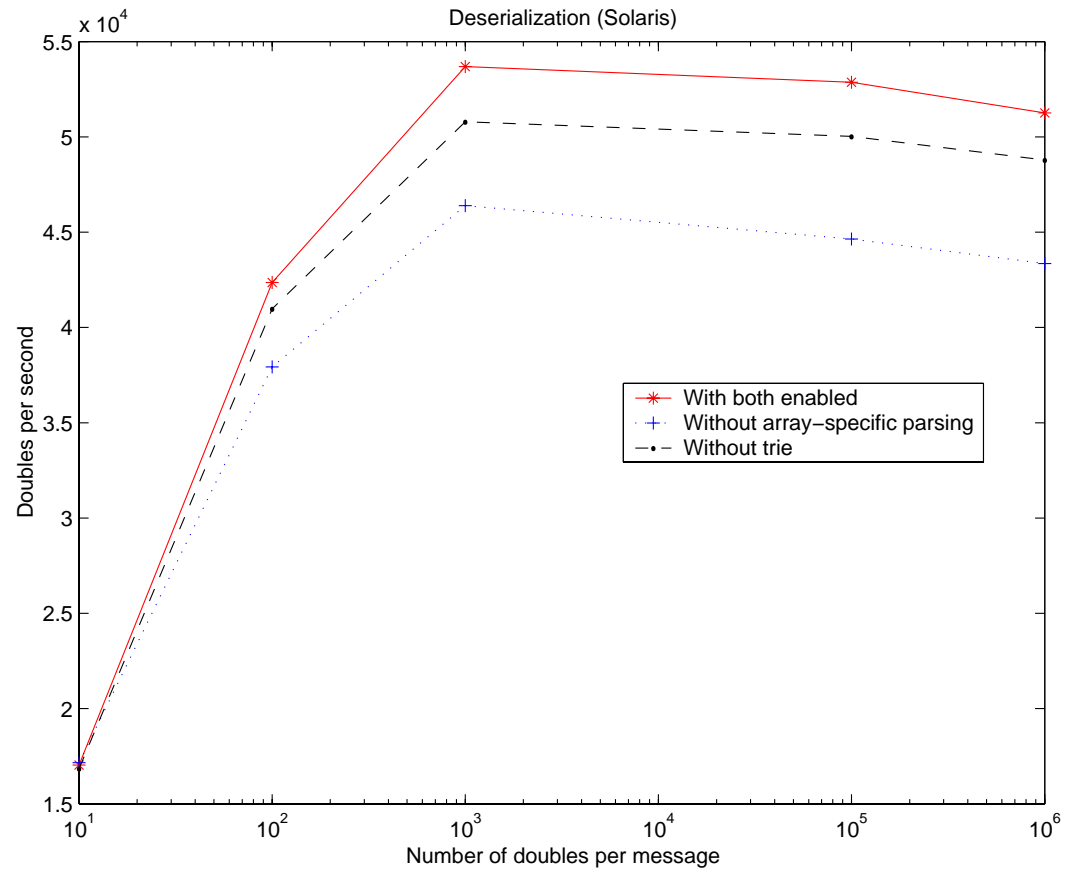
```
<mesh>  
  <x_coord>123</x_coord>  
  <y_coord>456</y_coord>  
  <field>1.234567901234567</field>  
</mesh>
```

Platform	STL Map	Trie	Improvement
Linux	73,000	96,000	32%
Solaris	22,000	27,000	23%

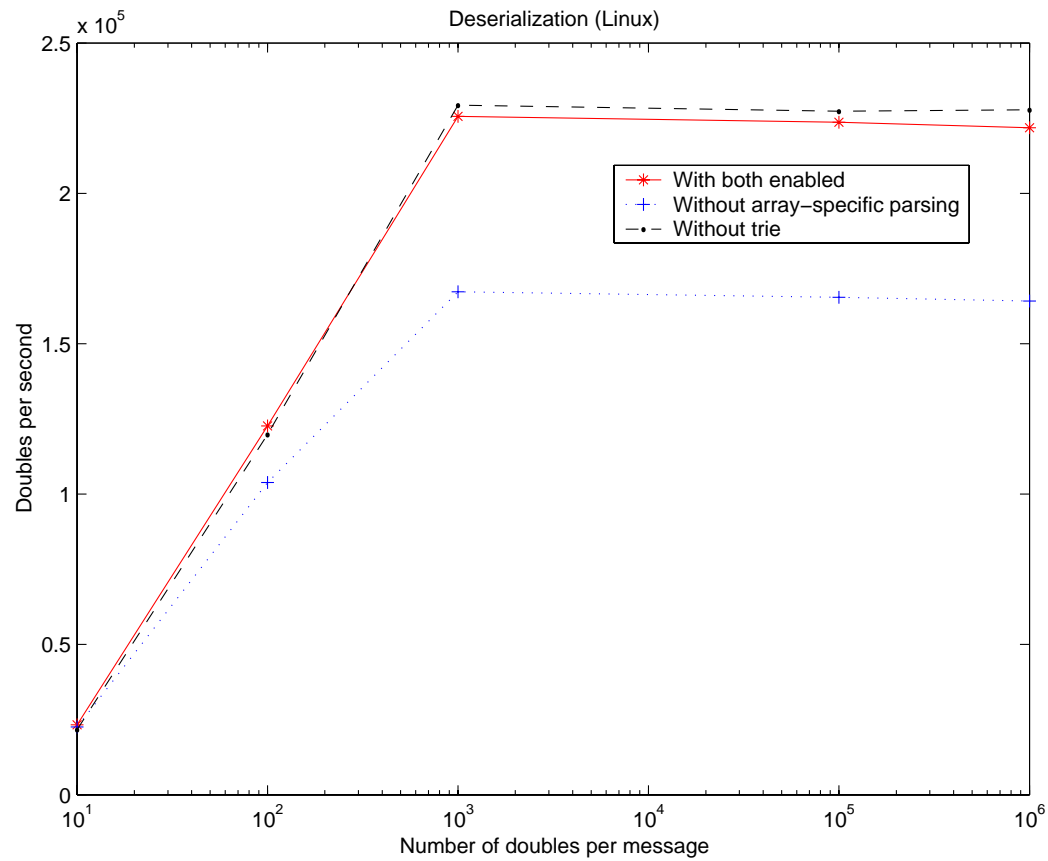
Schema-Specific Parsing

- Conventional wisdom expects validating parsers to be slower than non-validating parsers.
 - General XML parsing stage.
 - Validation stage.
- But the schema *restricts* the language. Generally, smaller languages are faster to parse.
- Exploit this by using parser coded to the schema.
- Tested this idea with an array parser. When, from the schema, we know we are in an array, we switch to a simpler parser.

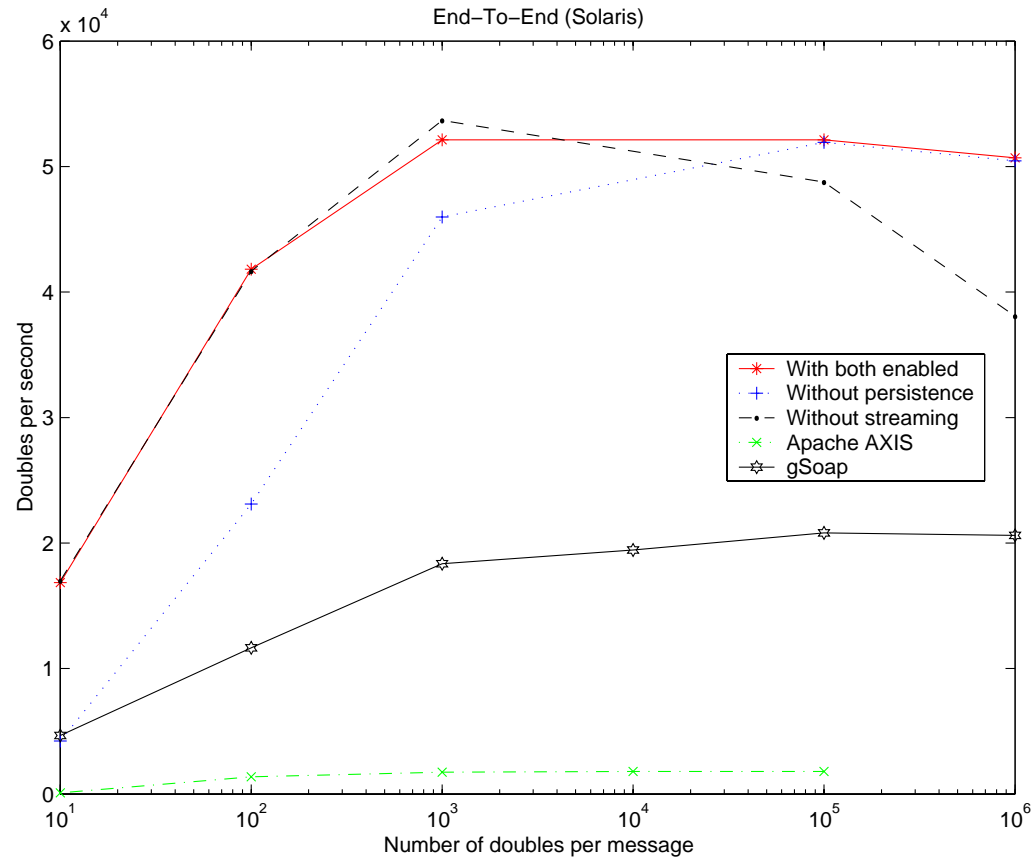
Array Parser (Solaris)



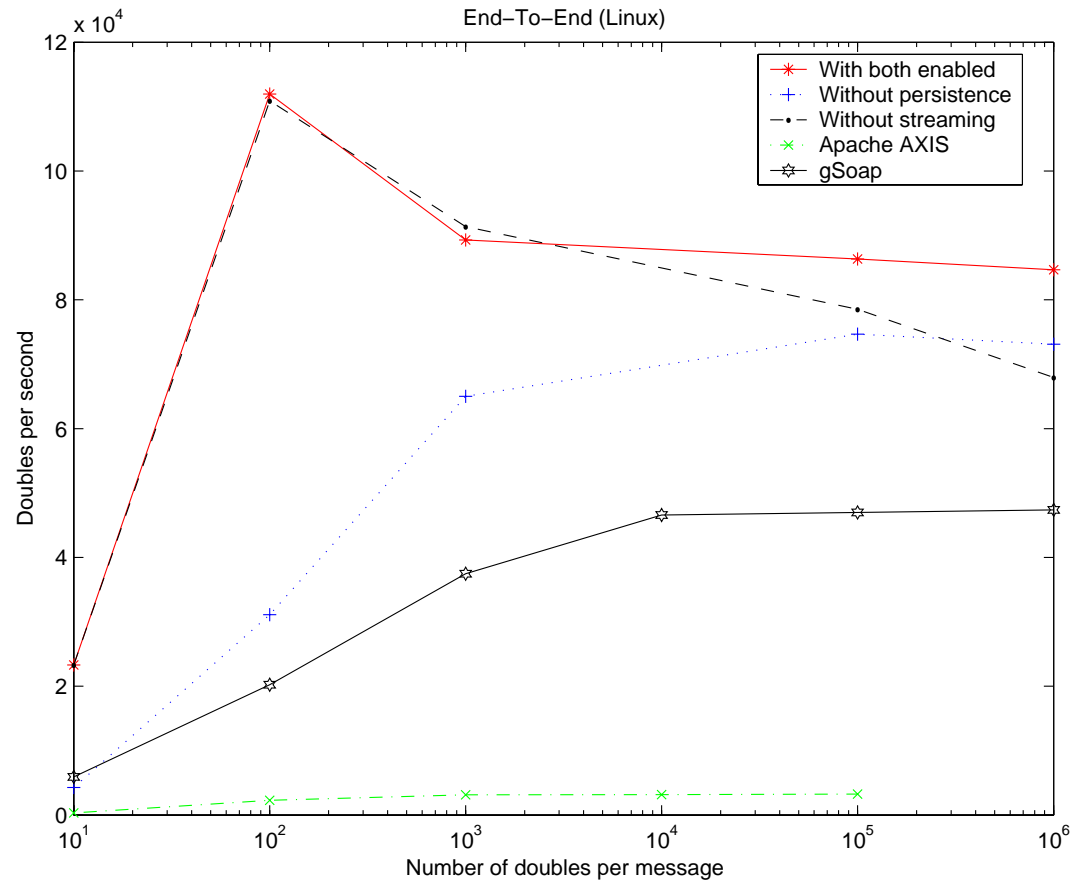
Array Parser (Linux)



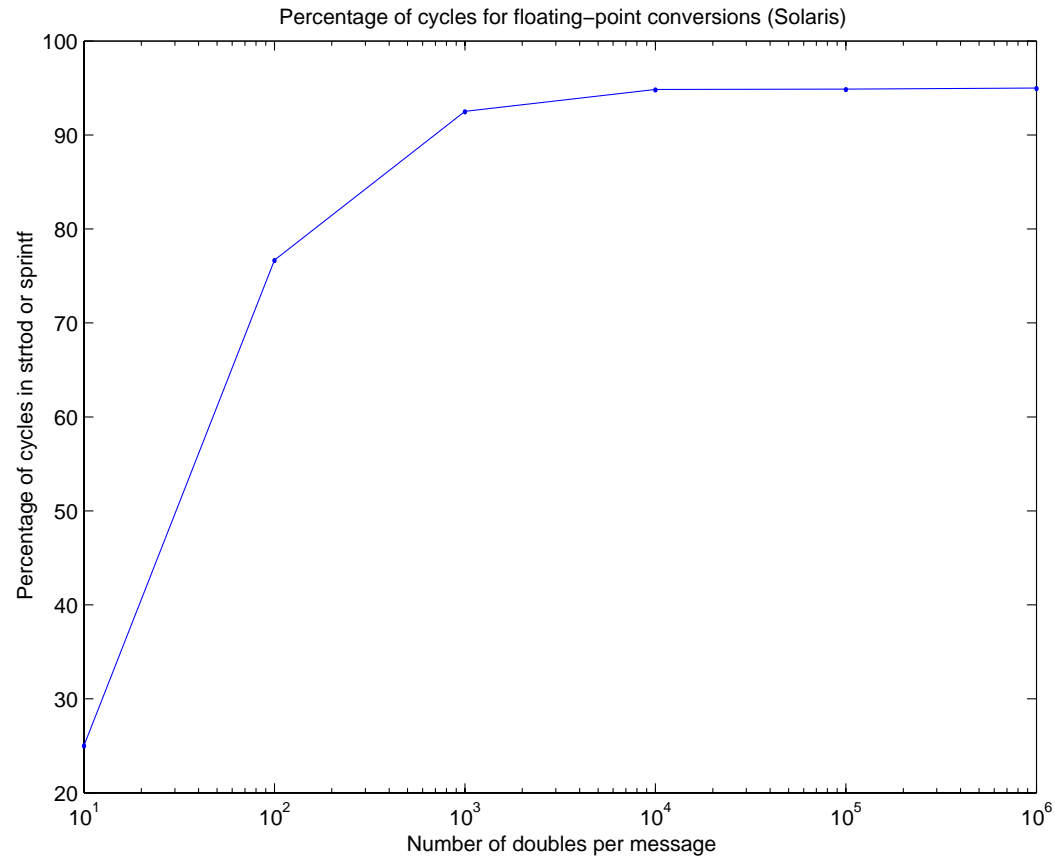
End-To-End (Solaris)



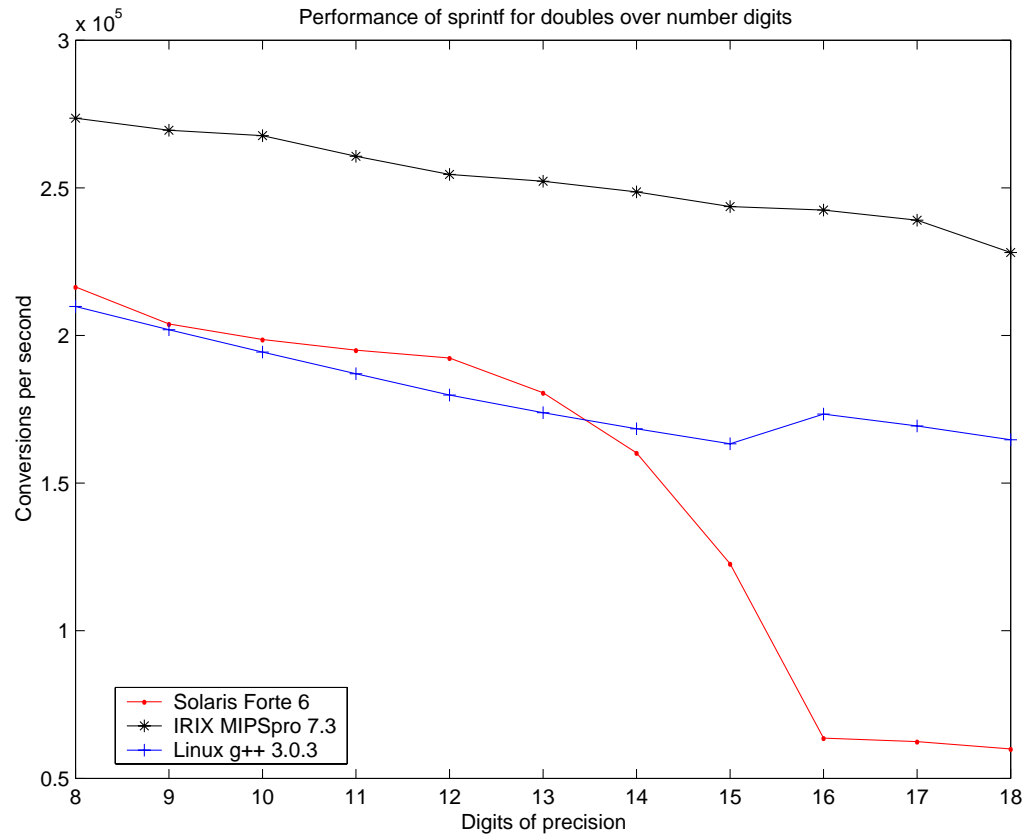
End-To-End (Linux)



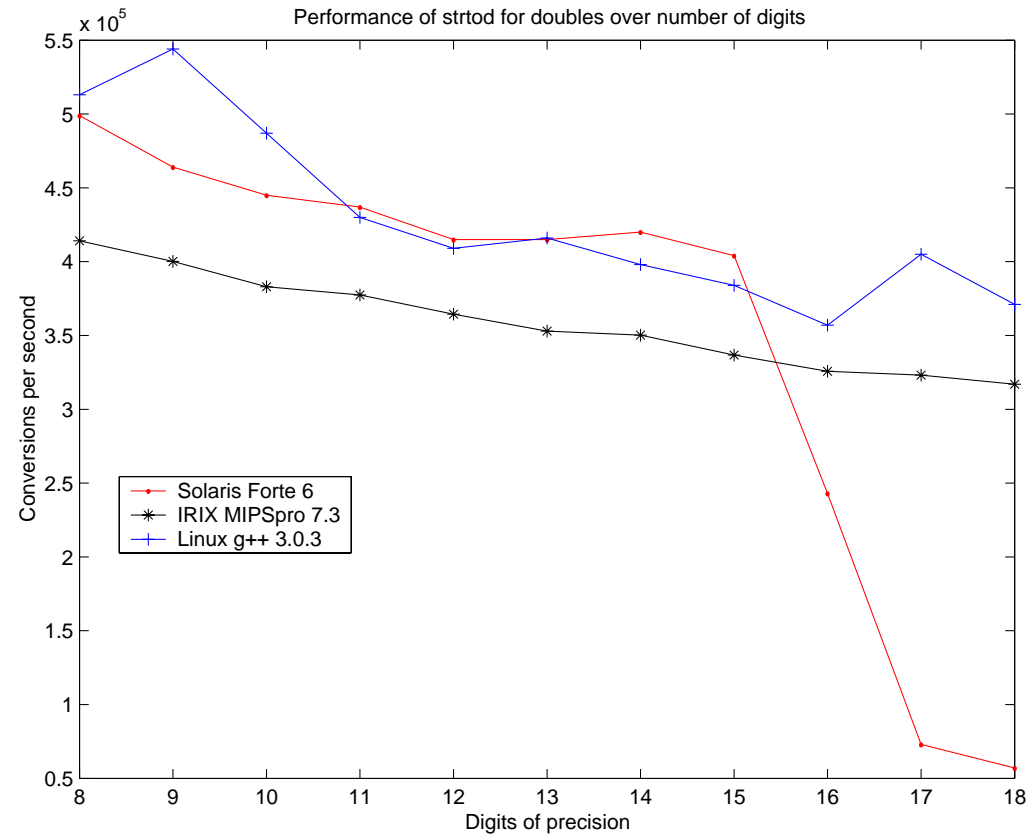
Relative Cost of ASCII Conversion



Sprintf Timings



Strtod Timings



Conclusions

- With optimizations such as schema-specific parsing, the bottleneck becomes ASCII-double conversion.
- Performance of ASCII-double conversion can be used as rough guide for protocol decisions.
- Two-pass method for HTTP 1.0 Content-Length compliance (like gSOAP) is likely to be slower. But can use a fast, upper-bound calculation and pad.

Future Work

- Investigate faster ASCII-double conversion algorithms.
Slow because they use arbitrary-precision integer arithmetic. IRIX is tantalizingly fast.
- Schema-specific parsing.
Compile schema directly to C, Java bytecode, or machine code. Trie open-coded (embedded in instruction stream) rather than table-driven.
- Hybrid schemes, both ASCII and binary.
- Binary XML.
Preferably based on XML Infoset (PSVI). Can be made compatible with Schema type system, which would leverage existing specs and code.

End

Serialization Phases

1. Traverse data structures representing message.
2. Convert machine representation of data to ASCII.
3. Write ASCII to buffer.
4. Initiate network transmission.

Analogous for deserialization. Just a framework, not necessarily correspond to functions.

Deserialization Phases

5. Read from network into memory buffer.
6. Parse XML.
7. Handle elements.
8. Convert ASCII to machine representation.

Analogous for serialization. Just a framework, not necessarily correspond to functions.