Benchmark Suite for Web Services

Madhu Govindaraju
Grid Computing Research Laboratory
Department of Computer Science
Binghamton University
State University of New York

Web Service Performance

- Performance is governed by the design and implementation choices of
  - **SOAP toolkit**
  - **XML parser**

Motivation for a Benchmark

- SOAP implementations are interesting and important to compare and contrast for three different reasons:
- 1. Web services based grid applications place disparate requirements on their communication substrate
  - **Diverse application requirements lead to a wide range of different implementation choices.**

Motivation (contd)

- 2. Various individual features of SOAP require clever implementation techniques to achieve improved performance.
  - **Often, the naïve implementation leads to considerable processing time.**
- 3. The number of SOAP implementations and toolkits is both large and growing.
  - **SOAP toolkits exist in languages such as C, C++, Java, C#, Perl and Python.**

Requirements for Web Services Based Applications

- High end-to-end performance
- Serialization and deserialization efficiency
- Small memory footprint
- Specific security requirements
- Chunking and streaming capability
- Minimal toolkit overhead
- Scalability
- Support for optimized handling of scientific data structures

Designing a SOAP toolkit

- Role of HTTP
  - Content Length of HTTP header
  - Chunking and Streaming (HTTP 1.0 and 1.1)
- Handling Namespaces
  - Requires efficient use of namespace-stack
- Multi-Ref
  - Needed to efficiently represent data structures
  - Naïve implementation can hurt scalability

Designing a SOAP toolkit (contd)

- Handling XML
  - SAX, DOM and XPP
- Dynamic Invocations
  - Flexibility vs Performance
- Compression
  - SOAP is usually CPU bound, not network bound
- Support for Scientific Data
  - Use *Differential Serialization* for optimization
  - Use *Trie* data structures for efficient parsing

Toolkits Compared

- gSOAP 2.4
- XSOAP/XSUL 1.2.23
- AxisJava 1.2
- AxisC++ 1.1
- .NET 1.1.4322

Performance Study

- End-to-End performance
  - **array of doubles**
  - **array of integers**
- Deserialization
  - **array of doubles**
  - **array of integers**
  - **array of strings**
- Serialization
  - **base64 data sendBase64Imp**
  - **array of doubles**

Differential Serialization for Optimized SOAP Performance

Michael J. Lewis

Grid Computing Research Laboratory
Department of Computer Science
Binghamton University
State University of New York

Motivation

- SOAP is an XML-based protocol for Web Services that (usually) runs over HTTP
- Advantages
  - **extensible, language and platform independent, simple, robust, expressive, and interoperable**
- *The adoption of Web Services standards for Grid computing requires high performance*

The SOAP Bottleneck

- *Serialization and deserialization*
  - **The in memory representation for data must be converted to ASCII and embedded within XML**
  - **Serialization and deserialization conversion routines can account for *90% of end-to-end time* for a SOAP RPC call [HPDC 2002, Chiu et. al.]**
- Our approach
  - ***Avoid* serialization altogether, whenever possible**

Differential Serialization (in *bSOAP*)

- *Save a copy* of the last outgoing message
- If the next call's message would be similar, then
  - **use the previous message as a template**
  - ***only serialize the differences*** **from the last message**
- Outline
  - **assumptions and requirements**
    - applications that repeatedly resend similar messages
    - data update tracking
  - **strategies and implementations**
    - decrease the cost of partial reserialization
    - shifting, chunking, stuffing, stealing
  - **performance**

Update Tracking

- How do we know if the data in the next message will be the same as in the previous one?
- If it is different, how do we know which parts must be reserialized?
- How can we ensure that reserialization of message parts does not corrupt other portions of the message?

Data Update Tracking (DUT) Table

```
struct MIO { int a; int b; double val;};
int mioArray(MIO[] mios)
```

Field	TPointer	SLength	FWidth	Dirty?
X		5	5	YES
Y		3	7	YES
Z		5	10	NO

```
POST /mioExample HTTP/1.1
.
<?xml version='1.0'?><SOAP-ENV:Envelope ...>
.
<x xsi:type='xsd:int'>12345</x>
<y xsi:type='xsd:int'>678</y>
<z xsi:type='xsd:double'>1.166</val>
.
</SOAP-ENV:Envelope>
```

Problems and Approaches

- Problems
  - **Some fields require reserialization**
  - **The current field width may be too small for the next value**
  - **The current message (or chunk) size may be too small**

Solving these problems enables DS, but incurs overhead

- Approaches
  - **shifting**
  - **chunking**
  - **stuffing**
  - **stealing**
  - **chunk overlaying**

Performance

- Performance depends on
  - **which techniques are invoked**
  - **"how different" the next message is (application specific)**
    - Message Content Matches
      - **identical messages, no dirty bits**
    - Perfect Structural Matches
      - **data elements and their sizes persist**
    - Partial Structural Matches
      - **some data elements change size**
      - **requires shifting, stealing, stuffing, etc.**
- We study the performance of all our techniques on synthetic workloads of scientific data
  - **(our other work models application traffic)**

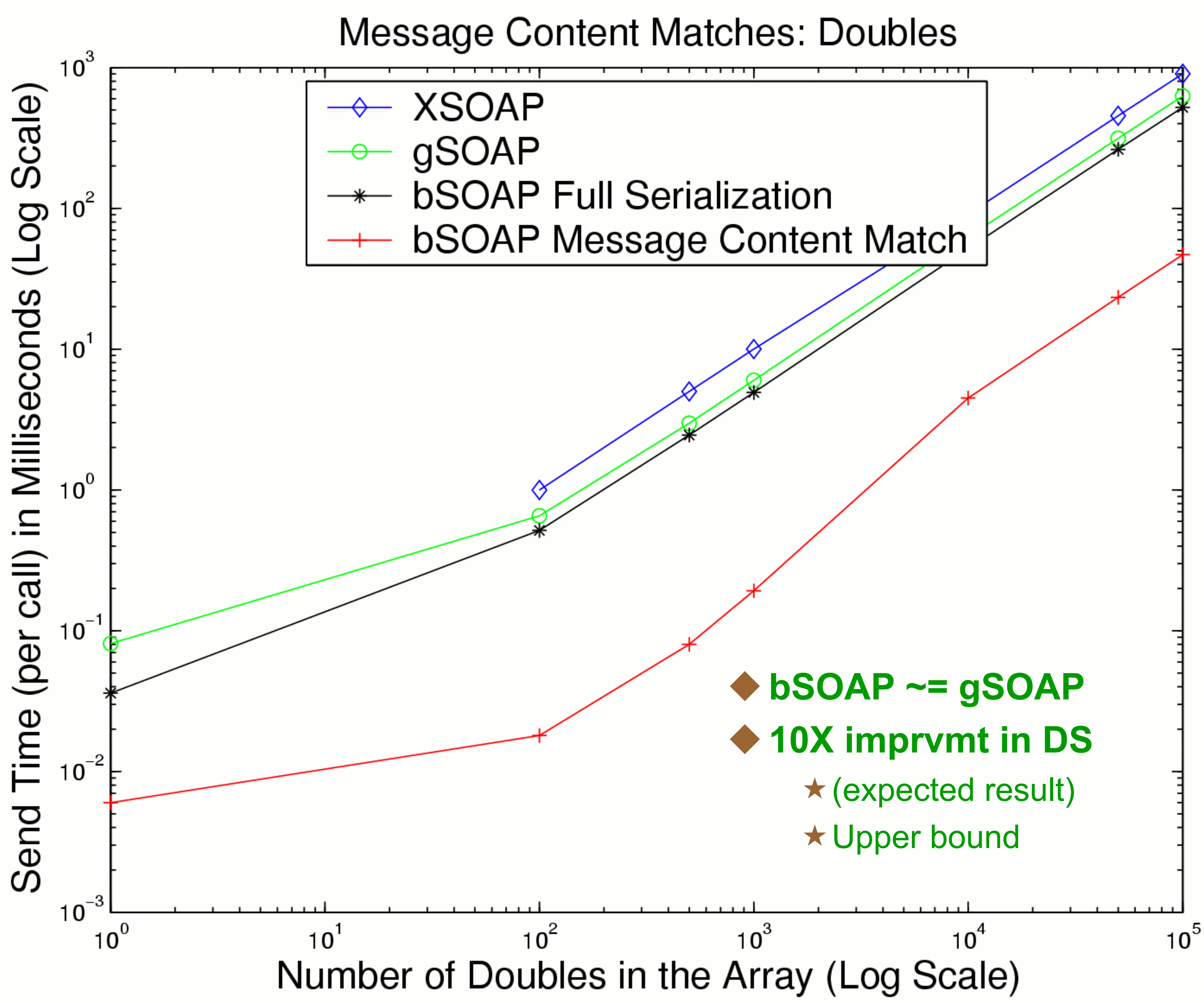
Experimental Setup

- Machines
  - **Dual Pentium 4 Xeon 2.0 GHz, 1 GB DDR RAM, 15K RPM 18 GB Ultra-160 SCSI drive.**
- Network
  - **Gigabit Ethernet.**
- OS
  - **Debian Linux. Kernel version 2.4.24.**
- SOAP implementations
  - ***bSOAP* and *gSOAP v2.4* compiled with gcc version 2.95.4, flags: -O2**
  - ***XSOAP 1.2.28-RC1* compiled with JDK 1.4.2**
  - **bSOAP/gSOAP socket options: SO_KEEPALIVE, TCP_NODELAY,SO_SNDBUF = SO_RCVBUF = 32768**
  - **Dummy SOAP Server (no deserialization).**

Message Content Matches

- *Message Content Match*:
  - **The entire stored message template can be reused without change**
  - **No dirty bits in the DUT table**
  - **Best case performance improvement**

- Performance Study
  - **compare gSOAP, XSOAP, and bSOAP, with differential serialization on and off**
  - **vary the message size**
  - **vary the data type: doubles and MIO's (not shown)**

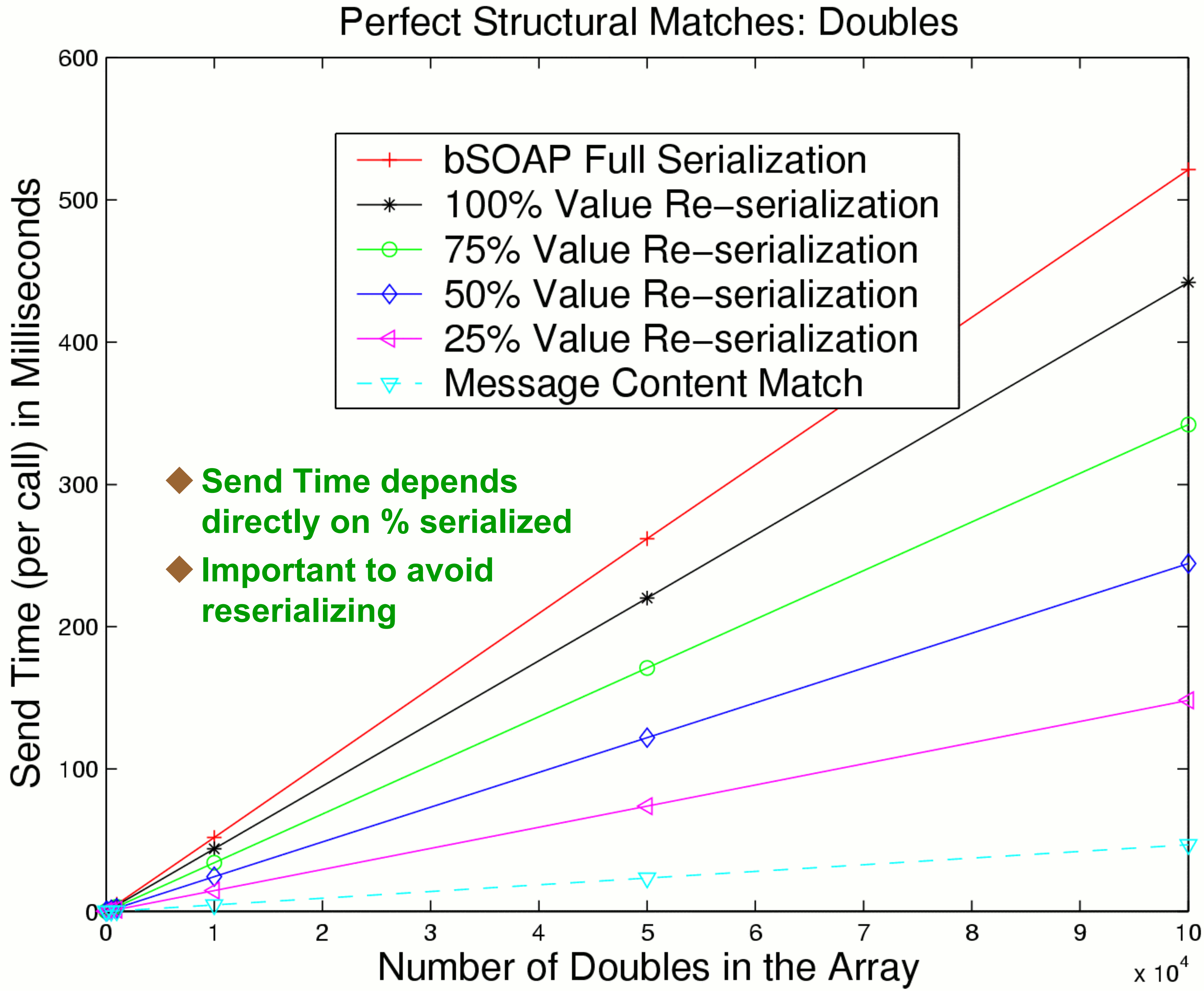


Message Content Matches: Doubles
XSOAP
gSOAP
bSOAP Full Serialization
bSOAP Message Content Match
Send Time (per call) in Milliseconds (Log Scale)
Number of Doubles in the Array (Log Scale)
10^3
10^2
10^1
10^0
10^-1
10^-2
10^-3
10^0
10^1
10^2
10^3
10^4
10^5
bSOAP ~= gSOAP
10X imprvmt in DS
(expected result)
Upper bound

Perfect Structural Matches

- *Perfect Structural Matches*:
  - **Some data items must be overwritten (DUT table dirty bits)**
  - **No shifting required**

- Performance study:
  - **vary the message size**
  - **vary the reserialization percentage**
  - **vary the data type**
    - *Doubles* and
    - *Message Interface Objects* (MIO's, <int, int, double>) (not shown)

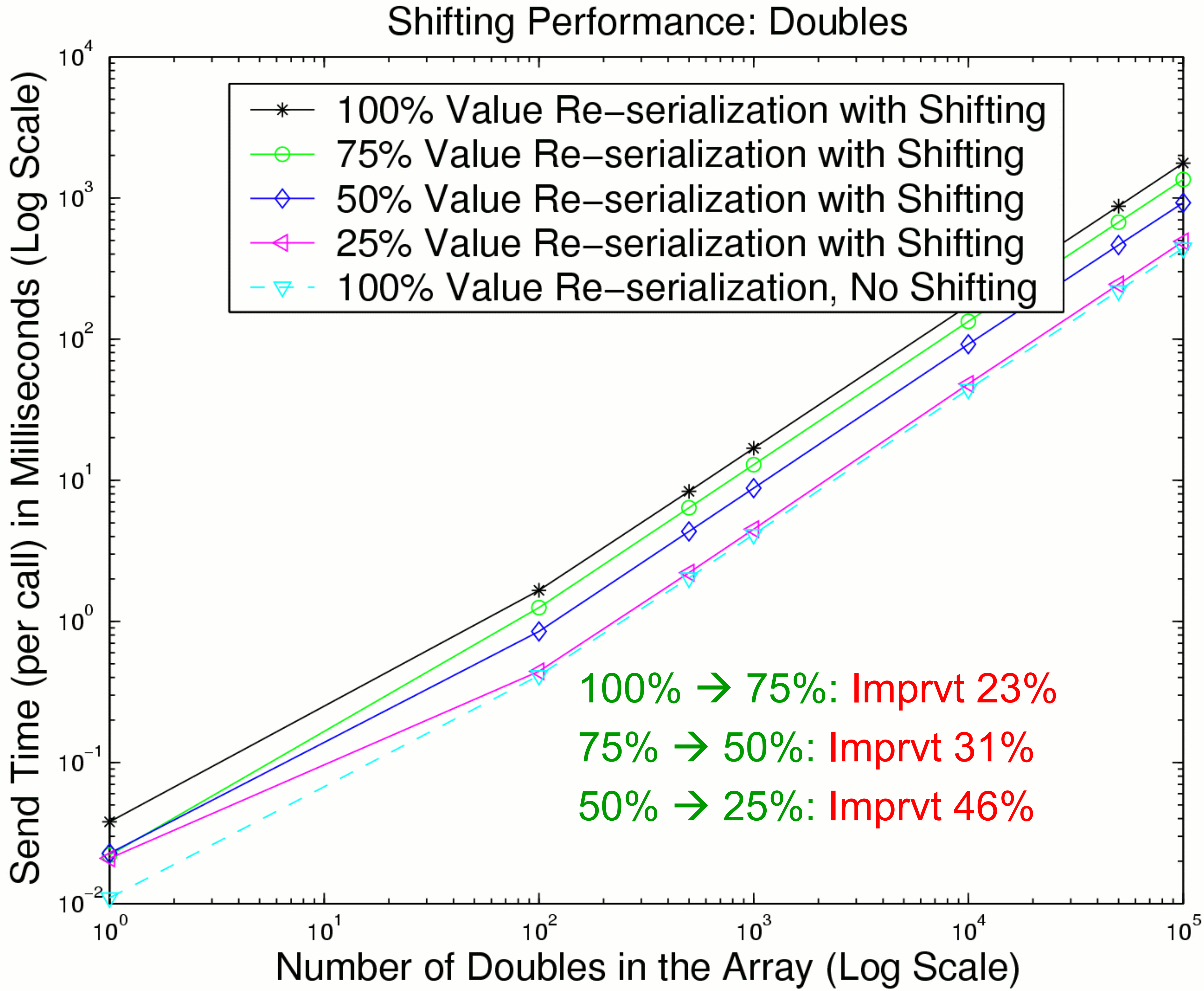


Perfect Structural Matches: Doubles
bSOAP Full Serialization
100% Value Re–serialization
75% Value Re–serialization
50% Value Re–serialization
25% Value Re–serialization
Message Content Match
Send Time depends directly on % serialized
Important to avoid reserializing
Send Time (per call) in Milliseconds
Number of Doubles in the Array
x 10^4
0
100
200
300
400
500
600
1
2
3
4
5
6
7
8
9
10

Shifting

- *Partial Structural Match*:
  - **Not all of array elements are reserialized**

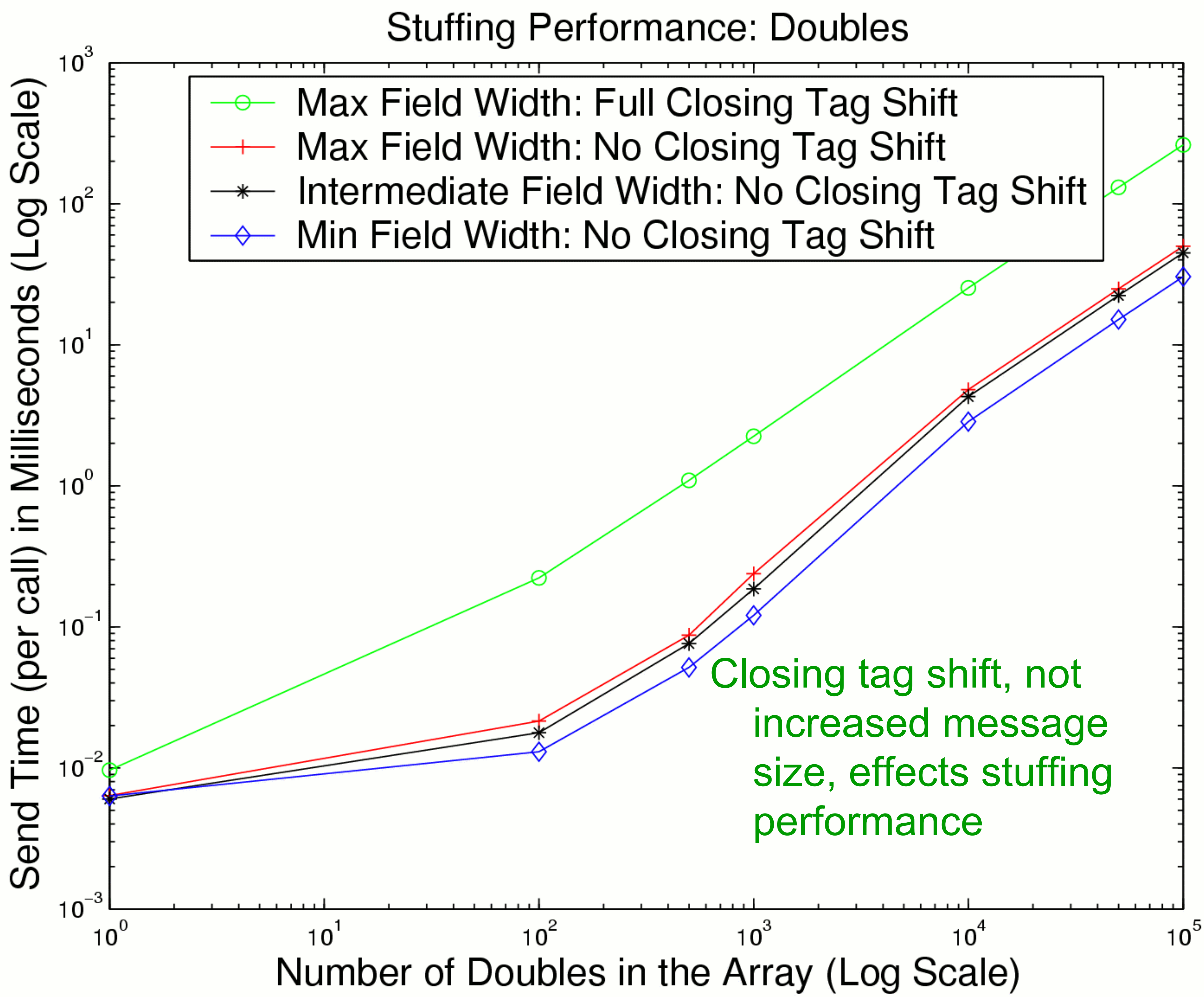
- Performance Study
  - **Intermediate size values to maximum size values.**
  - **Array of doubles (18 → 24)**
  - **Array of MIO's (36 →46)** ***(not shown)***


Shifting Performance: Doubles
100% Value Re–serialization with Shifting
75% Value Re–serialization with Shifting
50% Value Re–serialization with Shifting
25% Value Re–serialization with Shifting
100% Value Re–serialization, No Shifting
100% → 75%: Imprvt 23%
75% → 50%: Imprvt 31%
50% → 25%: Imprvt 46%
Send Time (per call) in Milliseconds (Log Scale)
Number of Doubles in the Array (Log Scale)

Stuffing

- *Closing Tag Shift*:
  - **Stuffed whitespace comes *after* the closing tag**
  - **Must move the tag to accommodate smaller values**

- Performance Study
  - **send smallest values (1 char)**
  - **vary field size: smallest, intermediate, maximum**
  - **Array of doubles (max = 24, intermediate = 18, min = 1)**
  - **Array of MIOs**
    - (max = 46, intermediate = 38, min = 3) *(not shown)*


Stuffing Performance: Doubles
Max Field Width: Full Closing Tag Shift
Max Field Width: No Closing Tag Shift
Intermediate Field Width: No Closing Tag Shift
Min Field Width: No Closing Tag Shift
Closing tag shift, not increased message size, effects stuffing performance
Send Time (per call) in Milliseconds (Log Scale)
Number of Doubles in the Array (Log Scale)
10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1} 10^{2} 10^{3}
10^{0} 10^{1} 10^{2} 10^{3} 10^{4} 10^{5}

Summary

- SOAP performance is poor, due to serialization and deserialization
- Differential serialization
  - **Save a copy of outgoing messages, and serialize changes only, to avoid the observed SOAP bottleneck**
- Techniques:
  - ***Shifting, chunking, chunk padding, stuffing, stealing, chunk overlaying***
- *Performance is promising (17% to 10X improvement), depends on similarity of messages*

Extra Slides

Other Approaches

- SOAP performance improvements
  - **Compression**
  - **Base-64 encoding**
  - **External encoding: Attachments (SwA), DIME**
- These approaches may be necessary and can be effective. However
  - **they undermine SOAP's beneficial characteristics**
  - **interoperability suffers**
- The *goal*
  - **improve performance, retain SOAP's benefits**

Applications that can Benefit

- *Differential Serialization is only beneficial for applications that repeatedly resend similar messages*

- Such applications do exist:
  - **Linear system analyzers**
  - **Resource information dissemination systems**
  - **Google & Amazon query responses**
  - **etc.**

Data Update Tracking (DUT) Table

- Each saved message has its own DUT table
- Each data element in the message has its own DUT table entry, which contains:
  - ***Location*****: A pointer to the data item's current *location* in the template message**
  - ***Type*****: A pointer to a data structure that contains information about the data item's type.**
  - ***Serialized Length:*** **The number of characters needed to store the last written value**
  - ***Field Width:*** **The number of allocated characters in the template**
  - **A *Dirty Bit* indicates whether the data item has been changed since the template value was written**

Updating the DUT Table

- DUT table dirty bits must be updated whenever in-memory data changes
  - **Current implementation**
    - explicit programmer calls whenever data changes
  - **Eventual intended implementation**
    - *more automatic*
    - variables are registered with our bSOAP library
    - data will have accessor functions through which changes must be made
    - when data is written, the DUT table dirty bits can be updated accordingly
      - **disallows “back door” pointer-based updates**
      - **requires calling the client stub with the same input param variables**

Shifting

- *Shifting*: Expand the message on-the-fly when the serialized form of a new value exceeds its field width
  - **Shift the bytes of the template message to make room**
  - **Update DUT table entries for all shifted data**

```
…</w><x xsi:type='xsd:int'>1.2</x><y xsi:type=….
```

becomes

```
…</w><x xsi:type='xsd:int'>1.23456</x><y xsi:type=….
```

- **Performance penalty**
  - DUT table updating, memory moves, possible memory reallocation

Stuffing

- *Stuffing*: Allocate more space than necessary for a data element
  - **explicitly when the template is first created, or after serializing a value that requires less space**
  - **Helps *avoid* shifting altogether**
  - **Doesn't work for strings, base64 encoding**

```
…<y xsi:type='xsd:int'>678</y><z xsi:type=…
```

can be represented as

```
…<y xsi:type='xsd:int'>678</y>□□□□<z xsi:type=…
```

Stealing

- *Stealing*: Take space from nearby stuffed fields
  - **Can be less costly than shifting [ISWS '04]**

```
…'>678</y><z xsi:type='xsd:double'>1.166</val>□□□□□
```

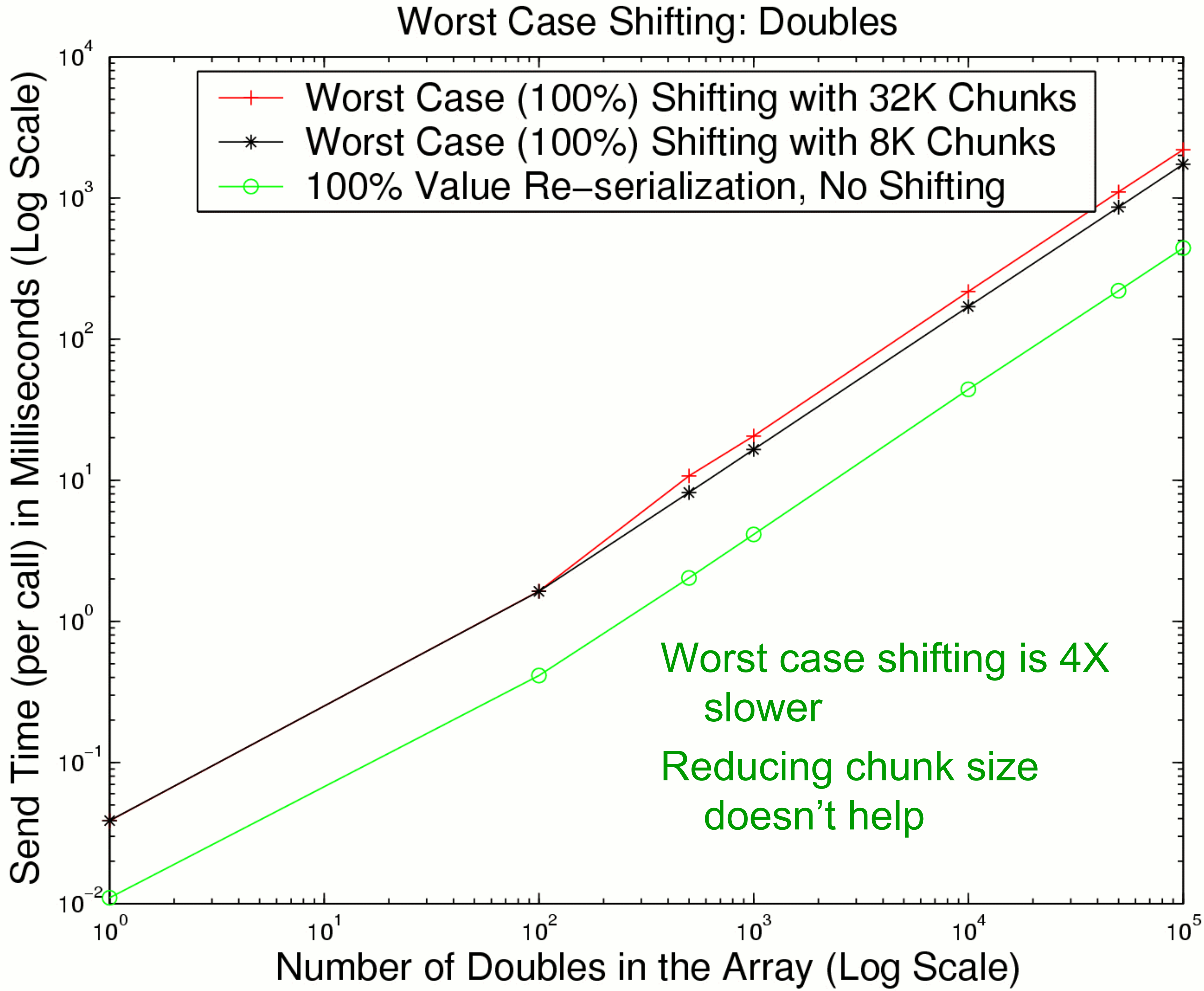
y can steal from z to yield…

```
…'>677.345</y><z xsi:type='xsd:double'>1.166</val>□
```

- Performance depends on several factors
  - ***Halting Criteria:* When to stop stealing?**
  - ***Direction:* Left, right, or back-and-forth?**

Worst Case Shifting

- *"Worst case shifting"*:
  - **All values are reserialized from smallest size values to largest size values.**
- Performance Study
  - **vary the chunk size (8K and 32K)**
  - **Array of doubles (1 → 24).**
  - **Array of MIOs (3 → 46)** ***(not shown)***


Worst Case Shifting: Doubles
Worst Case (100%) Shifting with 32K Chunks
Worst Case (100%) Shifting with 8K Chunks
100% Value Re–serialization, No Shifting
Send Time (per call) in Milliseconds (Log Scale)
Number of Doubles in the Array (Log Scale)
Worst case shifting is 4X slower
Reducing chunk size doesn't help

A Compiler-Based Approach to Schema-Specific Parsing

Kenneth Chiu
Grid Computing Research Laboratory
SUNY Binghamton

Sponsored by NSF ANI-0330568.

Motivation

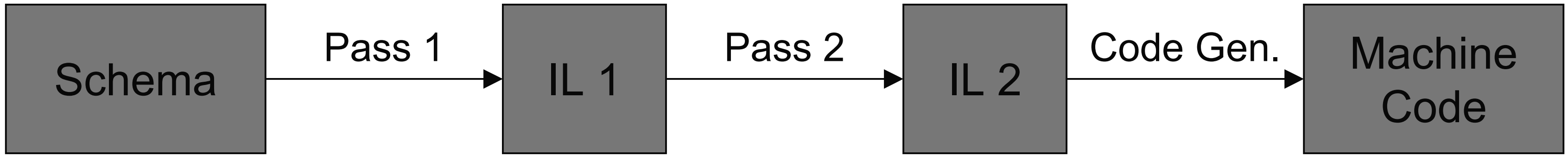
- Schema provides additional information.
- Use it to speed up parsing.
- Generate code as efficient as hand-written.
  - **From this:**

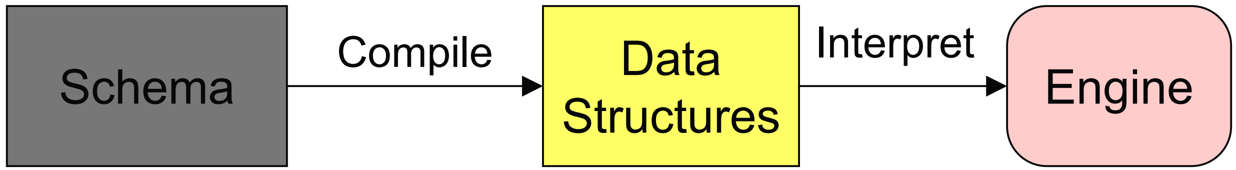
```
<element name="el3" maxOccurs="3" …>
```

  - **Generate this:**

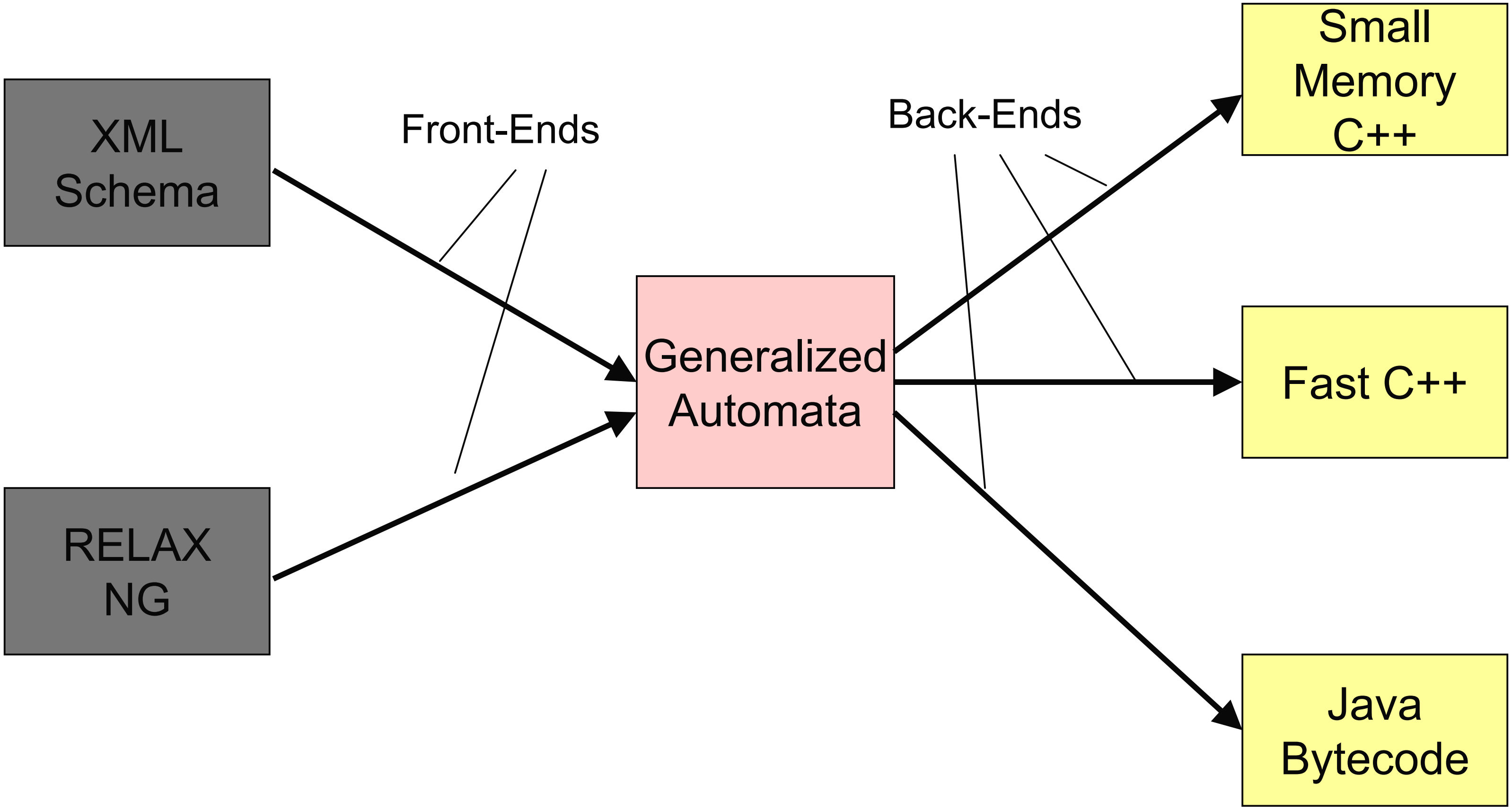
```
assure_3_chars_in_buf();
if (*c++ != 'e') goto error;
if (*c++ != 'l') goto error;
if (*c++ != '3') goto error;
if (++el3_count > 3) goto error2;
```

A Schema Compiler





Prototype Architecture



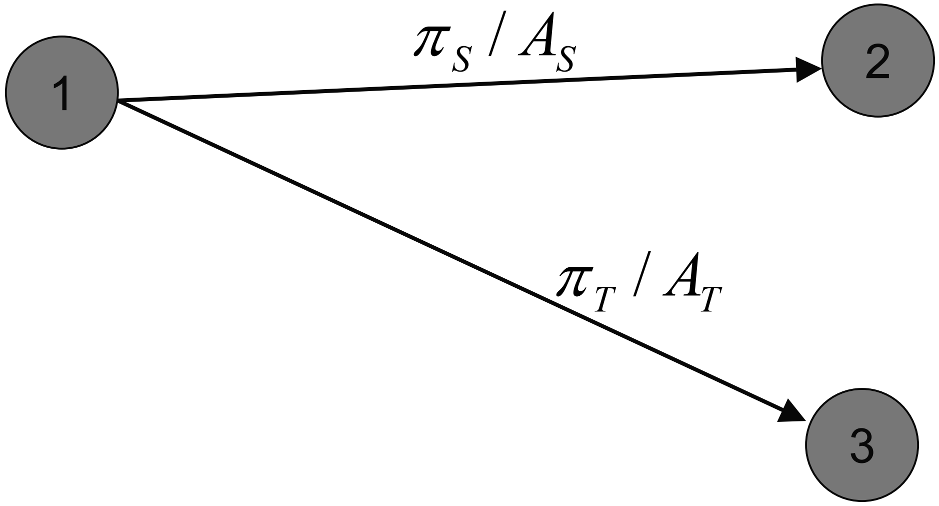
Generalized Automata

- A generalization of PDAs.
  - **Each GA has a set of variables.**
    - Possibly unbounded in value.
  - **Each transition is "guarded" by a predicate over the variables.**
  - **Each transition has a set of actions over the variables.**
    - Actions are executed when the transition is taken.
- Not a model for computation, since anything can happen in predicates and actions.
  - **In theory can handle any kind of schema construct. Real question is whether it enables generation of optimized code for that construct.**

Why Not CFGs?

- CFGs are very good for complex syntactic structures.
  - **Very good at things like recreating an AST for an expression from a sequence of chars.**
- XML structure is relatively simple.
  - **Easy to recreate the tree structure from a sequence of chars.**
- CFGs cannot model some things well, like occurrence constraints.
- Want something that permits a well-defined set of transforms, without being too restrictive.

Example



Predicates and Actions

- Predicates and actions are the instruction set of an abstract schema machine.
  - **Transformed into executable code.**
  - **Definition not part of GA model.**
- One set for all schema languages?
  - **Regular tree language**
  - **Efficiency**

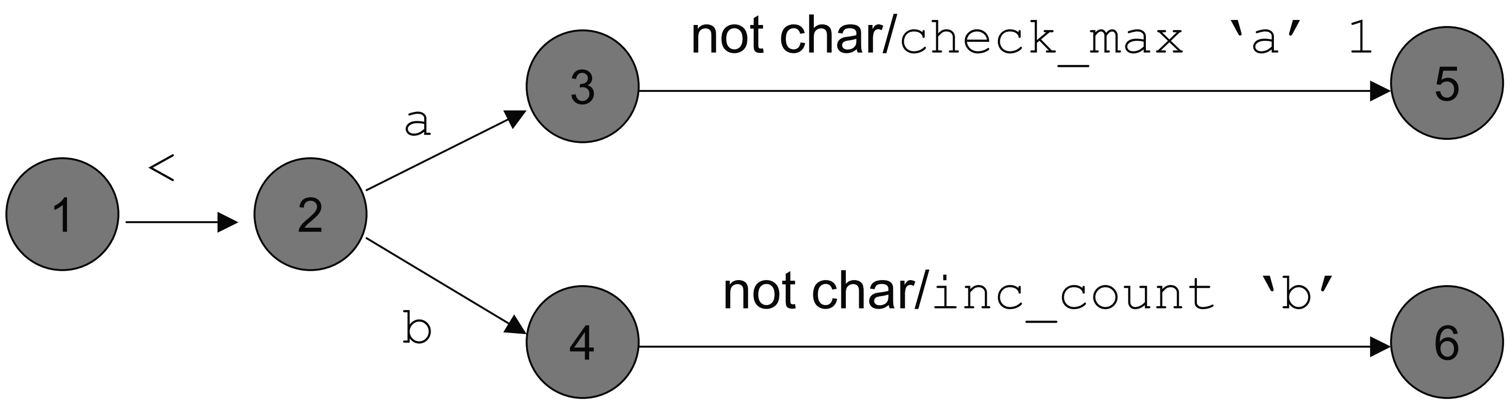
Examples

- `match 'a'`
  - **Current input character is 'a'.**
- `occurrence 'el3' '<= 5'`
  - **Element 'el3' has occurred no more 5 times.**
- `consume`
  - **Consume current input character.**
- `prefix_start`
  - **Beginning of namespace prefix.**
- `prefix_char 'a'`
  - **Encountered prefix character 'a'.**
- `prefix_end`
  - **End of prefix.**

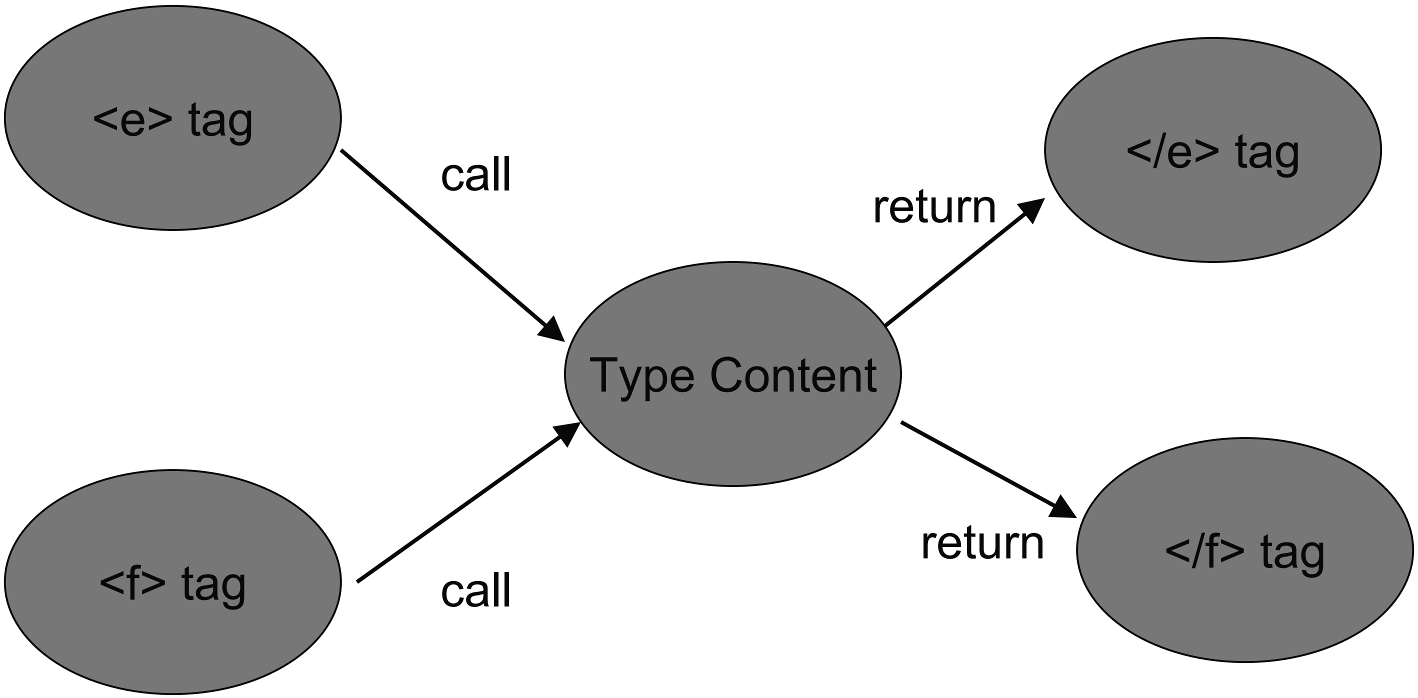
Examples

- RELAX NG `<interleave>`

```
<interleave>
  <ref name="a">
  <oneOrMore>
    <ref name="b">
   </oneOrMore>
</interleave>
```



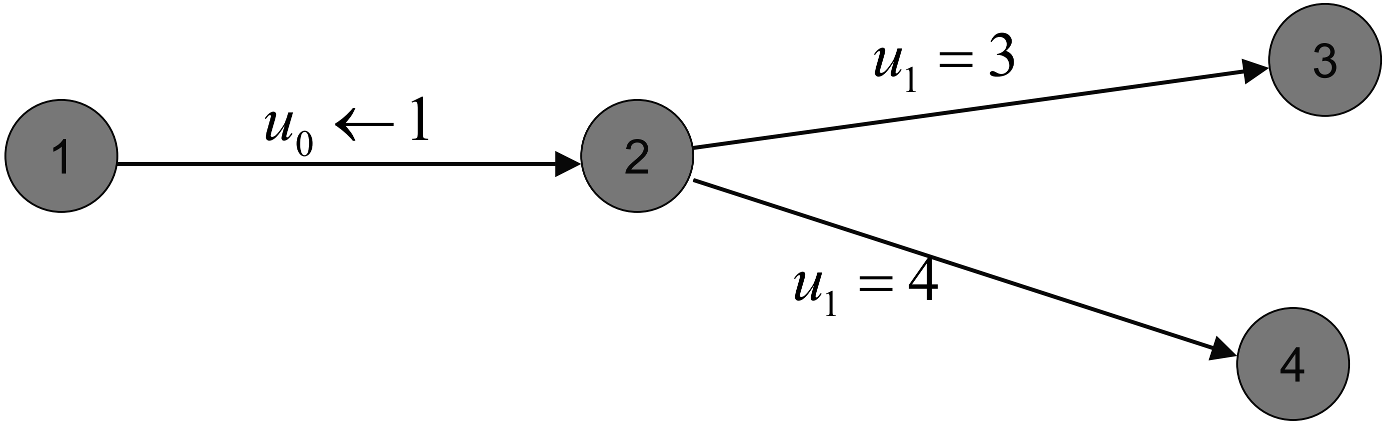
Content

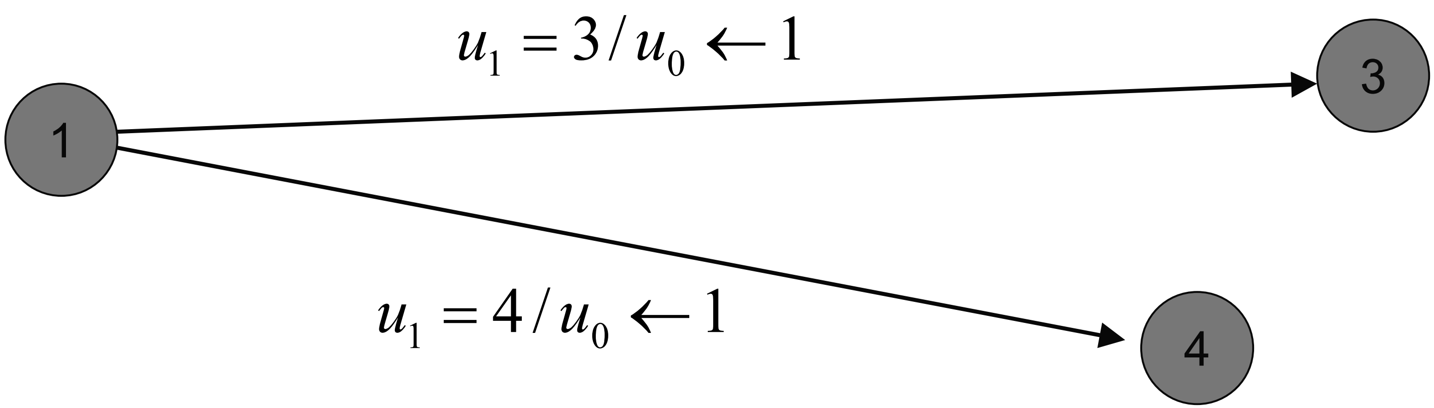


NGA to DGA

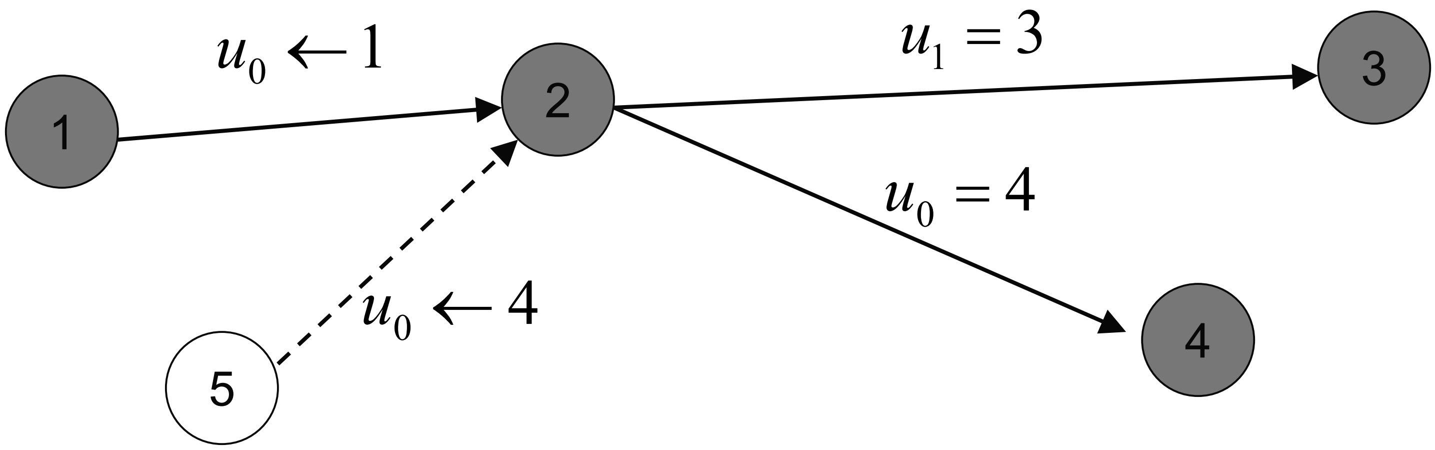
- Easier to generate NGAs than DGAs.
- Conversion takes two steps.
  - **Move compression**
    - Similar to epsilon closure.
  - **Subset construction**
- Each predicate has a readset.
  - **Variables it reads to evaluate.**
- Each action has a writeset.
  - **Variables it changes.**

Move Compression

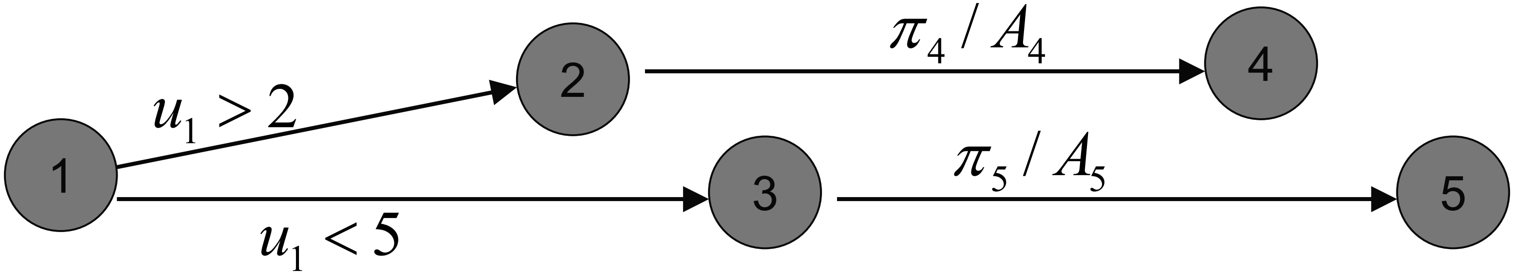


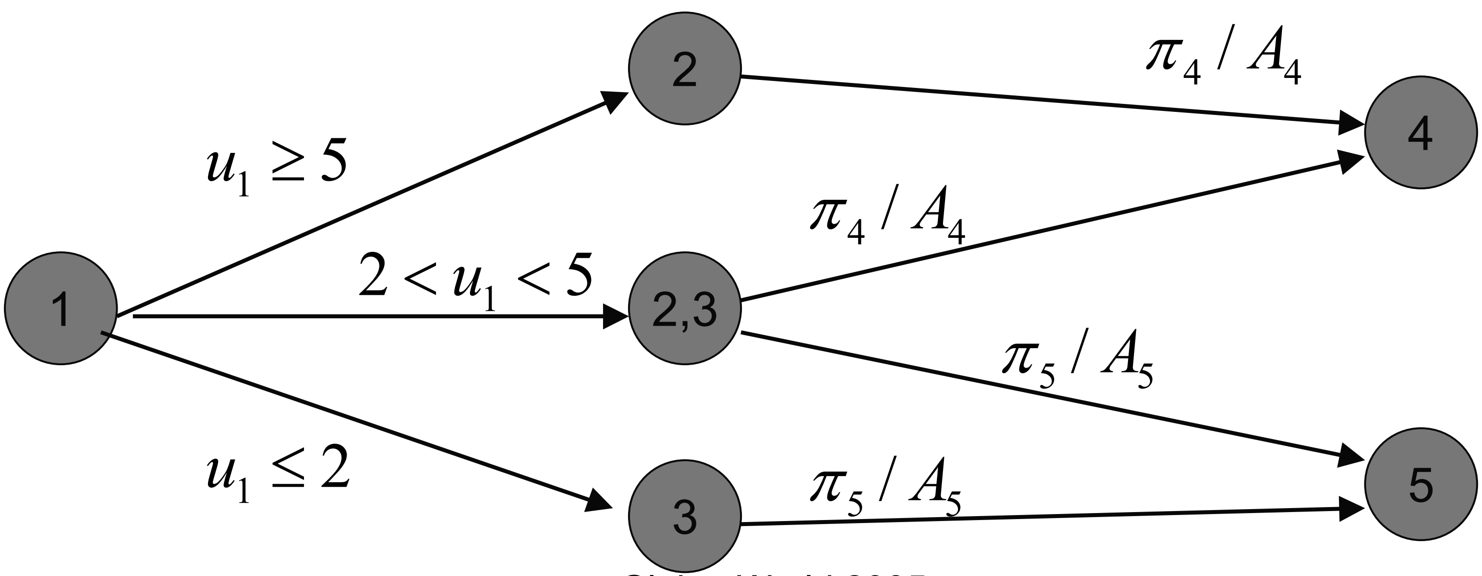


Move Compression



Subset Construction



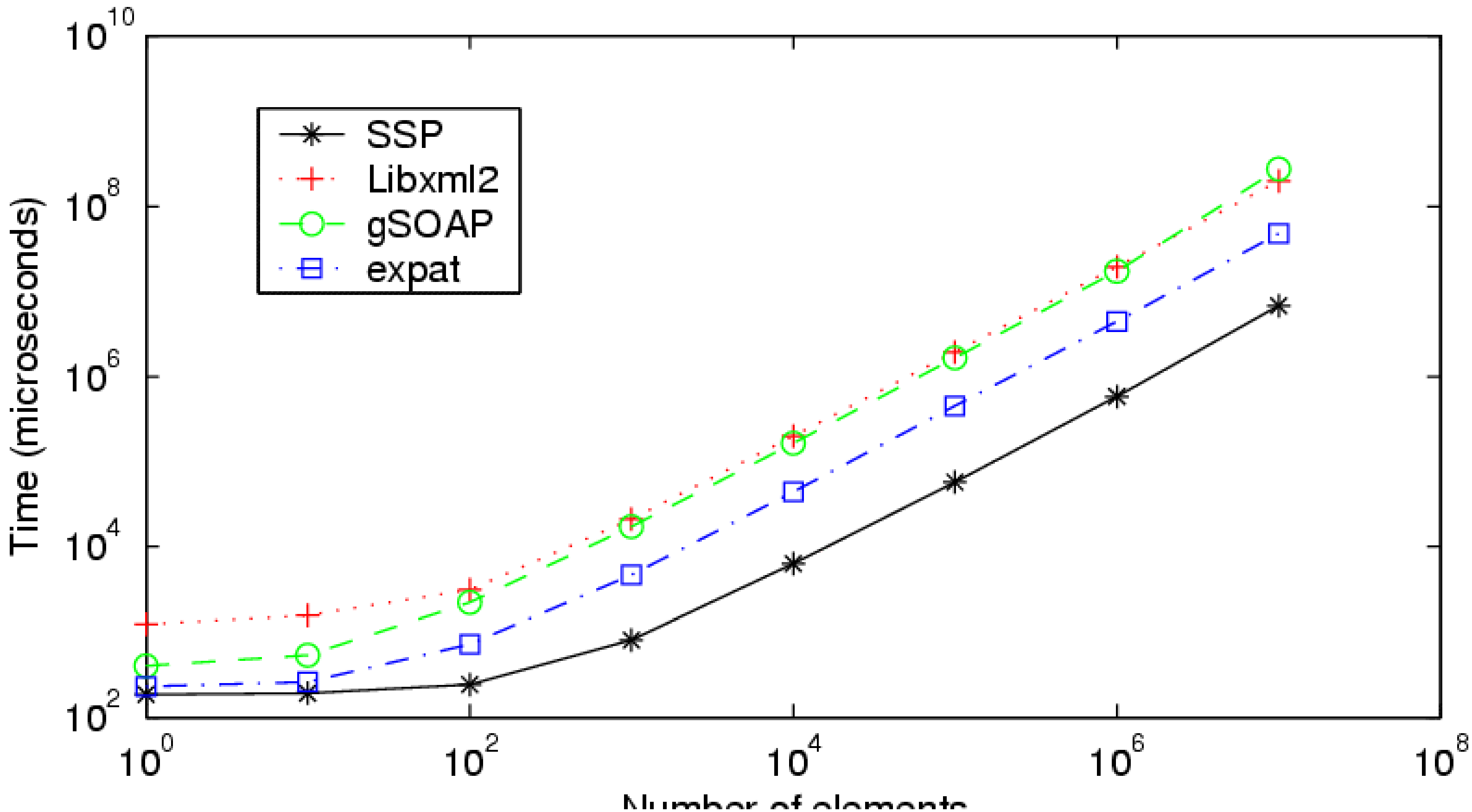


Performance Test

- Schema

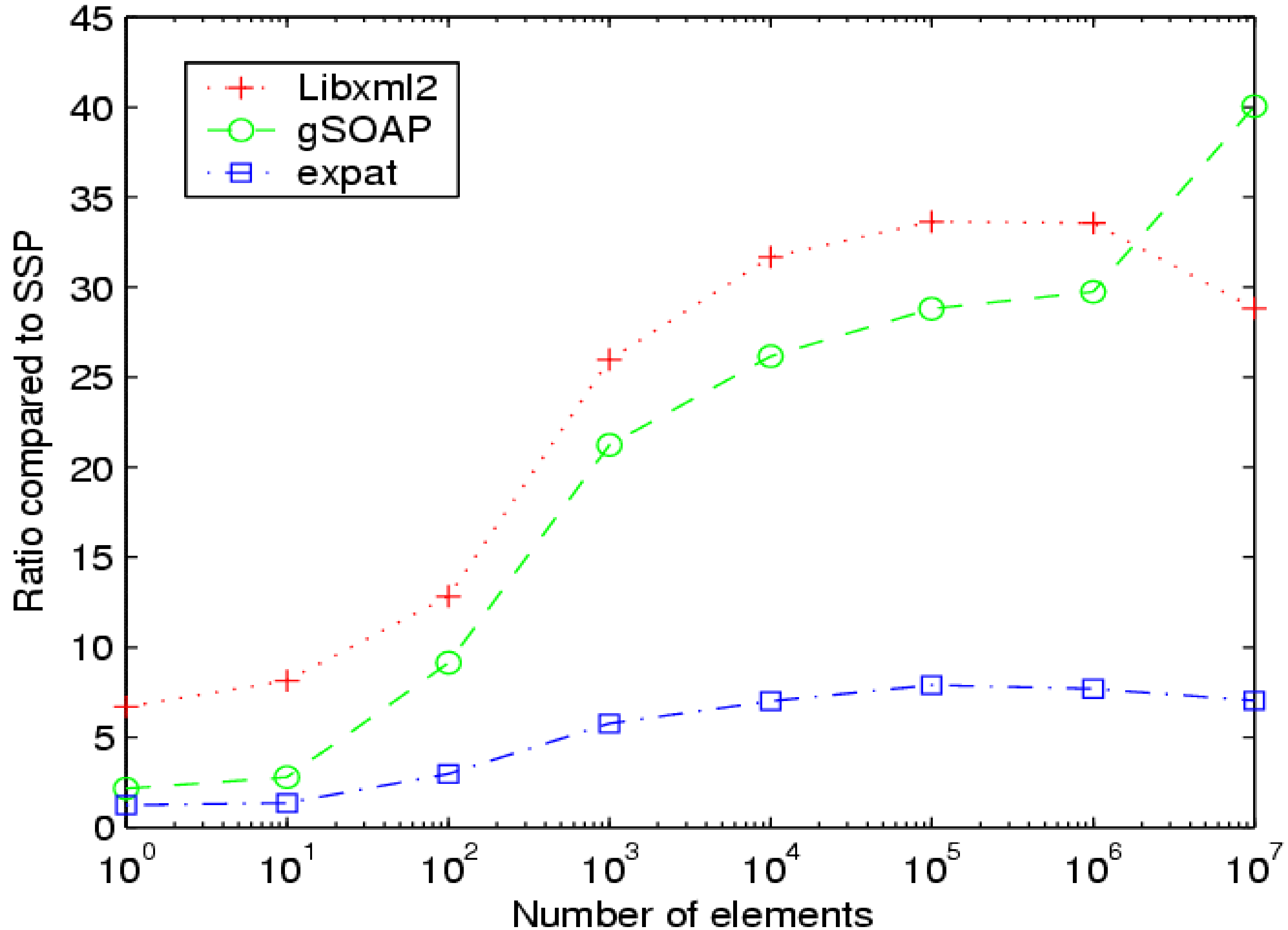
```
<schema>
  <complexType name="elemType">
    <choice>
      <element name="sub1" type="string"/>
      <element name="sub2" type="string"/>
    </choice>
  </complexType>
  <complexType name="topType">
    <sequence>
      <element name="elem" type="elemType"
       maxOccurs="N"/>
    </sequence>
    <attribute name="attr" type="string"/>
  </complexType>
</schema>
```

Results



SSP
Libxml2
gSOAP
expat
Time (microseconds)
10^{10}
10^{8}
10^{6}
10^{4}
10^{2}
10^{0}
10^{2}
10^{4}
10^{6}
10^{8}

Ratio to SSP



Libxml2
gSOAP
expat
Ratio compared to SSP
0
5
10
15
20
25
30
35
40
45
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
Number of elements

Conclusions

- Goal is to generate code as good as hand-written.
- Compile all the way down to low-level IL.
- Generalized automata seem to be an appropriate low-level IL.
- Preliminary results are encouraging, but not conclusive.
- Future work:
  - **More schema features, namespaces.**
  - **Optimizations.**
    - Outlining, reverse partial evaluation
    - Buffer precheck
  - **Higher-level IL?**
    - Enables different optimizations?
  - **Compiling to special architecture?**
  - **XSLT-like transforms? Given a transform that swaps two elements, can we generate code as efficient as can written by hand?**

Predictive XML Parsing with gSOAP

Robert van Engelen
Florida State University

The gSOAP Toolkit

- Project timeline

1/2000	v1.1	streaming XML; SOAP1.1 RPC
10/2001	v1.2	latency hiding; WSDL output
3/2002	v1.3	HTTPS SSL
1/2002	v2.0	MT/safe
7/2002	v2.1	DIME
12/2002	v2.2	streaming DIME
9/2003	v2.3	SOAP1.2
12/2003	v2.4	Doc/Lit; WSDL1.1 tools
1/2004	v2.5	WS-BP1.0a
3/2004	v2.6	
9/2004	v2.7	SwA; WS-*

>50,000 downloads

Early Versus Late Bindings

EARLY BINDING

performance

Static proxy generation with schema-specific DFA-based XML parsing

Static proxy generation with schema-specific PDA-based XML parsing

Static proxy generation with generic XML parsing

Dynamic proxy generation (DII) with generic XML parsing

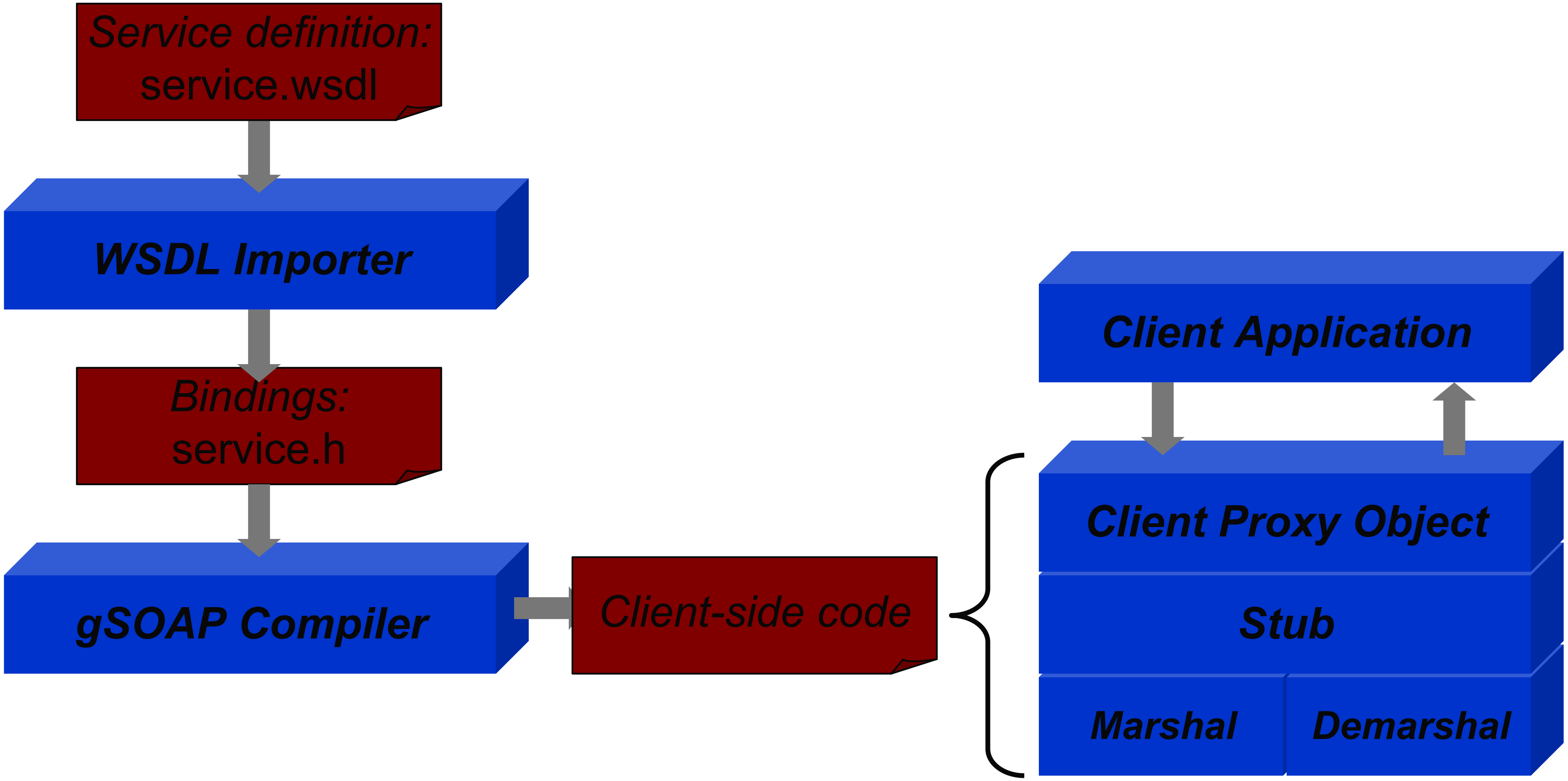
flexibility

LATE BINDING

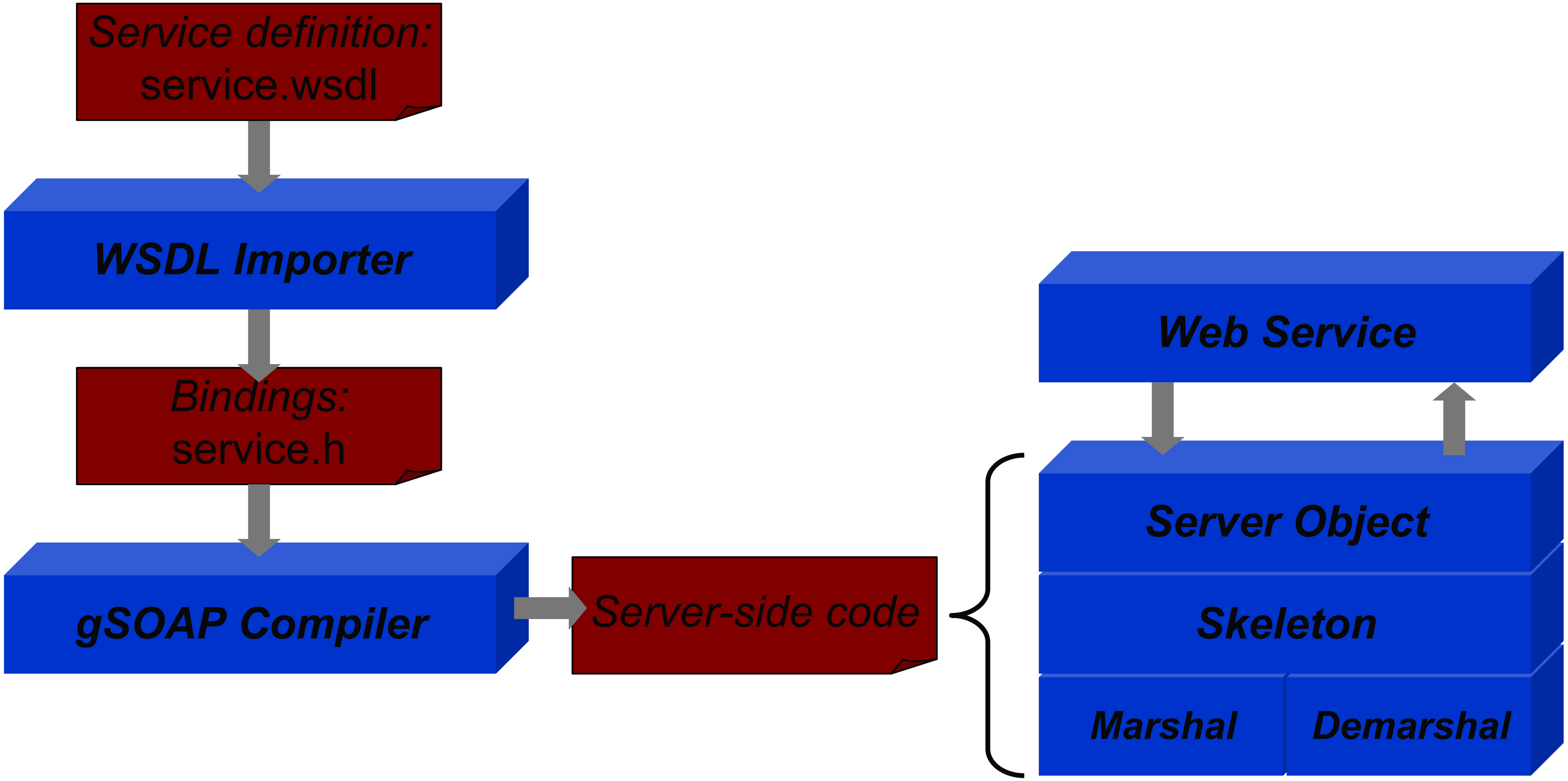
gSOAP Architecture

- Static binding
  - **WSDL tools to generate bindings**
  - **Stub/skeleton compiler to generate C and C++ code**
- Schema-specific predictive XML parsing
  - **Supports *in-situ* serialization and deseralization of application's native C/C++ data structures in XML**
- Integrated stacks
  - **TCP/IP - HTTP/S - DIME/MIME - SOAP/XML**
  - **Transport latency hiding**

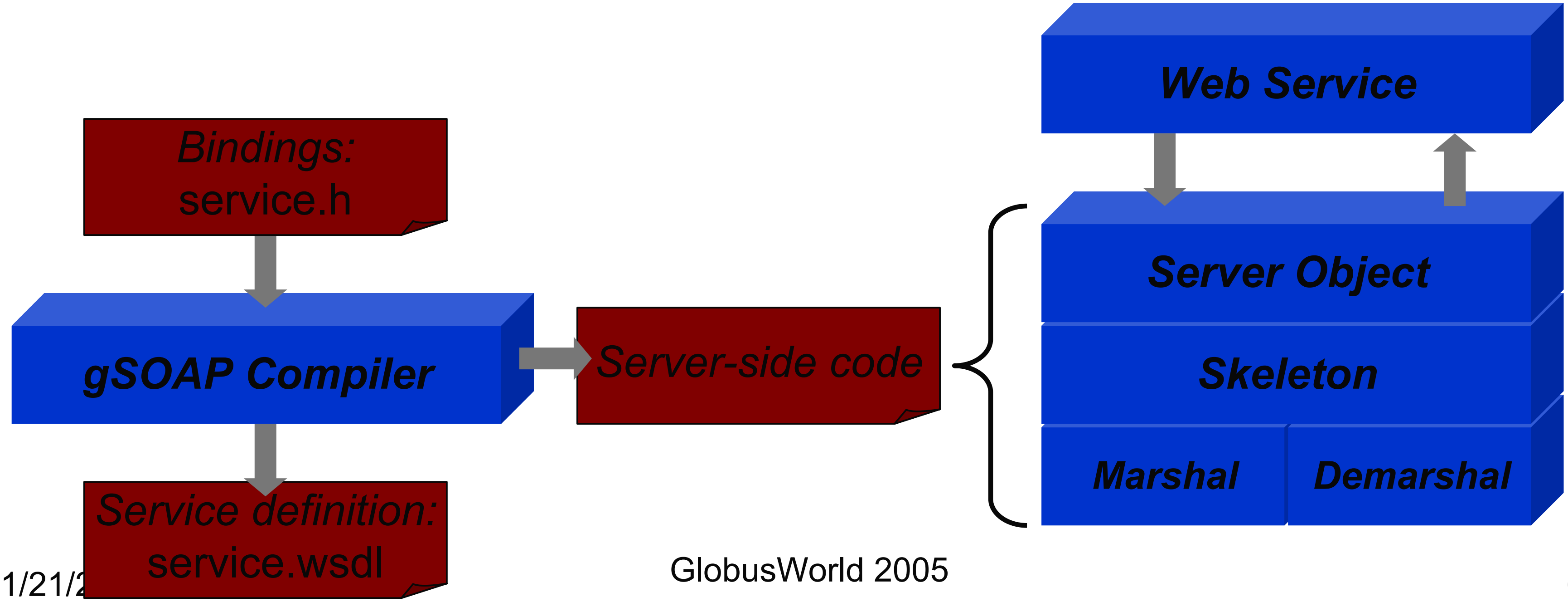
Client Application Development and Deployment



Server Development and Deployment



Server Development and Deployment (Alternative)



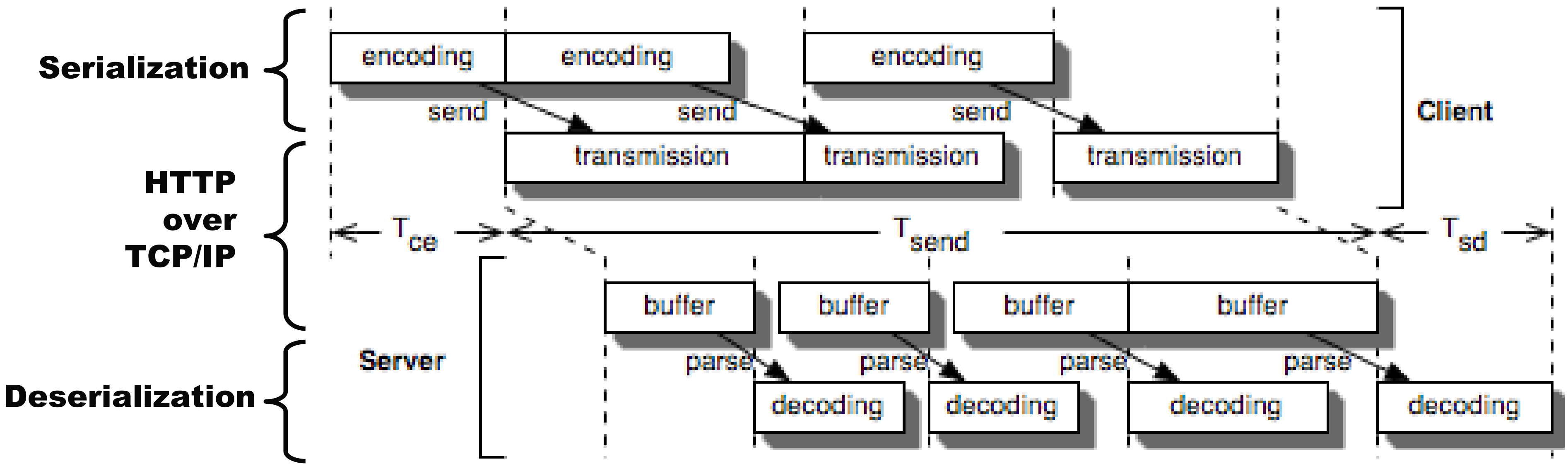
Schema-Specific Predictive XML Parsing


```
<complexType name="List">
  <complexContent>
    <sequence>
      <element name="item"
       type="xsd:string"
       maxOccurs="unbounded"/>
    </sequence>
  </complexContent>
</complexType>
```

```
class ns__List
{ std::vector<char*> item;
  int in(char* tag);
  int out(char *tag);
};
```

```
int ns__List::in(char* tag)
{ if (begin_element(tag) != OK)
    return TAG_MISMATCH;
  in_vectorOfstring(item, "item");
  end_element(tag);
}
```

Latency Hiding with Integrated Stacks



Latency and Speedup

Interop Round 2 Base echoVoid() latency

	gSOAP 2.4	XSOAP	AxisC++ alpha	.NET v1.1.4322	AxisJava v1.2
Latency (sec)	**0.0013**	**0.0016**	**0.0027**	**0.0034**	**0.0101**

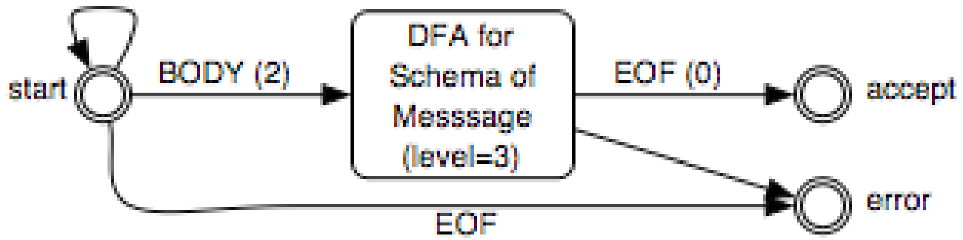
better

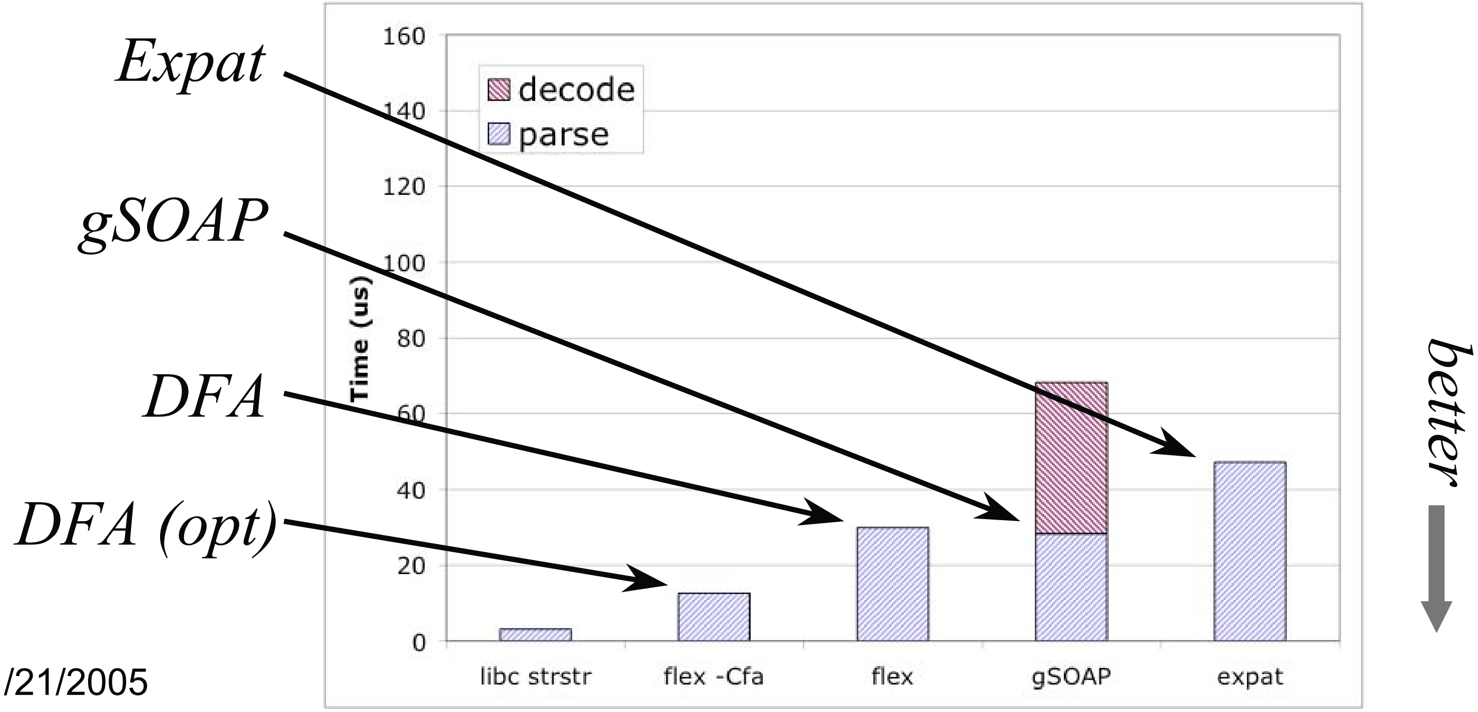
Relative average speedup for array-based SOAP messages (10 to 80,000 ints, doubles, and strings)

	gSOAP 2.4	XSOAP	.NET v1.1.4322	AxisC++ alpha	AxisJava v1.2
Speedup	**20.3**	**14.0**	**14.0**	**10.7**	**1.0**

better

DFA-Based Parsing





Conclusions

- Static bindings with predictive XML parsing delivers performance
- Two-stage compilation 1) bindings 2) code
- Integrated stacks to improve performance
- DFA-based parsing probably too limited for realistic applications
- More info: `http://gsoap2.sourceforge.net`

V4.0
WSRF-C
Performance Aspects

Sam Lang, ANL

GlobusWORLD

10 Feb 2005

GT4: Programming to Events

- Some Definitions:
  - **Event - System Call, I/O**
  - **Asynchronous - No ordering requirements for events, things happen when ready. Callbacks handle incoming events.**
  - **Non-blocking - A function that doesn't wait for an event to complete before exiting**
  - **Blocking - A function must wait for an event**
  - **Register - Mapping a handler to an event**
- GT4 WSRF-C Events
  - **Request/Response Sent**
  - **Request/Response Received**
  - **Notification (State Change)**

Event Programming Cont.

- Register for an Event
  - **A handler or callback function is written**

    myResourcePropertyCallback(ResourcePropertyValue val)
    {
    …
    }
  - **Callback is passed to a non-blocking register function**

    GetResourcePropertyRegister(endpoint, myResourcePropertyCallback);
  - **GetResourceProperty call gets a response, handler is called**
- Internals: Flavors and Threads
  - **Programming model internally manages threads**
  - **User must manage shared data**
  - **Can be built with/without threads**

Events and Performance

- Useful in Asynchronous Environments
- Performing Many WS operations
  - **In Sequence:**
    1. Send Request -> Wait -> Receive Response
    2. Send Request -> Wait -> Receive Response
    3. …
  - **Asynchronously:**
    1. Send Request A
    2. Send Request B
    3. …
    4. Receive Response A
    5. Receive Response B
    6. …

Events and WSRF

- Polling: WS-ResourceProperties
  - State is exposed by ResourceProperties
  - State is distributed in grid environments
- Pushing: WS-Notifications
  - Notifications are events
  - Implement a callback handler for notifications
  - Subscribe to Notification Topics (maybe RPs) and register callback for notifications
  - Many notifications, one callback
- Web Service Container
  - Invocations trigger event handling code, calling service impl

Performance Numbers

- Many GetResourceProperty operations
  - **In Sequence:**
  - **Asynchronously:**

PyGridWare
Performance Aspects

Keith R. Jackson
Lawrence Berkeley National Laboratory

Overview

- PyGridWare is a Python based implementation of the WSRF and WS-Notification specifications.
- Builds on top of the Python open-source SOAP toolkit ZSI.
- Uses XML tooling from both 4Suite and the Python standard library.
  - **Much of the underlying tooling is written in C.**
- Main development focus has been BP-1.1 and WSRF compliance, not performance.
  - **But …**

Initial Experience

- When we first looked at performance, our numbers were abysmal!
  - **Completely unacceptable for any real world usage.**
- Profiler showed we were defaulting to a Python based XML parser for parsing.
- Switching to 4Suite's cDomlette increased performance approximately 20 times.
  - **Adequate for now, but still not fast enough.**
- Shifting to an event driven container also made a huge difference.
  - **Based on the Twisted project.**

Current Performance

- Perf data for 100 add ops with breakdown of hotspots. W/wo security.

Planned Improvements

- Still major hotspots in the current code.
  - **Namespace handling**
  - **c14n**
- Evaluate the other XML toolkits with Python bindings.
  - **libxml2**
- Consider developing Python bindings to the GT WSRF-C asynchronous SOAP parser.
- Use C based implementations where possible to eliminate hotspots, e.g., c14n, http transport.

Conclusions

- Adequate performance is critical to the success of WSRF.
  - **Most of the overhead is in XML serialization and parsing (about 2 to 1 serialization to parsing).**
- We are focused on producing a standards compliant WSRF toolkit.
  - **Very interested in ongoing work in improved XML parsing techniques.**
- Hopefully we can take advantage of the great work others have described here today!