LYE: a high-performance caching SOAP implementation

Daniel Andresen*, David Sexton, Kiran Devaram, Venkatesh Prasad Ranganath,
Department of Computing and Information Sciences
Kansas State University
{dan, dms3333, kiran, rvprasad}@cis.ksu.edu

Abstract

The Simple Object Access Protocol (SOAP) is a dominant enabling technology in the field of web services. Web services demand high performance, security and extensibility. SOAP, being based on Extensible Markup Language (XML), inherits not only the advantages of XML, but its relatively poor performance. This makes SOAP a poor choice for many high-performance web services. In this paper, we present a new approach to implementing the SOAP protocol using caching on the SOAP server. This approach has significant performance advantages over current approaches while maintaining complete protocol compliance. We demonstrate its practicality by implementing a demonstration system under Linux, giving speedups of over 250% in our sample applications.

1. Introduction

Recently, there has been tremendous development in the area of web services in eCommerce, high-performance computing, and Computational Grid. In response to the need for a standard to support web services, SOAP became the standard binding for the emerging Web Services Description Language (WSDL) [12, 13]. SOAP is based on XML [2] and thus achieves high interoperability when it comes to exchange of information in a distributed computing environment. While carrying the advantages that accrue with XML, it has several disadvantages that restrict its usage. SOAP calls have a large overhead due to the considerable execution time required to process XML messages. In this paper, we partially mitigate a primary negative of SOAP: its speed of execution. We do this through the selective implementation of caching on the server side, feeling that a number of applications repetitively send the same information, often in a structured form. Examples might include "stock tickers," game broadcasts, or airline ticket pricing. Each of these is likely to send the same information multiple times, yet the information is also continuously changing, so a simple reverse proxy cache is inadequate.

In our previous work, we implemented caching on the client-side SOAP engine, and achieved speedups of over 800% for the client [8, 7]. In this paper, we optimize the server-side processing of a SOAP request, achieving speedups of 250% for structured datatypes and achieving at least a small optimization for all transactions. We use the Java implementation of in Tomcat 5.0.14, and choose the most normal model of SOAP that is used in distributed software, the RPC-style. This choice is common among Web developers, as it closely resembles the method-call model.

Detailed information about SOAP and how it is used in web services is presented in Section 2, as well as a discussion of other efforts in the field. In Section 3, we present our analysis on the factors and potential of caching within the SOAP architecture. We then discuss our implementation in Section 4, which is followed by the results of the experiments we conducted to compare performance of several variations of our algorithm. We will present our conclusions and future work to extend/improve these algorithms in Section 6.

2. Background and related work

We use HTTP as the underlying protocol for transporting SOAP XML payloads, although it is not mandatory according to the SOAP specification. Binding SOAP to HTTP provides the advantage of being able to use the formalism and decentralized flexibility of SOAP with the rich feature set of HTTP.To send a request to the server, the SOAP RPC client creates an instance of org.apache.soap.rpc.Call, a Java class that encapsulates a SOAP RPC method call. After specifying the name of the service and the method being invoked, we set the parameters, which in this case are the names of the two cities, using the setParam() method of the Call object. The actual communication with the server is done with the use of the invoke() method of the Call object to make a method call to the server. Fig. 1 shows the SOAP payload that the client generates. Being in ASCII text, this message

is very large in size when compared to a similar request of a JavaRMI client. Note that the source and the destination cities are stored in the *From* and the *To* tags of the SOAP payload.

Upon examination of the profile data of this SOAP RPC client, it is found that, about 50% of the execution time is spent in XML encoding and creating a HTTP connection[7]. XML encoding involves preparation of the SOAP payload, which is basically serializing and marshalling of the payload before it is transmitted to the server.

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: hostname
Content-Type: text/xml;charset=utf-8
Content-Length:
SOAPAction: ""
Accept-Encoding: gzip

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<ns1:getFlightInfo xmlns:ns1="urn:FlightInfoService"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<From xsi:type="xsd:string">Madison</From>
<To xsi:type="xsd:string">Las Vegas</To>
</ns1:getFlightInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 1. SOAP payload generated by SOAP RPC client.

Comparing several such requests from the client, it is found that the SOAP payloads differ only in the values of the *From* and the *To* tags (Figure 1). For each such request, the client has to prepare the SOAP payload, which takes significant amount of processing time involving XML encoding. From this observation, we figure out that, there can be a better way to handle similar multiple calls made from our clients.

There have been several studies comparing SOAP with other protocols, mainly binary protocols such as Java RMI and CORBA. All of this research has proven that SOAP, because of its reliance on XML, is inefficient compared to its peers in distributed computing. In this section we examine studies [3, 6, 4] which explain where SOAPs slowness originates and consider various attempts to optimize it.

SOAP, relying heavily on XML, requires its wire format to be in ASCII text. This is the greatest advantage of using SOAP, as the applications need not have any knowledge about each other before they communicate. However, since the wire format is ASCII text, there is a cost of conversion from binary form to ASCII form before it is transmitted. Along with the encoding costs, there are substantially higher network-transmission costs, because the ASCII encoded record is larger than the binary original [3]. Reference [3] shows that there is a dramatic difference in the amount of encoding necessary for data transmission, when XML is compared with the binary encoding style followed in CORBA.

Extreme Lab at Indiana University [4] developed an optimized version of SOAP, namely XSOAP. Its study of different stages of sending and receiving a SOAP call has resulted in building up of a new XML parser that is specialized for SOAP arrays, improving the deserialization routines. This study employs HTTP 1.1, which supports chunking and persistent connections.

Reference [11] states that XML is not sufficient to explain SOAPs poor performance. SOAP message compression was one attempt to optimize SOAP; it was later discarded because CPU time spent in compression and decompression outweighs any benefits [11]. Another attempt in [11] was to use compact XML tags to reduce the length of the XML tag names. This had negligible improvement on encoding, which suggests that the major cost of the XML encoding and decoding is in the structural complexity and syntactic elements, rather than message data [11].

In Reference [1], O. Azim and A. K. Hamid, describe client-side caching strategy for SOAP services using the Business Delegate and Cache Management design patterns. Each study addressed pinpoints an area where SOAP is slow compared to its alternatives. Some present optimized versions of SOAP using such mechanisms as making compact XML payload and binary encoding of XML. While said mechanisms achieved better efficiency, none could match Java RMIs speed and simultaneously preserve compliance to the SOAP standard.

3. Cost Analysis

The previous section provides numbers that indicate that marshalling incurs the highest cost while processing a SOAP message processing. In this section we shall consider various steps involved in processing a SOAP message, associate cost functions to these steps, and illustrate the impact of possible caching-based optimizations based on the changes to the cost functions. These can be broken down into the following steps.

1. Reception of the request by the router,
2. Identification of the provider and the dispatch of the call to the provider
3. Dispatch of the call to the actual service and reception of the subsequent response,
4. Marshalling of the envelope, and
5. Marshalling of the response.

Of these steps, 1 and 3 cannot be optimized, while the rest can be optimized via caching. If there are multiple re-

quests for the same data then Step 2 can be optimized by caching the marshalled form of the response or marshalled response [1] for each distinct request. A request is *distinct* from another request if either the service differs or any of the arguments in the request differ in value. However, *staleness* of response has a major impact on the correctness of the approach and the performance of cache. Hence, in this exposition we assume that there is suitable logic to ensure staleness constraints on the cached data is honored[2]. We introduce 3 variables to capture the performance cost incurred in step 2 and beyond.

$bCacheLT$ is the time taken to lookup the marshalled data for a given response object,

$t_{dispatch}$ is the time taken to dispatch the call to the provider,

$t_{marshall}$ is the time taken to prepare the marshalled response based on the response, and

$bCacheMiss$ is the probability of cache misses.

Hence, the total cost of processing per message is given by $cost(1) = bCacheLT + (t_{dispatch} + t_{marshall}) \times bCacheMiss$ as the dispatch and marshalling cost are only incurred when there is a cache miss in step 2.

We have observed that the envelope for a service is independent of the request, hence, it is possible to cache the envelope in it's marshalled form and avoid redundant marshalling operations in step 4. This leads us to a cost function $cost4(1) = envelopeMT \times (1 - cacheEnabled)$ where *cacheEnabled* ranges over *0* indicating caching in step 4 is disabled and *1* indicating caching in step 4 is enabled.

In step 5, we have identified 2 possible caching strategies given below.

Marshalled response caching If marshalled response is cached, upon receiving a response from the provider the router checks if there is a marshalled form of the given response in the cache. If so, the marshalled form is used. If not, a new marshalled response is generated from the given response and cached.

Marshalled response template caching For a service, the structure of the marshalled response does not change between any two similar responses[3]. Also, marshalling of response involves wrappers that wrap the serialized form of primitive data types. Hence, it is possible to create and cache a template of the marshalled response that contains the required wrappers with "holes" in them and the serialized form of the primitive data components of response can be injected into the "holes" while instantiating the template. When compared to the previous strategy, this strategy *should* incur more cost in terms of template instantiation.

As both the above strategy involve cache lookups based on responses and as responses are in general complex data types, the cost associated with lookup is non-negligible when compared with lookup based on primitive data type. The cost is incurred by "walking/visiting" each component of the response in order to check if it exists as a key in the cache. The situation would be similar even if we were to resort to hash-based cache lookups but the cost would probably increase by a multiplicative constant when there are collisions during hashing. This cost is also incurred while marshalling a response in non-cached scenario. Hence, we capture the cost of exploring the response as $t_{explore}$ and the sum of cost of marshalling each component of a response as $t_{responseMTime}$. Let *eCacheMiss* be the probability of cache misses in step 5 and $t_{instantiateTime}$ be the cost of instantiating a template of a marshalled response. From these, the total cost of processing per message in step 5 can be calculated as $cost5(1) = t_{explore} + t_{responseM}$ when no caching (NC) is used, $cost5(1) = t_{explore} \times (c_1 + eCacheMiss) + t_{responseM} \times eCacheMiss$ when ***marshalled response caching(MRC)*** is used and $c_1 \geq 1$, and $cost5(1) = t_{explore} + t_{instantiate}$ when ***marshalled response template caching(MRTC)*** is used.

Upon substituting *cost5* and *cost4* in *cost1* we get the following cost functions for processing n messages.

$$
\begin{aligned}
cost1_{NC}(n) = \; & (bCacheLT + (t_{dispatch} + cost4(1) \\
& + \underline{t_{explore} + t_{responseMTime}}) \\
& \times \underline{bCacheMiss}) \times n \\
cost1_{MRC}(n) = \; & (bCacheLT + (t_{dispatch} + cost4(1) \\
& + \underline{t_{explore} \times (c_1 + eCacheMiss)} \\
& \underline{+ t_{responseMTime} \times eCacheMiss}) \\
& \times \underline{bCacheMiss}) \times n \\
cost1_{MRTC}(n) = \; & (bCacheLT + (t_{dispatch} + cost4(1) \\
& + \underline{t_{explore} + t_{instantiate}}) \\
& \times \underline{bCacheMiss}) \times n \\
& + \underline{t_{marshall}}
\end{aligned}
$$

In the above functions, the underlined factors of the equation strongly influence the overall cost. Hence, we shall compare each of these equations based on the underlined terms.

NC vs MRC When $eCacheMiss \ll 1$, MRC outperforms NC. This happens in cases when the request is fixed over time in terms of the values of its components. Hence, approach MRC should scale elegantly with support to enforce staleness constraints or well tuned

1 From here on, we will refer to the marshalled from of the response as *marshalled response*.

2 This is part of our future work.

3 Responses are similar if the termination condition of the dispatch is the same.

cache flushing logic. Either of these two features is required to control the growth rate of the cache as it is directly proportional to the number of distinct responses.

NC vs MRTC In this case MRTC should outperform NC with marginal memory overhead when $t_{instantiate} \ll t_{responseM}$, since the cost of instantiating a template should be relatively smaller than marshalling the entire response.

MRC vs MRTC In this case *eCacheMiss* decides the validity of the pivotal relation $t_{instantiate} \ll t_{responseM} \times eCacheMiss$. Assuming $t_{instantiate}$ and $t_{responseM}$ are comparable, the relation holds in case of services in which the high rate of change of response leads to increased *eCacheMiss* value. In such situations MRTC is bound to outperform MRC. If the rate of repetition of response is high then it leads to decreased *eCacheMiss* value, hence, MRC is bound to outperform MRTC. However, if $t_{instantiate}$ is considerably smaller than $t_{responseM}$ then the performance of MRC may be comparable that of MRTC. Independent of the relation, MRTC has a better memory cost when compared MRC as MRTC' cache growth rate is proportional to the number of services used in a period of time while MRC' cache growth rate is proportional to the number of distinct responses in a period of time.

From the above inferences, we can conclude that caching of responses at various stages can improve the performance of a SOAP server. We can also conclude, and verify in Section 5, that *marshalled response template caching* can provide in improvements in both, time and space, when compared to *marshalled response caching*.

4. Implementation

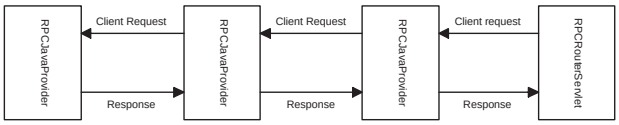

Figure 2. Standard SOAP RPC flow of control

Our intent was to create multiple SOAP caching implementations with minimal changes to the current SOAP architecture. We were able to use the flexibility of the current SOAP architecture with slight modifications to develop our caching strategies. As per design, the client request processing is controlled by the provider implementation. The provider can be grouped with 4 other classes that constitute the flow of a request as it passes through the server (See Figure 2).

The first stage of execution is related to the RPCRouterServlet. This class accepts a client request message and transforms it into Java objects. The client request contains the provider name along with the service method name and parameters. The provider is then located using the information supplied by the client request.

The client request is then passed to the provider who forwards the request to the RPC router, the second stage of execution. The router is responsible for finding and executing the service requested by the client.

The third stage of execution is the service implementation. The service is a custom implementation not associated with the server. In our experiment, the available services consisted of generating and returning simple and complex data types. The value returned from the service is wrapped in an org.apache.soap.Response object and returned to the provider.

The provider is the last stage of execution. The provider controls the request processing and response generation and, therefore, a majority of the execution time required to complete process the client request is within the provider. The provider interface allowed us to use the existing org.apache.soap.provider.RPCJavaProvider as a model to develop our own providers for each of the three caching strategies. There was a need for some slight modifications and additions, but all the existing method signatures remained the same.

The RPCJavaProvider implementation itself can be broken into four different sections. These sections consist of invoking the service, building an envelope, marshalling the envelope and setting resulting xml into the current context. The profiled data from the RPCJavaProvider showed a large percentage of the time spent by the provider was in the XML encoding or marshalling process. If there was a scenario where the service was to produce the same response multiple times, this would prove costly and inefficient. The same response xml would need to be regenerated for every client request.

To avoid the regeneration of the same response xml, a cache was constructed for use with every new provider implementation. This cache implementation is based on a java.util.Hashtable. The specifics on how the cache was used and the modifications required to support the cache is stated with each provider implementation.

4.1. Complete Caching

The first of three caching implementations consists of complete caching of the response object. In this implementation, a unique client request is received and a response

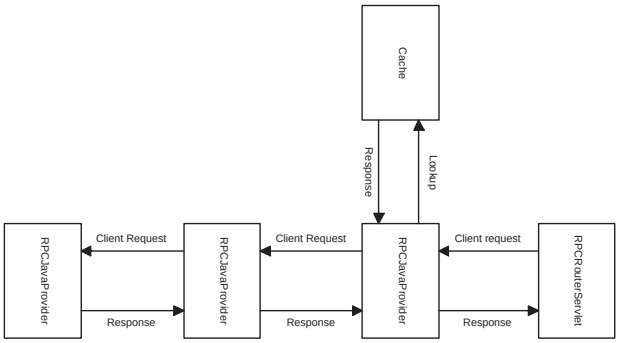

Figure 3. Complete Caching flow of control

generated. Once the response has been marshaled into xml form, it is cached using the client request as the key. This will ensure the response will be correct for an identical client request. This will also negate the time required to generate a response for a non unique client request.

The cache exists within the provider and is keyed by the client request encapsulated in a Call object generated from the client request. The Call object is used as the key to provide that every aspect of the client request is used to identify the correct response stored in the cache. In addition to other objects used to encapsulate the client request, have the equals and the hashkey methods overwritten from to allow the cache to function correctly. The complete caching strategy allowed the server to skip several stages, including the service execution and the response marshalling.

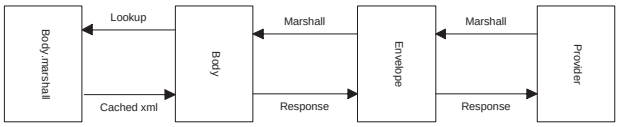

Figure 4. Body Caching flow of control

4.2. Body Caching

For the next implementation, we started to focus on the sections of the response xml message. We noticed that the body element within the response did not change for the same values returned by the service. If the Body element was cached, there would be no need to remarshaling or serialization of any repeated values returned from the service.

The body section of the xml message was cached using the response from the service as the key. The Response object returned from the RPCRouter will be used as the key. This will guarantee the cached body element is correct for the response value.

There were several object modified in this implementation. For the Hashtable to function correctly, the equals and hashkey methods were added to Response and org.apache.soap.rpc.Parameter. The Parameter object contains the return value from the service. The hashkeys returned by the Response object were the hashkeys of the return values of the service.

The objects in the envelope hierarchy also needed to be modified to contain the cache. The cache existed within the Response node, but also required a modification to the Body element.

4.3. Envelope and Body Caching (Template)

In the last implementation, we also realized that for every request message sent by our clients, the envelope element of the response did not change. This allowed us to keep a static copy to avoid the process of marshalling the envelope object. We continued to cache the Body element as with the Body Caching implementation described above. We are again focused on avoiding the remarshaling of any part of the response message that will not change from an identical response.

The Envelope object required the only additional modification in this implementation. The marshalling code was removed and replaced with a static envelope header definition.

5. Experimental results

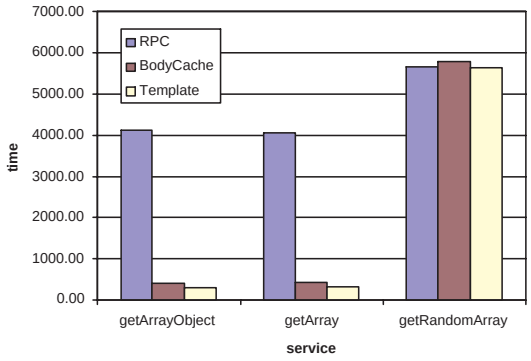

Figure 5. Relative times to marshal array objects for an array of objects, a repeated array of integers, and an array of random integers. (Pentium 4 system, time in ticks)

Our experimental results are positive. We were able to reduce the amount of time to generate a response in each of

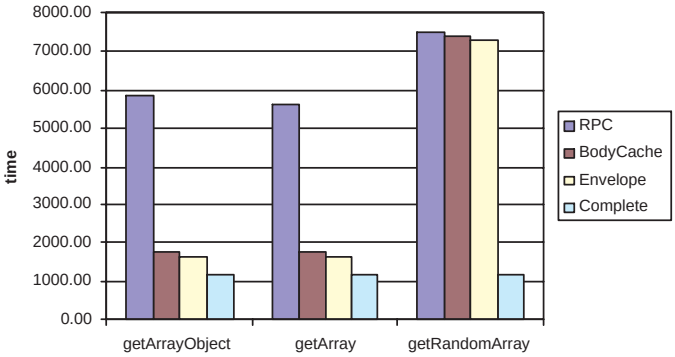

Figure 6. Relative times to completely fetch array objects for an array of objects, a repeated array of integers, and an array of random integers. (Pentium 4 system, time in ticks)

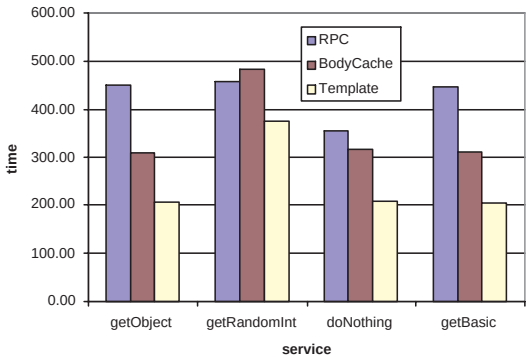

Figure 7. Relative times to marshall simple and compound objects. Responses include a simple compound object, random integer, null response, and integer. (Pentium 4 system, time in ticks)

the caching implementations. The different provider implementation performed as expected with the complete caching strategy providing the largest speedup followed by the envelope plus body caching and body caching only.

Environment For each provider test, the same client configuration was used. The only difference between each of the client requests was the provider name in the envelope payload. For each test run, the different service calls were grouped together. Each provider client was allowed to make a request for a single service. This was repeated 25 times for each service. Before starting a new service grouping, the cache was cleared for each of the caching provider implementation.

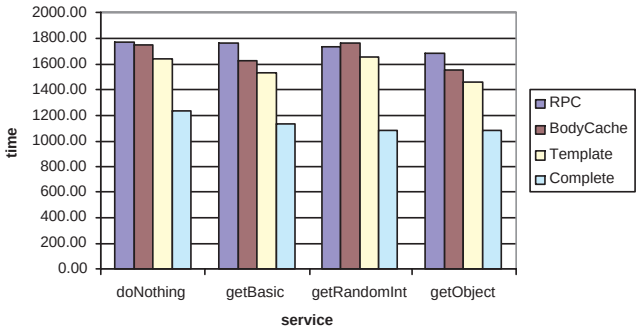

Figure 8. Relative times to completely fetch simple and compound objects. Responses include a simple compound object, random integer, null response, and integer. (Pentium 4 system, time in ticks)

The services provided in our experiment were designed to return a variety of different values. The values returned where void, int, string, an array of ints and an array of objects (See Figures 8 and 7). The arrays returned always contained 100 elements.

Our software environment was Tomcat 5.0.14, J2SDK 1.4.2_02, and J2EE 1.3.1 running under Linux 2.4.x kernels on a 500Mhz. Pentium III, 1.8Ghz. Pentium 4, and dual Athlon MP 1800+. Each machine had sufficient RAM that paging was not an issue. All results are from the Pentium 4 system unless otherwise noted.

We used the sun.misc.Perf timer included in the j2sdk 1.4.2. This timer gave us a resolution of 1,000,000 per second. The time between sequential calls to the timer was less than 5 ticks on all machines.

Caching strategies The four providers used in the experiment where the RPCJavaProvider, BodyCacheProvider, TemplateCacheProvider and CompleteCacheProvider. Each of the last three providers included a Caching strategy as mentioned in Section 4. The RPCJavaProvider was provided by the original SOAP implementation. All of the caching strategies provided in this experiment provided positive results with all producing faster response times, with the exception of the random services, where in one case a small (¡2%) performance slowdown was noted.

- *CompleteCache* There was no need for work within the server for the complete caching provider implementation, with the exception of a cache lookup and retrieval. There client request message was service in an average of around 65% of the time required by the RPCProvider for basic response types and an average of around 20% for an array, for a speedup of over 500% (Table 1). This caching strategy, however, blocks com-

	null	*Basic*	*Random int*	*Object*	*Array Object*	*Array*	*Random Array*
SOAP/RPC	1767	1760	1732	1689	5860	5608	7485
Body	1750	1629	1759	1553	1783	1780	7382
Template	1633	1533	1656	1454	1637	1643	7290
Complete	1234	1136	1085	1082	1164	1149	1149

Table 1. Average response time for retrieving various types of objects under different caching strategies (Pentium 4 system, time in ticks).

munication with the service and ignores of any possible changes that might have occurred with the service response.

- *Envelope & body caching* The Envelope and Body caching strategy show the next best performance gain. In this implementation, there was no need to regenerate the Envelope and Body elements for the xml response message. However, this strategy was sensitive to the changes in any changes in a response from the service and the service was allowed to execute normally. The time required by this provider was 75 – 92% of the RPCProvider profile for basic response types and around 28% for an array. The Random services are not included in these numbers because they negate the use of a cache. They still show a small performance gain, however, because of the Envelope caching.

- *Body only* Our last implementation consisted of a Body only caching strategy. As the name suggests, the Envelope element is not cached. Our interest was the caching the serialized value returned from a service. The time required by this provider was 91 – 99% of the RPCJavaProvider profile for basic response types and around 31% for an array. The random services in this implementation produced times that were higher than expected, exceeding the time required by the RPCJavaProvider by a small percentage. This is as expected since the only time difference between this provider implementation and the RPCJavaProvider implementation is the addition of a cache lookup and adding the body element to the cache (Figures 9 and 10).

The numbers mentioned above was the total time required to service each client request. In theory, the only values that should have changed in each implementation should have been the time required to marshal the data. If you compare the marshalling time (Figures 5, 6, 7, and 8) required by the Body only and Envelope and Body caching strategies to the RPCJavaProvider marshalling, you will see a performance gain of around 10% and 15% respectively for basic values and around 250% for arrays.

Scalability We can also see from Figure 11 that, as might be expected, the scalability of the server is significantly en-

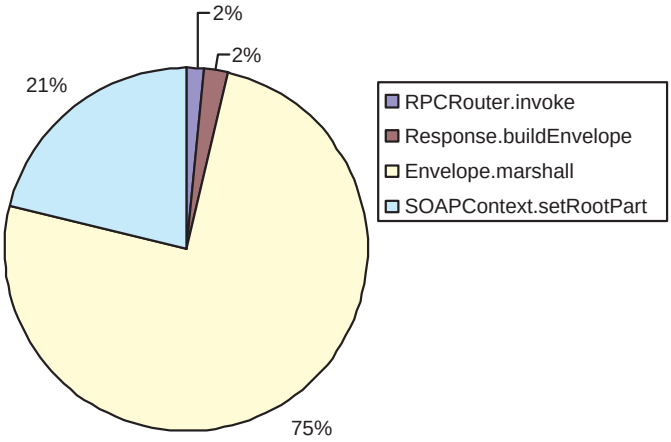

Figure 9. Time distribution for fetching an array under SOAP RPC. (Pentium 4 system)

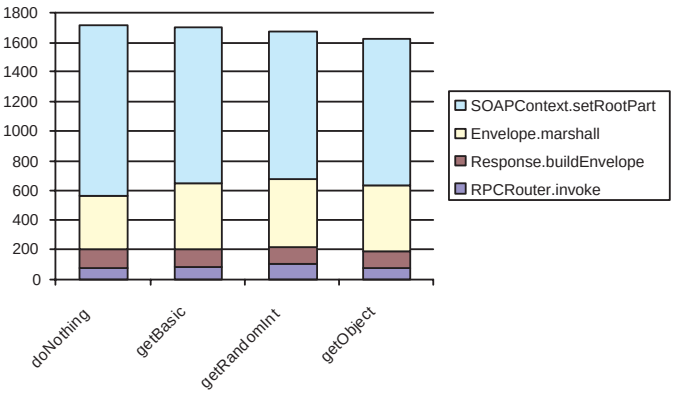

Figure 10. Time distribution for fetching simple objects. (Pentium 4 system)

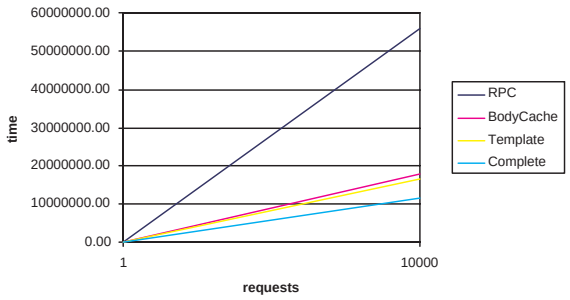

Figure 11. Response times of various algorithms over varying loads servicing a burst of simple requests.

hanced through lower processing loads. The overhead imposed by the cache lookup times has been measured to be practically negligable, so almost the full benefit of reduced XML data handling is available to improve overall server performance.

6. Future Work and Conclusion

We are currently working to enhance our system in two major areas. First, we plan to implement subelement-level caching, allowing partial caching to occur even if a subset of the subelements in a reply have been modified from a previous call. We anticipate this will give significant improvements for SOAP communication in which many elements are constant, but a few change regularly. Second, we plan to explore applying partial evaluation (aka program specialization) to produce optimized response modules for SOAP-based web services [10, 9, 5]. We are also exploring various cache management options, and the possibility of submitting our code for inclusion into the primary Apache code base.

In this paper we have presented a new approach for accelerating the performance of a vital portion of the eCommerce infrastructure through the use of directed caching of responses. Our theoretical analysis predicts, and experimental results confirm, that this approach can give substantial speedups (250%+) for many practical applications, while exacting no performance penalty for applications unsuited to the architecture beyond the memory devoted to the cache.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under the award numbers CCR-0082667 and ACS-0092839. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] O. Azim and A. Hamid. Cache SOAP services on the client side. *JavaWorld: IDG's magazine for the Java community*, Mar. 2002. http://www.javaworld.com/javaworld/jw-03-2002/jw-0308-soap.html.

[2] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. *W3C*, Feb. 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[3] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient wire formats for high performance computing. In *Proceedings of Supercomputing 2000*, pages 64–64, 2000.

[4] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02)*, page 246. IEEE Computer Society, 2002.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, June 2000.

[6] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 407–412, 2002.

[7] K. Devaram and D. Andresen. SOAP optimization via client-side caching. In *Proceedings of the First International Conference on Web Services (ICWS 2003)*, pages 520–524, Las Vegas, NV, June 2003.

[8] K. Devaram and D. Andresen. SOAP optimization via parameterized client-side caching. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, pages 785–790, Marina Del Rey, CA, Nov. 2003.

[9] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proceedings of the 1999 ACM Workshop on Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, Jan. 1999.

[10] J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997.

[11] C. Kohlhoff and R. Steele. Evaluating SOAP for high performance business applications: Real-time trading systems. In *Proceedings of WWW2003*, Budapest, Hungary, 2003.

[12] Simple object access protocol (soap) 1.1, Feb. 2003. http://www.w3.org/TR/SOAP/.

[13] Web Services Description Language (WSDL), 2001. http://www.w3.org/TR/wsdl.