Semantic Caching for Web Services*

Stefan Seltzsam[1], Roland Holzhauser[2], and Alfons Kemper[1]

[1] TU München, D-85747 Garching, Germany, ⟨first name⟩.⟨last name⟩@in.tum.de
[2] Universität Passau, D-94030 Passau, Germany, holzhaus@fmi.uni-passau.de

Abstract. We present a semantic caching scheme suitable for caching responses from Web services on the SOAP protocol level. Existing semantic caching schemes for database systems or Web sources cannot be applied directly because there is no semantic knowledge available about the requests to and responses from Web services. Web services are typically described using WSDL (Web Service Description Language) documents. For semantic caching we developed an XML-based declarative language to annotate WSDL documents with information about the caching-relevant semantics of requests and responses. Using this information, our semantic cache answers requests based on the responses of similar previously executed requests. Performance experiments—based on the scenarios of TPC-W and TPC-W Version 2—conducted using our prototype implementation demonstrate the effectiveness of the proposed semantic caching scheme.

1 Introduction

Service-oriented architectures (SOAs) based on Web services are emerging as the dominant application on the Internet. Mission critical services like business-to-business (B2B) or business-to-consumer (B2C) services often require more performance, scalability, and availability than a single server can provide. Server side caching, e.g., [1, 2], and some kind of cluster architecture alleviate some of these problems. A major drawback remains: all clients must still access the Web service directly over the Internet, which is possibly resulting in high latency, high bandwidth consumption, and high server load.

There are many Web services characterized by read-mostly interactions, e.g., B2C and B2B services offering query-like interfaces to access product catalogues. Such services are also used in standard benchmarks like TPC-W [3] and TPC-W Version 2 [4]. Another important category of Web services includes information services like stock quote services, news services, weather services, etc., which typically offer read-only access. There are Web services with different access patterns but since the Web service categories described above are very common and important, this paper focuses on them.

Our generic approach to achieving higher performance and scalability is called *Semantic SOAP Protocol Level Cache* (SSPLC). The performance increase is based on semantic caching of responses from Web services in request/response message exchange patterns on the SOAP [5] protocol level. Clients are not directly accessing the origin service anymore; instead they are accessing instances

* This research is supported by the Advanced Infrastructure Program (AIP) group of SAP and the German National Research Foundation under contract DFG Ke 401/7-2.

of SSPLC. As long as requests can be answered based on cached data, the origin server hosting the Web service is not involved anymore. Therefore, the load at the origin server is reduced, bandwidth consumption is diminished, and latency is reduced. The advantage of a semantic cache is that it reuses the responses to prior requests to answer similar requests, not only the exact same requests. Thus, if request R_1 retrieves all books written by "Rowling" and afterwards a request R_2 retrieves all books written by "Joanne Rowling", a semantic cache reuses the response to R_1 to answer the more selective request R_2.

Our proposed cache can be used like traditional HTTP proxies, i.e., SSPLC instances need not be hosted by service providers themselves, but can easily be run by, e.g., companies and universities, just like HTTP proxies nowadays. However, SSPLC can also be used as reverse-proxy cache or edge server cache. with the additional advantage that server-driven cache consistency techniques are applicable.

Our approach relies on service provider cooperation. All instructions to control the SSPLC are embedded by the provider of a service in SOAP result documents and in the WSDL [6] description of a service. The SOAP results are augmented with information about cache consistency. This is the only modification to a Web service required for the use of SSPLC. The effort necessary to generate these annotations depends on the consistency strategy and the complexity of the application logic and is subject to further investigations. Simple annotations, e.g., TTL values, can be inserted by the SOAP-engine in a post-processing step without modifications of the Web service. More complex annotations demand some coding effort. Additionally, the WSDL document of the service is annotated with information about the caching-relevant semantics of a service. This is done manually using an XML-based declarative language because automatic reasoning about the semantics normally results in a very conservative caching behavior. Writing these annotations is considered to be quite easy for the developers of a Web service as they already have the required knowledge. Altogether, we assume that the additional effort for the provider to make a Web service cachable is clearly outweigh by its benefits.

The remainder of the paper is organized as follows: In Section 2 we present background information and introduce an example Web service used as running example. Several basic design decisions are described in Section 3. A detailed description of SSPLC, the embedded control instructions of service providers, and some sophisticated features of the SSPLC are presented in Section 4. Experimental results follow in Section 5. Section 6 surveys related work and Section 7 presents our conclusions.

2 Background Knowledge and Running Example

2.1 Fundamentals of Semantic Caching

Semantic caching is a client-side caching technique introduced in the mid 90s for DBMSs to exploit the semantic locality of queries [7, 8]. A semantic cache is managed as a collection of *semantic regions* which group together semantically related objects. Regions are composed of *region descriptor* and *region content*.

The descriptor basically contains a predicate (like 'author = "Joanne Rowling" ') describing the region content. The content stores the objects related to a region descriptor. Access history is maintained and cache replacement is performed at the granularity of semantic regions.

Every query sent to a semantic cache is split into two disjoint parts: a *probe query* and a *remainder query*. The probe query extracts the relevant portion of the result already available in the cache while the remainder query is sent to the origin server to fetch the missing, i.e., not cached, part of the result. If the remainder query is empty, the cache does not interact with the origin server. In the context of DBMSs or Web sources, all participating components have been full-fledged DBMSs. Since Web services normally have a more constrained query interface, semantic caching must be adapted to these limitations (see Section 4).

2.2 Running Example

Amazon offers a SOAP-based Web service interface which is very similar to their broadly known HTTP interface. Since Amazon is in fact a "real-world implementation" of the TPC-W benchmark, we use parts of their interface for our example and the TPC-W benchmark scenario as basis for performance experiments conducted using our prototype implementation. Our example service is called *Book Store Light* and is a slim version of Amazon. The relevant operation of this service is a search for books written by certain authors (*author search*). The XML documents used by Amazon are too large to be presented entirely in this paper. We shortened and simplified them to a reasonable degree and removed all namespaces and types from the presented documents for better readability and a more concise presentation.

2.3 The Communication Protocol SOAP

SOAP [5] is an XML-based communication protocol for distributed applications. The root element of a SOAP message is an `Envelope` element containing an optional `Header` element for SOAP extensions and a `Body` element for the payload. SOAP is designed to exchange messages containing structured and typed data and can be used on top of several different transfer protocols like HTTP, SMTP, and FTP. The usage of SOAP over HTTP is the default in the current landscape of Web services. Figure 3 shows an example SOAP response corresponding to the request shown in Figure 1.

2.4 The Description Language WSDL

WSDL (*Web Service Description Language*) [6] is an XML-based language to describe the technical specifications of a Web service, in particular the operations offered by a Web service, the syntax of the input and output documents, and the communication protocol to use for communication with the service. The exact structure of a WSDL document is complex and out of the scope of this paper, but we will give a brief overview of the WSDL standard. At first, a service in WSDL is described on an abstract level and afterwards bound to a specific protocol, network address (normally a URL), and message format. On

the abstract level *port types* are defined. A port type is a set of operations (like author search). Every operation has a number of input and output messages associated defining the order and type of the messages sent to/received from the operation. The messages themselves are assembled from several typed *parts*. The types are defined using XML Schema.

On the non-abstract level, port types are bound to concrete communication protocols and concrete formats of the messages using so-called *bindings*. At last, a service in WSDL is defined as a set of *ports*, i.e., bindings with associated network addresses (normally URLs).

Since SSPLC is currently mainly based on annotations at the abstract level we will focus on this level. Figure 4 shows a fragment of a WSDL document defining the port type of the Book Store Light service (`BookStoreLightPort`) having one operation (`AuthorSearchRequest`). This operation expects an `AuthorSearchRequest` message as input and produces an `AuthorSearchResponse` message as an output document. These messages are defined just above the `portType` element. Messages are composed of several `part` elements. As shown in the figure, the request message has one part of type `AuthorRequest` and the response message has one part of type `ProductInfo`. These types are defined using XML Schema in another fragment of the WSDL document, shown in Figure 2. An element of type `AuthorRequest` has the elements `author` and `levelOfDetail`, both of type `string`, in its content. In our example, `levelOfDetail` can be "heavy" or "lite" and influences the level of detail of the result. Figure 1 shows an example SOAP message requesting the most important information about books written by "Joanne Rowling".

An element of type `ProductInfo` contains the two subelements `TotalResults` and `DetailsArray`. The former is of type `int`, whereas `DetailsArray` is, in short, an array of `Details` elements. `Details` is another type defined inside the WSDL document, having the three subelements `Asin`, `Title`, and `Authors`. The first two subelements are of type `string`, the last one is of type `AuthorArray` which is an array of `string`s representing the authors of the book. For our example, we assume that `Asin` is only present in a result if `levelOfDetail` was "heavy".

3 Basics of the Web Service Cache

We will now discuss our design decisions on several basic caching aspects. These concerns are not the main focus of our work so we used existing solutions as far as possible and adapted existing work where necessary.

3.1 Replacement Policy

Since cache memory is a limited resource, the cache may have to discard some regions to free memory for new regions. After experimenting with some different replacement strategies, we decided to use our own modified version of the 2Q strategy [9], which is a low overhead approximation to LRU-2. Empirically, standard 2Q is a smart choice because of good replacement decisions and low

```
<Envelope encodingStyle="http://...">
 <Body>
  <AuthorSearchRequest>
   <AuthorSearchRequest>
    <author>Joanne Rowling</author>
    <levelOfDetail>lite</levelOfDetail>
   </AuthorSearchRequest>
  </AuthorSearchRequest>
 </Body>
</Envelope>
```

Fig. 1. Example SOAP Request

```
<types><schema>
 <complexType name="AuthorRequest"><all>
  <element name="author" type="string" />
  <element name="levelOfDetail"
           type="string" />
 </all></complexType>
 <complexType name="ProductInfo"><all>
  <element name="TotalResults" type="int" />
  <element name="DetailsArray"
           type="DetailsArray" />
 </all></complexType>
 <complexType name="DetailsArray">
  <complexContent>
   <restriction base="Array">
    <attribute ref="arrayType"
               arrayType="Details[]" />
   </restriction>
 </complexContent></complexType>
 <complexType name="Details"><all>
   <element name="Asin" type="string" />
   <element name="Title" type="string" />
   <element name="Authors"
            type="AuthorArray" />
 </all></complexType>
 <complexType name="AuthorArray">
  <complexContent>
   <restriction base="Array">
    <attribute ref="arrayType"
               arrayType="string[]" />
   </restriction>
 </complexContent></complexType>
</schema></types>
```

Fig. 2. Type Definitions

```
<Envelope encodingStyle="http://...">
 <Body>
  <AuthorSearchRequestResponse>
   <return>
    <TotalResults>200</TotalResults>
    <DetailsArray arrayType="Details[200]">
     <Details>
      <Title>
       Harry Potter and the Sorcerer's Stone
      </Title>
      <Authors arrayType="string[2]">
       <Author>Joanne K. Rowling</Author>
       <Author>Mary GrandPré</Author>
      </Authors>
     </Details>
     <!-- 199 more Details elements -->
    </DetailsArray>
   </return>
  </AuthorSearchRequestResponse>
 </Body>
</Envelope>
```

Fig. 3. Example SOAP Response

```
<message name="AuthorSearchRequest">
  <part name="AuthorSearchRequest"
        type="AuthorRequest" />
</message>
<message name="AuthorSearchResponse">
  <part name="return" type="ProductInfo" />
</message>
<portType name="BookStoreLightPort">
  <operation name="AuthorSearchRequest">
    <input message="AuthorSearchRequest" />
    <output message="AuthorSearchResponse" />
  </operation>
</portType>
```

Fig. 4. Messages and Port Types

```
<CacheControlHeader>
 <CacheConsistency>
  <TTL>P0Y0M0DT12H0M0S</TTL>
 </CacheConsistency>
</CacheControlHeader>
```

Fig. 5. Cache Consistency Information

```
<binding name="BSLBinding" type="BookStoreLightPort">
 <binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
 <operation name="AuthorSearchRequest">
   <operation soapAction="BookStoreLight" />
   <!-- ...mappings of input and output message... -->
   <OperationCacheControl>
    <fragmentationXPath>
     /Envelope/Body/AuthorSearchRequestResponse/return/DetailsArray/Details
    </fragmentationXPath>
    <reassemblingXQuery> <!CDATA[
      let $details := ##RESULT_FRAGMENTS##
      return
      <Envelope encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
       <Body>
        <AuthorSearchRequestResponse>
         <return type="ProductInfo">
          <TotalResults type="int">##COUNT_RESULT_FRAGMENTS##</TotalResults>
          <DetailsArray arrayType="Details[##COUNT_RESULT_FRAGMENTS##]" type="Array">
           {$details}
          </DetailsArray>
         </return>
        </AuthorSearchRequestResponse>
       </Body>
      </Envelope> ]]>
    </reassemblingXQuery>
   </OperationCacheControl>
 </operation>
</binding>
```

Fig. 6. Annotation of the AuthorSearchRequest Operation

CPU overhead, but this algorithm is designed to handle objects of uniform size. As semantic regions can be of different size, we had to modify the standard 2Q strategy by introducing a simple but efficient *cost-to-size ratio*. More details on our modifications of 2Q can be found in the extended version of this paper [10].

3.2 Distribution Control/Cache Consistency

SSPLC gives providers exclusive control over distribution and cache consistency using a SOAP header extension. Since cache consistency mechanisms are not the focus of this work, we assume service-specific TTL in the following discussion. If a provider allows caching, it must explicitly state some cache consistency information. For example, the `CacheControlHeader` element shown in Figure 5 allows caching and states that the response is *fresh* for at least the given duration (12 hours). After this duration, the cached version of the response must be removed from the cache.

3.3 Physical Storage of Semantic Regions

Using a cache requires a large amount of memory to be able to serve lots of clients based on a reasonably large number of semantic regions. Since disks are considerably larger and cheaper than main memory, it is obviously a good idea to use them for the storage of semantic regions. Since it is orthogonal to the issues discussed in this paper whether the cache is based on main memory, disk, or both, we assume for the rest of the paper that the cache is only based on main memory. Our prototype system is main memory-based as well.

4 Semantic Caching in SSPLC

Basically, semantic caching in SSPLC is done by annotating WSDL documents with information about the caching-relevant semantics of services using the language presented in the next section. This information is used for mapping SOAP requests to predicates, for fragmenting responses, and for reassembling responses. Thus, adapted semantic caching algorithms can be applied.

4.1 WSDL Annotations

Our language is designed both to cover common capabilities of existing Web service interfaces and to preserve efficient solvability of the *query containment problem*, which is intrinsic to semantic caching. The annotation of WSDL documents is done using XML Schema annotation elements and WSDL extensibility elements. Thus, compatibility to the original WSDL document is preserved, because applications which cannot handle the annotations ignore them.

Fragmentation and Reassembling Since Web services deliver monolithic XML documents rather than tuple-oriented responses, SSPLC needs some information about how to fragment such documents to obtain fine-granular response units comparable to tuples in database caching. These units are called *fragments*.

We use an XPath-expression to specify the fragmentation. Additionally, SSPLC needs further instructions regarding the generation of a complete response document based on fragments of prior requests. This information is specified using the XQuery language. Both the XPath-expression and the XQuery, are provided using an additional element (`OperationCacheControl`) inside the `binding` element of the WSDL document of a service because it depends on the actual coding of the messages.

Figure 6 gives an example for our Book Store Light. The marked region depicts the annotated information for the SSPLC while the rest of the document constitutes a standard SOAP binding. Referring to our book store example, we are interested in the individual books, i.e., `Details` elements, contained in a response document of our example service. The XPath-expression inside the `fragmentationXPath` element in Figure 6 fragments a response document accordingly. The XQuery to reassemble a response is shown in the figure inside the `reassemblingXQuery` element. The macros `##COUNT_RESULT_FRAGMENTS##` and `##RESULT_FRAGMENTS##` are expanded by the SSPLC before evaluating the XQuery and represent exactly the fragments (respectively their number) which should be reassembled to a complete response document. Since an introduction to XQuery lies outside the scope of this paper, we will not explain the XQuery shown in the figure. It should be obvious that the result of the XQuery is a SOAP response like the one shown in Figure 3.

Predicate Mapping We need predicates to describe the fragments stored in a region. Thus, we need some information about the semantics of requests. Moreover, we want to be able to filter semantic regions, e.g., if we are looking for all books written by "Joanne Rowling" in a region storing all books written by "Rowling". Therefore, we need to know how to access the individual "attributes" (elements) of a tuple (fragment). This information is annotated to the type definitions of requests in WSDL documents.

We will explain the annotations using our Book Store Light example. The original type definition of `AuthorRequest`, which is the request type of our service, is shown in Figure 2. Currently, we assume that if there are several parameters defined in a request, i.e., `levelOfDetail` and `author`, they are combined by an AND operator. Thus, the request shown in Figure 1 means that we are looking for all books written by "Joanne Rowling" and we are only interested in the most important facts of the books. Additionally, we assume that if there are several elements inside an array, the elements are logically ANDed together, too. This is also true for responses (see the `Author` elements inside the `Authors` element shown in Figure 3). The annotated version of the `AuthorRequest` type is shown in Figure 7.

We annotate every parameter of the request using one or more `CacheControl` elements. It is necessary to specify some context information because a parameter can be used for several operations having different semantics. Also, if another binding is used, the coding of the parameter might be different, requiring some modifications inside the `CacheControl` element. Thus, the context information given by the attributes of `CacheControl` defines when to use the information

```
<complexType name="AuthorRequest"><all>
  <element name="author" type="string">
   <annotation><appinfo>
     <CacheControl context="AuthorSearchRequest"
                   bindingContext="BSLBinding">
      <StringParameter>
       <required>true</required>
       <fragmentXPath>
        Authors/Author/text()
       </fragmentXPath>
       <implicitOperator>contains_wwo</implicitOperator>
       <caseSensitive>false</caseSensitive>
       <operators>
        <and> </and><and>,</and>
       </operators>
      </StringParameter>
     </CacheControl>
    </appinfo></annotation></element>
  <element name="levelOfDetail" type="string">
   <annotation><appinfo>
     <CacheControl context="AuthorSearchRequest"
                   bindingContext="BSLBinding">
      <StringParameter>
       <required>true</required>
       <implicitOperator>equals</implicitOperator>
       <caseSensitive>true</caseSensitive>
      </StringParameter>
     </CacheControl>
    </appinfo></annotation></element>
 </all></complexType>
```

Fig. 7. Annotated WSDL Type Definition

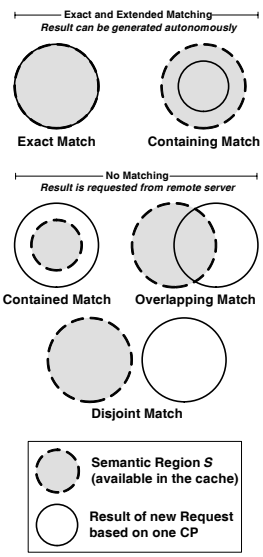

Fig. 8. Match Types

inside the `CacheControl` element. A `StringParameter` element defines that the parameter is of type string. The content of this element gives more detailed information about how to handle this string parameter. We also defined elements for other parameter types, e.g., an `IntegerParameter` element. Each of these elements contains further information (e.g., operators) depending on the parameter type.

Looking at the example in Figure 7, we observe that the author parameter is mandatory (`required` element). If a parameter is optional, a default value of the parameter that is used in case of absence of the parameter in a request must be specified using a `default` element (not available in the example document). The `fragmentXPath` element specifies how to extract the information from result fragments that correspond to this parameter (compare Figure 3). For example, if we ask for books written by an author, the `fragmentXPath` can be used to find the authors in the result fragments. If, as in our example, an XPath is specified, the cache can inspect the fragments to look up the actual author(s) of a book. This information can be used to filter all fragments contained in a semantic region. If there is no XPath specified, the cache is not able to do such filtering because it is constrained to the information obtained from the request.

The element `implicitOperator` defines the operator of the parameter. Currently, we support the following operators (for appropriate parameter types): $>$, $\geq$, $<$, $\leq$, $=$ (or *equals*), *contains*, *contains_wwo*, *starts_with*, and *ends_with*. In our example, the operator is *contains_wwo* which is a contains operator that looks for "whole word only" occurrences of the given pattern in a string, i.e., "Joanne Rowling" does not contain_wwo "Rowl", but contains_wwo "Rowling".

The comparison of strings is case insensitive as defined by the `caseSensitive` element.

Additionally, we support the logical operators AND and OR to support complex predicates. We also support parentheses for precedence control. Currently, we are not supporting the $\neg$ operator (logical NOT operator) because there are virtually no Web services offering this operator and we are interested in keeping the query containment problem efficiently solvable. The `operators` element in Figure 7 defines two AND operators for the author parameter: a space character and a comma.

The second parameter is `levelOfDetail`. This is also a mandatory string parameter. The implicit operator is a case sensitive "equals". There is no `fragmentXPath` defined because in the response document of our Web service no explicit information about whether it is a "heavy" or a "lite" result is contained. As this information is stored as part of the region predicate, this information is not lost.

Using these annotations SSPLC can figure out the semantics of a request and is able to extract relevant elements from fragments. Also, it is able to generate a predicate from a request. The request shown in Figure 1 is mapped to the following predicate:

$$\begin{gathered}\text{author contains\_wwo\_case\_insensitive “Joanne”} \wedge \\ \text{author contains\_wwo\_case\_insensitive “Rowling”} \wedge \\ \text{levelOfDetail equals\_case\_sensitive “lite”}\end{gathered}$$

4.2 Matching and Control Flow

Using our annotations we are now able to understand the caching-relevant semantics of requests and responses. We will now describe how this information is used for caching. First of all, a SOAP request R is mapped to a predicate P as described above. Although the Book Store Light does not offer a logical OR operator for the author parameter, we will use the following predicate P (operator names are shortened) for demonstration purposes throughout this section:

$$\begin{gathered}(\text{author contains “Rowling”} \vee \text{author contains “GrandPré”}) \wedge \\ \text{levelOfDetail} = \text{“lite”}\end{gathered}$$

After the mapping, P is transformed into *disjunctive normal form* (DNF) and split into *conjunctive predicates* (CPs), i.e., predicates only containing simple predicates connected by logical AND operators. If there is no logical OR in a request, P is processed as is. The transformation of our example predicate P results in:

CP_1: author contains "Rowling" $\wedge$ levelOfDetail = "lite"
CP_2: author contains "GrandPré" $\wedge$ levelOfDetail = "lite"

All CPs are processed in parallel. First, match types of a CP with all semantic regions are determined, i.e., the correlation between every semantic region S and the result of CP is determined. There are five different match types as shown in Figure 8. The best match type for a CP and a semantic region S is, of course, the exact match. The next best match type is a containing match because we

only have to filter S by eliminating all fragments fulfilling the region predicate but not CP to get the fragments for the response. The other three match types require server interaction because we do not have all fragments cached to answer the request. Since most Web services do not have adequate interfaces to be able to process complicated remainder requests, we handle all three match types as disjoint match. Thus, we are sending a request generated from the CP to the Web service even though there already might be some relevant fragments available in the cache. Even if a Web service can process complicated remainder requests, processing of such complex requests is likely to be costly. As one of the goals of SSPLC is to reduce processing demands of the central servers, usage of complex remainder requests could be counterproductive. The response of the Web service is fragmented and afterwards stored in the cache. If there are already regions in the cache that are a subset of the response (i.e., in the case of a contained match), these semantic regions are replaced with the new (larger) semantic region. In all other cases, the fragmented response is inserted as a new semantic region using CP as the region predicate. After all CPs have been processed, SSPLC calculates the result of P as the union of the results of all CPs. By default, duplicates are eliminated, i.e., SSPLC implements the very common set semantics. Alternatively, SSPLC calculates the result without duplicate elimination. This behavior is controlled by an optional `distinct` element inside the `OperationCacheControl` element (not shown in the example document). Fragments are considered equal if their contents are equal or if keys are defined, their keys are equal. Keys can be defined via a `key` element inside the `OperationCacheControl` element using the standard XML Schema syntax for keys. Usage of keys considerably speeds up duplicate elimination. We do not further investigate keys in the scope of this paper. The result of P is (conceptually) written to an XML document D. After that, the `reassemblingXQuery` is evaluated with the macro `##RESULT_FRAGMENTS##` expanded to D and the macro `##COUNT_RESULT_FRAGMENTS##` expanded to $|D|$. Finally, the response is sent back to the client.

4.3 Sorting and Generalization

Since the order of elements can be important in XML documents, SSPLC is aware of it. XML documents are inherently ordered by the sequence of the elements (*document order*). As long as the document order generated by a Web service offers no real added value (e.g., lexicographical order by title), it does not matter in which order the fragments emerge in the response. Also, as long as we are using fragments of only one semantic region (filtered or not), order is abided and we can generate correctly ordered results as in the Book Store Light example.

If a Web service orders fragments using some information available in the response, there are two possibilities to establish the same order even if we are merging fragments of several semantic regions to generate the response. First, if the order is fixed, i.e., always the same, the `reassemblingXQuery` can be modified to do the sorting using the *order by* clause of XQuery. Second, if the order depends on a request parameter, we can annotate this parameter using

a `SortParameter` element. This element contains a mapping from the service's sorting facilities to order by clauses of XQuery. For example, if a Web service has a parameter `sort` and the value "`+title`" means "sort by title", a mapping to XQuery could look like "`order by $fragment/Title`". The appropriate order by clause is inserted into `reassemblingXQuery` before evaluation. The value of a sorting parameter is stored in the region descriptor because it is relevant for determining the match types.

Another enhancement of our semantic caching scheme is the usage of generalization for better decisions on the query containment/predicate subsumption problem. Our SSPLC supports two different types of generalization. First, tree-structured containment relations for values of parameters can be defined. For example, if there is a parameter defining whether we are interested in paperback, hardcover, or both, we are able to annotate this parameter to point out that "hardcover $\subseteq$ both" and "paperback $\subseteq$ both". This information is used during match type computation and for filtering of semantic regions. The second type of generalization can be seen in our example. There is a parameter `levelOfDetail` that influences the level of detail of the response. Since "heavy" fragments simply contain some extra elements, it is possible to define an XQuery filter to transform "heavy fragments" to "lite fragments" by removing the surplus elements like the `Asin` elements in our example. This information is also used during match type computation and region filtering.

5 Performance Evaluation

We implemented a prototype of SSPLC for the service platform ServiceGlobe [11] using Java and conducted several performance experiments based on the scenarios of TPC-W [3] and TPC-W Version 2 [4].

5.1 Benchmark Scenario 1 (TPC-W)

The first scenario is related to the online bookstore scenario of the TPC Web commerce benchmark (TPC-W). Because TPC-W does not aim at SOAP Web services and semantic caching, but instead at traditional Web servers and back-end servers, major modifications to TPC-W (system architecture as well as data generation) are necessary to adjust the benchmark to the context of our SSPLC in a reasonable way. Thus, we decided to model our benchmark scenario on the SOAP interface of Amazon, just as the scenario of TPC-W is modeled on the HTTP interface of Amazon. We chose to use Amazon's author search request for our benchmarks because this search functionality is also addressed in TPC-W.

Experimental Setup Due to space restrictions, we only present a survey of the experimental setup of benchmark scenario 1. A detailed description can be found in the extended version of this paper [10].

To show the effectiveness of our semantic cache, we implemented a simulation service rather than using Amazon directly because Amazon delivers its results page-wise (i.e., 10 books per SOAP response), which is an unusual behavior for

Web services. The requests and responses of our simulation service are identical to those of the Amazon service despite the fact that our service delivers all results to a request in one response. For that purpose, we materialized some of the data of Amazon to be able to work with real data. Since our simulation service delivers these materialized results extremely fast, we are delaying results to simulate processing time of a Web service. We conducted some experiments to assure that SSPLC is able to deliver its results as fast or faster on average than the origin Web service. Since these results depend heavily on the performance of the origin server and of the machine running SSPLC, we do not present quantitative results.

Our benchmark scenario is based on several top-300 bestseller lists (top selling science books, top selling sports books, ...) of Amazon. We used these different bestseller lists to generate different traces as described below and we always present the average of all performance experiments conducted using these different traces. If an author's book is present on the bestseller list, people will be interested in other books published by the same author, too. Thus, an author search request is more likely for authors whose books are ranked high on the bestseller list. Since studies [12] have found that the request characteristics of many Internet applications are adequately modeled through a Zipf-like distribution, we use such a distribution (with parameter theta (θ) set to 0.75) on the top-300 bestseller lists to select books. Using the names of the authors of a book, we generate a request for our simulation service. We randomly choose which names (surnames, first names) are used for the request. Every request contains at least one surname of an author. This is done to challenge semantic caching. We generated traces of 2000 requests each for the performance experiments. Additionally, we conducted some experiments using traces of 10000 requests showing similar results.

Some of the requests produce very large response documents containing up to 32000 fragments. Since the size of such documents is about 40 MB, it is very likely that Web services do not generate such large responses. Rather, they generate a fault response informing the caller that there are too many results and that the request has to be refined. Thus, our simulation service sends fault messages for results containing more than 2000 fragments. SSPLC caches these fault messages because they are marked cachable in the SOAP header.

We conducted several performance experiments varying different parameters and we present the results in this section. For the experiments in this section, the TTL of responses was set to 30 minutes, if not explicitly stated differently. The maximum size for responses to be cached was set to about 1000 fragments (1.2 MB). Larger responses were fetched from the remote Web service and forwarded to the client without caching. We conducted the experiments using three different cache sizes: small (10% of the data volume of the unique-trace[3]), standard (20%), and large (30%). The cache was warmed up by running every trace twice and measuring the second one, although there are only minor differences between the two runs.

[3] The term *unique-trace* refers to a trace where all duplicates are removed.

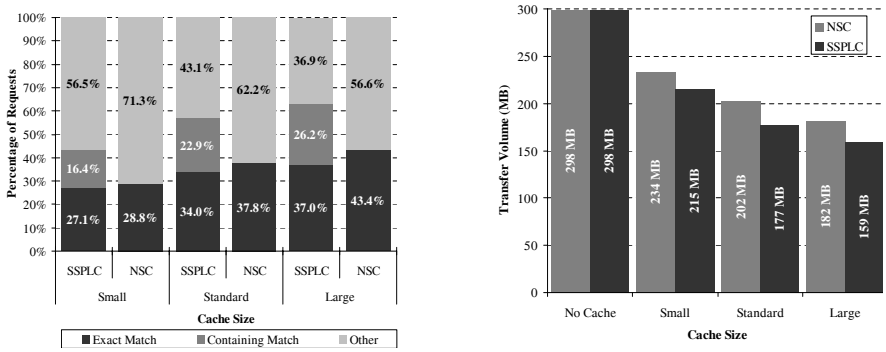

Fig. 9. Match Distribution (Left) and Transfer Volume (Right) Varying Cache Size

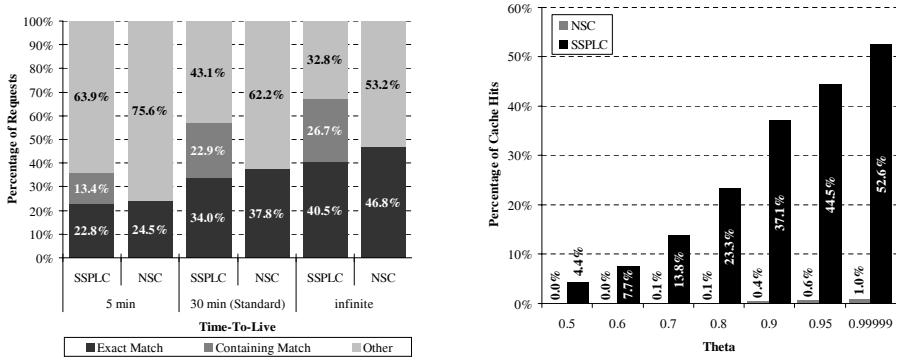

Fig. 10. Match Distribution Varying TTL

Fig. 11. Cache Hits Varying Theta

Experimental Results Due to space restrictions, we only present the core results of benchmark scenario 1. Detailed experimental results are presented in the extended version of this paper [10].

The main goal of the SSPLC is to improve scalability of Web services. Figure 9 shows[4] that already the smallest semantic cache is able to answer 43.5% (exact matches + containing matches) of all requests using data stored in the cache, reducing processing demands on the central servers significantly. A traditional (non-semantic) cache (NSC) achieves much smaller hit rates (28.8%). The bigger the caches are, the better the hit rates become even though the increase rate is not linear with the cache size increment. This is due to the fact that already the standard cache size is large enough to cache most of the hot spot responses. The only advantage of a larger cache is that it is able to additionally store some of the less frequently requested responses. SSPLC benefits more from a larger cache than NSC because SSPLC can exploit the semantics of the requests.

[4] Please note that the sum of exact matches, containing matches, and other matches is not always exactly 100% due to rounding errors.

Figure 9 demonstrates the reduction of bandwidth consumption. Running the trace without cache results in the transfer of 298 MB across the network. The smallest semantic cache reduces the transfer volume by approximately 28%, the standard semantic cache by approximately 41%. The large semantic cache reduces the transfer volume even more, but the difference is not linear with the cache size increment due again to the reasons above. The increased hit rate of SSPLC does not 1:1 translate into equally large bandwidth savings in this scenario. For example, the hit rate of SSPLC is about 19% higher compared to NSC using the standard cache size. This hit rate increment translates into about 14% bandwidth savings. The correlation between hit rate increment and bandwidth savings depends on the size of the cached semantic regions and the traces. Nevertheless, the transfer volume of NSC is on average more than 12% larger than that of SSPLC.

Figure 10 shows results for varying time-to-live periods. Of course, the longer the TTL period is, the more effective the caches are. Depending on the TTL, SSPLC performs about 43% to 50% better than NSC.

5.2 Benchmark Scenario 2 (TPC-W 2)

The Transaction Processing Performance Council published a first draft of TPC-W Version 2 (TPC-W 2) for public review. This new version of TPC-W is aiming at Web services. Thus, we decided to conduct some additional performance experiments based on TPC-W 2. Due to incomplete specifications and time constraints, we did not implement the full benchmark. Rather, we chose the "product detail Web service interaction" of TPC-W 2 to conduct our experiments. The data was generated conforming to the rules of TPC-W Version 2, i.e., 100000 books were generated and stored in the DBMS. We configured our remote business emulator (RBE) to run 8 emulated businesses (EB) concurrently. The TTL was set to 5 minutes[5] and a total of 3000 requests were sent to the SSPLC. The cache was able to store about 2500 books. Every request asked for detailed information about a randomly chosen number (1 to 10) of books. According to the TPC-W 2 specifications, the books should be selected using a given non-uniform random distribution, but this distribution generates values which are distributed too uniformly for any cache. Therefore, we used a Zipf-like distribution to select the books.

If a client requests product details for, e.g., book 2 and book 8, SSPLC translates the request to the predicate "book = 2 $\vee$ book = 8". Thus, SSPLC splits up the request into two CPs, as described above, and generates a request for every single book if not available in the cache. For this reason, there are only exact matches and disjoint matches in this scenario. If not all books of a request are available in the cache, the SSPLC rates the request as exact match and disjoint match according to the ratio of books available in the cache to books not available in the cache. For example, if a client requests details about eight books and six books are available in the cache, the request is rated as 0.75 exact match and 0.25 disjoint match.

[5] Every benchmark run lasted for about 20 minutes.

Figure 11 shows the exact matches for the benchmark varying theta of the Zipf-like distribution. A non-semantic cache (NSC) is virtually useless in this scenario because the cache hits are less than 1%, even if $\theta = 0.99999$. This is because NSC can only answer requests from the cache if two requests are exactly the same, i.e., the number of product details requested must be the same, the books must be the same, and the order of the books must be the same. SSPLC works very well for sufficient large θ, even though the cache size is small (about 5% of the data volume available at the origin server) and the TTL is short. For a realistic θ, i.e., greater or equal to 0.8, the SSPLC is able to answer more than 23% of the requests.

6 Related Work

Caching in the context of Web services has been addressed, e.g., by the usage scenarios S032 and S037 of the World Wide Web Consortium [13]. The proposed approaches are either described very abstractly, or are limited to a more or less straightforward store-and-resend of SOAP responses. Our approach differs in that it takes advantage of the fact that query-style requests can be cached more efficiently using semantic caching. Thus, this paper proposes an alternative solution which is more flexible and powerful.

A solution for a similar but simpler problem in the area of Web sources and respectively Web databases, was presented by [8]. They focus on wrapper[6] level caching. Therefore, they are able to take advantage of the semantics of the declarative query language SQL, i.e., they automatically deduce region predicates from SQL queries. In the area of Web services, no such standardized declarative language exists. Due to our declarative language for the annotation of WSDL documents with information about caching-relevant semantics, we are able to apply semantic caching to Web services in, e.g., B2B and B2C scenarios. Additionally, we investigate sorting and generalization issues. Thus, our solution is more comprehensive and more flexible. The basic techniques of both SSPLC and [8] are based on prior work on semantic caching, e.g., [7].

A different usage of caching for Web services is presented in [14]. They use caching techniques for reliable access to Web services from, e.g., PDAs or similar unreliably connected mobile devices. The authors use one representative service to demonstrate the benefits of a Web service cache and expose a number of issues in caching Web services. They do not present a generic solution, but they do conclude that extensions to WSDL are needed to support cache managers. We think that the language presented in this paper constitutes a good base for such extensions.

7 Conclusions and Future Work

We presented the semantic cache SSPLC that is suitable for caching responses from Web services on the SOAP protocol level. We introduced an XML-based

[6] Wrappers are used to extract data from Web sources.

declarative language to annotate WSDL documents with information about the semantics of services. We demonstrated the validity of our proposed caching scheme by performing a set of experiments.

We plan to investigate some ideas on how SSPLC can be further improved. The declarative language can be extended to integrate additional semantic knowledge like *fragment inclusion dependencies* [8] to transform as many overlapping or contained matches as possible into exact or containing matches. Furthermore, we intend to improve our caching scheme by taking advantage of richer interfaces of services.

References

1. Yagoub, K., Florescu, D., Issarny, V., Valduriez, P.: Caching Strategies for Data-Intensive Web Sites. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), Cairo, Egypt (2000) 188–199
2. Larson, P., Goldstein, J., Zhou, J.: MTCache: Transparent Mid-Tier Database Caching in SQL Server. In: Proceedings of ICDE, Boston, MA, USA (2004) 177–189
3. Transaction Processing Performance Council: TPC Benchmark W Version 1.8 (2002) `http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf`.
4. Transaction Processing Performance Council: TPC Benchmark W Version 2.0r (2003) `http://www.tpc.org/tpcw/spec/TPCWV2.pdf`.
5. Box, D., et al.: Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP11` (2000)
6. Christensen, E., et al.: Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315` (2001)
7. Dar, S., Franklin, M.J., Jónsson, B.T., Srivastava, D., Tan, M.: Semantic Data Caching and Replacement. In: Proceedings of VLDB, Mumbai (Bombay), India (1996) 330–341
8. Lee, D., Chu, W.W.: Towards Intelligent Semantic Caching for Web Sources. Journal of Intelligent Information Systems (JIIS) **17** (2001) 23–45
9. Johnson, T., Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In: Proceedings of VLDB, Santiago de Chile, Chile (1994) 439–450
10. Seltzsam, S., Holzhauser, R., Kemper, A.: Semantic Caching for Web Services – Extended Version. `http://www-db.in.tum.de/research/publications/techreports/SemCachingExtended.pdf` (2005)
11. Keidl, M., Seltzsam, S., Kemper, A.: Reliable Web Service Execution and Deployment in Dynamic Environments. In: Proceedings of the International Workshop on Technologies for E-Services (TES). Volume 2819 of Lecture Notes in Computer Science (LNCS)., Berlin, Germany (2003) 104–118
12. Adamic, L., Huberman, B.: Zipf's Law and the Internet. Glottometrics **3** (2002) 143–150
13. He, H., Haas, H., Orchard, D.: Web Services Architecture Usage Scenarios. `http://www.w3.org/TR/ws-arch-scenarios` (2004)
14. Terry, D.B., Ramasubramanian, V.: Caching XML Web Services for Mobility. ACM Queue **1** (2003) 70–78