A Browser-based Framework for Data Cache in Web-Delivered Service Composition

Jiyu Huang Xuanzhe Liu Qi Zhao Jianzhu Ma Gang Huang*

Key Laboratory of High Confidence Software Technologies, Ministry of Education,
School of Electronics Engineering and Computer Science, Peking University,
Beijing, China

huangjy07@sei.pku.edu.cn liuxzh@sei.pku.edu.cn zhaoqi06@sei.pku.edu.cn majzh@sei.pku.edu.cn huanggang@sei.pku.edu.cn

***Abstract*—Service Oriented Computing (SOC) paradigm aims at building applications by composing available services over the Internet. Recently, one important application direction of SOC is to compose the Web-delivered services (SOAP and RESTful Web services, RSS/Atom feeds, etc) for new Web applications (e.g., the Web mashups). Due to the fact that a number of Web-delivered services fetch data from remote servers, Web applications might require browser cache in order to cooperate with the performance issues (e.g. traffic and latency). However, cache strategies are usually predefined by service providers and brought into effect by browsers. Such pattern of cache makes developers, who are exactly responsible of composing services for Web applications, hardly customize their own cache strategies according to their own application contexts. We argue that, these limited cache strategies may cause unnecessary communications and reduce user experience.This paper proposes a browser-side cache framework for Web-delivered services composition. The main efforts of this paper are as follows. Firstly, our framework allows developers to customize their own cache strategies, such as expiration time, cache granularity and so forth. Secondly, we propose an adaptive technique to adjust cache strategies dynamically, in order to improve cache performance. Finally, we evaluate our framework by conducting experimental studies with the real-world Web-delivered services.**

Keywords-Web-delivered service; composite application; cache

I. INTRODUCTION

Service Oriented Computing paradigm aims at composing available existing services for new applications. Currently, there have been a large number of services delivered via Web, such as SOAP Web services, RESTful web services or RSS/Atom feeds[1]. And a lot more could be easily discovered via search engines [1]. With the powerful support of Web technologies, e.g., AJAX (Asynchronous JavaScript and XML) and RIA (Rich Internet Applications), these Web-delivered services can be retrieved, accessed and invoked within the users' web browser. More and more developers are capable of composing the Web-delivered services to form new Web applications. Such Web applications, or the so-called Web-delivered composite applications in this paper, have been flourishing over the Web and become a very important application direction and research topics in SOC [2].

Under our investigation, we find that a large number of Web-delivered services provide functionality for fetching data from remote servers (e.g., database). Previous exploratory study [3] has revealed that over 60% of web services provide data lookup functions. As for the state of the art, 30 out of the total of 49 eBay services are for fetching data[2], and similar distributions exist in other popular commercial service providers as well.

Due to the vast demand for "fetching data", Web applications should employ browser cache in order to improve the traffic and latency issues. To this end, cache strategies (such as expiration time indicated in HTTP Header or Meta Tag[3]) are customized by service providers and brought into effect by browsers[4]. This cache "pattern" applies to not only the traditional Web applications, but also to most composite applications as well.

However, when composing services, developers may find that the cache strategy of a service is improper or even undesirable for their context. And this becomes more possible when more services are involved in a composite application. Eventually, the absence of customized cache strategies might lead to unnecessary communications and user experience issues. For example, consider three different developers (A, B, C) have leveraged the Google weather service to build composite applications. The service is assigned a "no-cache" strategy by the provider, as Google might consider the result is very time sensitive. While this cache strategy works well with developer A in a real-time context, it might be inappropriate for the others. Suppose developer B employs the result to feed other services which care less about the weather accuracy. And suppose developer C only needs to extract the temperature for today from the response, which varies little and should be cached separately and longer. In other words, B desires a cache strategy which refreshes the entire response less frequently, while the developer C calls for a strategy that would cache on basis of fine-grained structures within a response.

* Corresponding Author: huanggang@sei.pku.edu.cn.

[1] ProgrammableWeb now has already hosted 1254 (Until April 12nd, 2009) of API entries

[2] eBay API, http://developer.ebay.com/products/overview/

[3] HTTP/1.1, http://www.w3.org/Protocols/rfc2616/rfc2616.html

[4] For example, Twitter.com sets the expiration date to "Tue, 31 Mar 1981 05:00:00 GMT" in http://twitter.com/statuses/show/id, which indicates the cached data is always invalid.

Nonetheless, according to Google's original cache strategy, users might not leverage any cache data in all above cases. Therefore, both of the clients and servers are associated with unnecessary communications.

A scrutiny of the example urges the necessity for a new cache pattern—to enable developers to customize their own cache strategies. Within the cache strategy, they could set up expiration time, cache granularity and so forth. On the other hand, most current Web applications cannot meet such need, since browsers always execute cache strategies specified in HTTP Head or Meta Tag from service providers (e.g. IE[5], Firefox[6]). Besides, popular Mashup tools have done little work in providing cache strategies for developers [4] (e.g. Yahoo Pipes[7] and so on). Consequently, two issues are raised. First and foremost, both clients and servers are associated with unnecessary communications. For example, browsers might make redundant invocations according to inappropriate cache strategies. Secondly, user experience is affected, such as making calls to unreachable Web-delivered services according to providers' cache strategies.

Challenges to implement the new cache pattern are threefold. Firstly, in order to incorporate more flexibility for developers when configuring strategies, it might be appropriate to cache on basis of fine-grained structures within responses. Reusing duplicate data among these data structures could be tough work. For example, different weather services might assign different names for the data structure of "weather forecast". This makes it ambiguous to reuse this data when sending requests to either of the services. Secondly, since cache would be done on these fine-grained structures, it is vague to determine a hit on such structures merely by the request URL. Finally, if the expiration time of the strategy could adapt to user behaviors (including hit-ratio and access-frequency) we could achieve a better performance. Though this seems natural, to find a sound adaptive model is challenging. For a better illustration, suppose two examples and their potential solutions. Firstly consider a case about hit-ratio, if a user sends 1000 different requests to the same web service. This leads to 1000 times of cache miss and unnecessary memory occupation. To solve this, our cache framework might set a threshold value of hit ratio, while low hit ratio is associated with disabling the cache and vice versa. This threshold value relies heavily on the adaptive model and must be tested with empirical experiments. As for the influence of the access frequency, what happens if users access some data too frequently? Obviously, hit-ratio would then change dramatically and lead to unstable performance. So we have to figure out the covert relationship between the two and make the expiration time adapt to them.

Depending on the location of the composing resides, caching could be achieved either on server-side or the browser-side. Within this paper, we mainly target at the latter case, and we treat the realization of the server-side runtime as our future work. We propose a browser side cache framework to enable the developers to customize their own cache strategies. The framework is integrated in our previous work—a browser-based composition tool called iMashup [10]. iMashup is a client side tool to compose Web-delivered services. It consists of a component model for separating service business and user interface, and a composition environment to compose Web-delivered services on-the-fly. Within iMashup, our cache framework assists developers to bind data models with Web-delivered services, and to make their own cache strategies for data models. Subsequently, the framework takes over clients' requests, and applies the cache strategies to data models. The evaluation substantiates that our framework improves the traffic and latency issues, and enhances user experience.

This paper is organized as follows: section 2 discusses a motivating example, which is later referred to in the approach overview of section 3. Section 4 is the implementation of our framework and section 5 is for an evaluation study. Final conclusion and future work are discussed in Section 6 and 7.

II. MOTIVATING EXAMPLE

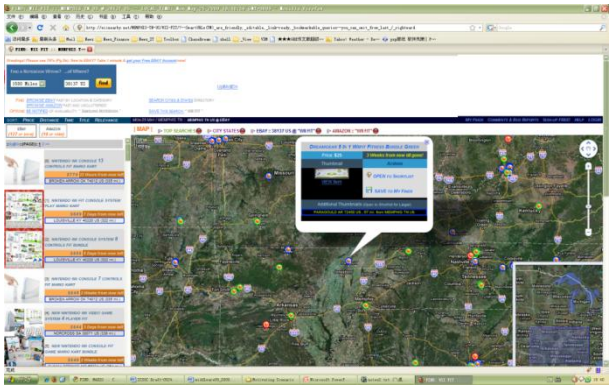

Figure 1. Notebook Seeker

To begin with, we might look at a "Notebook Seeker" Web Application. This application would check out all available notebook Providers' retail addresses, shipment dates, and prices. It leverages the following three Web APIs: Amazon eCommerce, eBay and Google Maps.

"Notebook Seeker" processes data in the following sequences. Firstly, users input a range of location (e.g. 1000 Miles) around his/her home. Secondly, the scripts on the web page would send inquiries to both the eBay and Amazon APIs, each of which responses a list of notebook providers with addresses and prices. Thirdly, based on the location information from the list, all providers are displayed on the Google Map on the webpage. Finally, when users click on one marker on the Google Map, an information box is popped up with the details of the provider retrieved from the corresponding services. The responses from the services have the following structures. When requesting the list of notebook providers (denote this service as the "list service"), the user receives a response result compromising an array of desirable providers. When requesting the detail information (denote this service as the "detail service") as clicking on a marker, the user receives

[5] Cache in Internet Explorer, http://support.microsoft.com/kb/263070/en-us

[6] Cache in Mozilla Firefox, http://www.mozilla.org/projects/netlib/http/http-caching-faq.html

[7] Yahoo Pipes, http://pipes.yahoo.com/

an elaborated description compositing name, address and price for the provider.

The service providers of eBay and Amazon determine the cache strategies in the scenario. Both of the strategies require that the refresh is executed on every request, and the cache is executed on basis of the entire responses. However, when implementing the application, the developer needs such a cache strategy: refresh is not necessary within one minute from last response and cache could be done on basis of notebook providers.

Such scenario would induce concerns as follows. Consider if the user has a low speed of connection to the Internet, he/she has to communicate with the servers at every request since service providers require him/her to do so. Moreover, if the user is only concerned with the prices and addresses of the notebook providers, then the content of "detail service" is by this means a subset of the "list service". If the detail service is unreachable, when the user click on a marker to view the detail, he/she gets nothing without utilizing the cached response from "list service". Both of the above should be resolved by incorporating customized cache strategies from developers.

III. APPROACH OVERVIEW

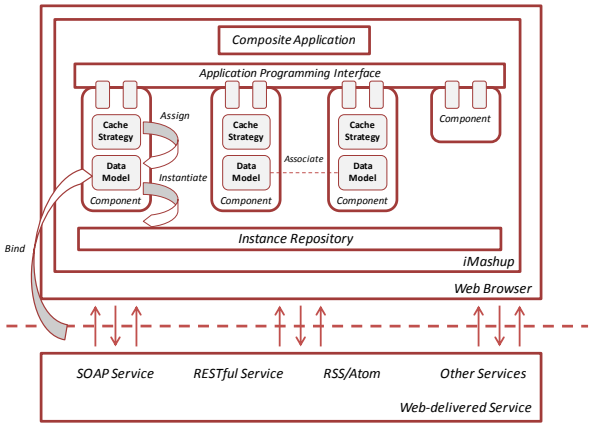

Figure 2. Data Model and Cache Strategy in iMashup

As described in the motivating scenario, a feasible solution is to enable developers to customize cache strategies in a browser-side framework. The design of the framework is independent of language, and could be incorporated easily in various composite application tools such as [5] [7]. In this paper, the implementation of the framework is realized in iMashup in order to facilitate the demonstration of our approach. Figure 2. shows the general architecture of our framework. We define the basic unit of component models in the browsers, which encapsulate both the application logic and UI. After building the component models from the data or logic of services, programmers need to build up data models and the corresponding cache strategies. This process is only required for those components the developers need to cache. The rest of components which should not be cached (as illustrated on the right side of the figure) would be left alone. When executed at run-time in Web browsers, everything including validation is transparent to developers.

When building the data model within a component, as illustrated in Figure 2. , developers are required to do two things: binding Web-delivered services with data models, and setting up cache strategies for models. We believe the data centric approach during the binding process is suitable, since there are many previous work in separating data transformation logic and I/O concerns in composite applications such as Reo [6].

- Binding services with models: binding is a customization process. Through our framework, developers select the desired properties and map them into data models. This is a one-to-many mapping because a service could be perceived as various presentation forms. After the mapping, developers set up data fields, which sources from the original services. The last step is to decide the primary keys for the model, which are used as the criterion for indexing. For example, the city name could be the primary key of the weather service.
- Setting up cache strategies for models: aside from the pure data, every data model has to carry an additional property (we denote it as Meta-Property) which includes the caching strategies. Developers would set up most parameters in the Meta-Property, including data type of data fields in the models and cache strategy. The cache strategy includes but not limited to enable/disable cache, expiration, the time since last access, entry time and access frequency as discussed in [9]. In our framework, the first two should be assigned the initial values by developers, and the others are covered by the framework. For example, developers might assign "enable cache, expiration time of 5 minutes" as the default policy to a model, and our framework would compute other strategies in an on-the-fly manner. After the above two steps, all the models and corresponding cache strategies are ready to apply.

At run-time, as illustrated by Figure 3. , our framework cache data models automatically. To this end, it follows these steps.

- Intercepting user requests: to take over all the necessary requests, it is necessary for developers to redirect their requests to our framework. On the other hand, all the requests otherwise would not be affected. By this means our framework brings more flexibility to developers. After intercepting the requests, some logical operations are performed to convert the requests into attribute-oriented queries (Attribute Op Value) to the instance repository. This repository is a hash table in realization.
- Querying the instance repository: when a request is invoked, all the related instances are checked for a cache hit. This process is executed on both the desired data model and all the related models—as the desired data model might be contained by other data models. Judging the hit is accomplished by comparing request parameters in the target URI with primary keys of every model, which are specified once building data models. If an instance is hit, our framework returns the desired model

instance. If no hit produced, our framework proceeds to the next step for inquiries.

- Inquiring and caching: the URI for remote servers is specified in each data model. Therefore, the service calls are automatically done and are totally transparent to developers. Our framework would call the remote services, parse and store the generated instances to instance repository, and then invoke the callback function specified by users.

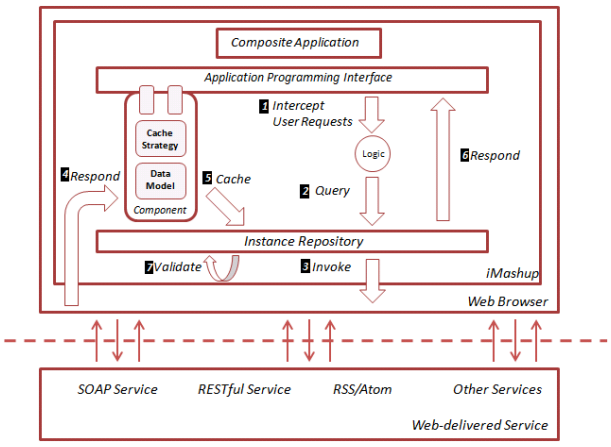

Figure 3. Cache Framework at Run-time

For our scenario, the general process of using our framework is as follows. In the first step, we build components for all services, and then bind the list service and the detail service of eBay and Amazon with models. In the models, the "list model" contains an array of the "detail models". In the cache strategies, an expiration of 30 minutes is set up for models. Subsequently, during the run-time, all models are instantiated and stored in the instance repository. Therefore, both of the "list" and "detail" models would reuse these instances in the following cases. First of all, users do not need to communicate with servers when they are sending identical requests within the customized expiration time. Secondly, when fetching the providers' detail information, all instances of the detail model and all instances in the minor structure of the list model would be searched.

IV. The Cache Framework Embedded in Browser

A. Principles of Implementation

For all Web-delivered services with well-structured responses, we claim that each of their responses compromises either an object or a collection of objects. The types of responses could be classified as $\{object_i \mid i = 0,1,2,...\}$ or LT (list type), and $object$ or OT (object type).

Based upon the above, when two service calls are invoked sequentially, there are five cardinal cache cases where the framework could provide cached data from the first call to feed the second request. We denote these cases as cache cases in the rest of this paper.

- Identical. The second request has the exactly same URL and parameters with the first one. It is the classical case of cache.
- List-Object: $\{object_i \mid i = 0,1,2,...\} \ni object_k$. This case comes when the OT response is contained by LT. Imagine the first service is to return a list of 10 bestsellers, while the second one is to return the most popular book. Then second response is included thoroughly in the cache from the first response.
- Object-List: $object_i \ni \{object_k \mid k = 0,1,2,...\}$. If an OT response is a composite data structure with a list structure inside, then the following LT call might reuse the overlapping data. For instance, the OT is a description for a person. Within the description, there is a list of his/her friends. Therefore, after retrieving the person's description, all requests for his/her friends might benefit from cache.
- Object-Object: $object_i \ni object_k$. This is the case when users send OT requests for different integrities of the response. Suppose if the first call is for detailed information of a person with ID and description, while the second call is only to retrieve the person with his/her ID.
- Object-Object: $\{object_i \mid i = 0,1,2,...\} \supseteq \{object_k \mid k = 0,1,2,...\}$. This is the case when users send LT requests in different scopes. Suppose if the first call inquiries a collection of 10 bestsellers from Amazon, and the second call is for the top 5.

The four cache cases facilitate our realization as stated in the following.

B. Realization of Cache

The response type and cache cases of a Web-delivered service are set up in the data model, when building a component. During this process, developers assign an XPath[8] expression for every data field indicating how to parse it. After that, developers need to decide the primary keys in a model for model comparison. Then, the response types (LT/OT), cache cases, the other party in each case for the model are specified as well.

In our framework, a match is achieved based on a strict URL comparison. When a request comes, our framework looks up its corresponding data model according to the URL pattern. After that, the framework needs to determine all related data models to be searched in the instance repository. The processing logic for different cases is discussed as follows.

- The first four cases discussed in the previous section would be handled automatically and uniquely, which is (1) to retrieve the data model of the other party in such relationships; (2) to search instances of the data models in (1) in the repository;

[8] XPath, http://www.w3.org/TR/xpath

- The last case (one list contains another) is more complicated. We demand all developers to supply an extra query interface for such a LT model (only for models involved in such a cache case). In that query interface the programmers should explicitly indicate how to query related models. For instance, if a LT model is to return a list of books from Amazon, it should embody a query interface which takes the start and end indices as parameters to return the "slice" list. Therefore, after caching a list of "top 100 books", a subsequent inquiry for a list of "top 30~top 50" books could be handled by this inquiry interface with desired results.

When searching all related data models, judgment for a cache hit is realized through primary keys and the parser logics specified in the model. If not matched or the instance is stale, our framework would make the remote call and return to users after the server responds. Otherwise, a hit is achieved and the matched instance is returned.

The Meta-property in the data model consists of:

- Data Type: The data type declares data type for each property in the model to facilitate the inquiries. It could be either simple data types (date & time, float, object, string etc.) or complex data types (such as other data models).
- Cache Strategy: The cache strategy is employed to achieve expiration and replacement. Currently, we support the following policies in the strategy:

TABLE I. THE CACHE STRATEGY OF THE META MODEL

Policy Name	Implementation
Time since Last Access (TLA)	A timestamp for the last access
Entry Time (ET)	A timestamp for instantiation
Frequency of Access (FOA)	A number to keep on counting the frequency
Expiration	A timer for validity of the model or data fields

In the above table, the expiration field and ET determine when cached data items are expired. And TLA and FOA determine the target for replacement when repository exceeds the size limit.

Our framework sets up the above Meta model for each data model. During run-time, after the instantiating of every model, a countdown timer is initiated to time the expiration, which triggers the framework to invalidate. The models are requested again according to users' need.

C. Adaptive Cache

Users access the cached data irregularly, so a fixed expiration might be inappropriate over the time and then affect the whole system's performance. On one hand, if the expiration is too short-lived, the hit ratio will be low and the system must send more queries to the services, wasting more resources. On the other hand, if it is too long, users have to use the stale results, making the data less reliable.

Moreover, many factors contribute to the changes of the cache. These factors come from both the user behaviors and service behaviors. For the user factors, hit-ratio and access frequency would refine the expiration time. As for the services factors, the update frequency of services would update cache as well. For example, some services change their results by seconds like services fetching stock prices or messages, while others just return the same result most time we invoke it. Though tempting to include all factors, we ignore the service ones in a client-side framework.

Thus, we propose an improved cache strategy. In this strategy, our framework would compute the desired expiration time by an algorithm for users based on hit ratio and access frequency. The algorithm run on changes of these two factors, which includes the following occasions:

- Users access the cached data, which returns either a hit or a miss. So the hit ratio is changed.
- Users access the cache either more "quickly" or "slowly", which in turn alter the access frequency.

The algorithm borrows the idea of the simulated annealing algorithm [14], but differs from it in addressing the requirements of finding the most appropriate expiration time for users. In this strategy, expiration time adapts to users dynamically according to the hit ratio and users' behavior in the history (access frequency).

The estimated value Vi denotes the expiration time for each cache item Ii. We denote Va for a threshold value, then Vi is calculated as below.

Formula 1:

$$V_i = e^{\frac{\alpha \times hit\_ratio}{temperature}}, if\ (Vi > Va)$$
$$V_i = NA, if\ (Vi \le Va) \quad (1)$$

The variables in formula (1) are explained as the followings.

- Variable hit_ratio denotes the percentage of the hit times from all the visiting of cache item Ii. We enlarge the influence of hit_ratio by multiplying it by a constant α, which is assigned to 1000 in the implementation. The higher the hit_ratio's value is, the larger the expiration time value will be. This is what cache is designed for. It is also reasonable to simply enlarge the expiration time to make the cached data be effective for a longer period of time. In contrast, if the hit_ratio is small, it means that the current cached data has little merit to be visited again, indicating more potential to become stale. In this case, we should assign shorter expiration time to cut down the data's lifespan. Since an extreme short expiration makes the cache little use, we set up a threshold value and would turn off cache when the expiration drops below it.
- Variables temperature denotes the sensitivity for the expiration time to change with hit_ratio. In addition, it essentially reflects the current frequency of accessing the services. The user may use the same query to visit the same service many times very quickly, causing hit_ratio to increase remarkably if it is very sensitive. Oppositely, if user doesn't use the service quite often, we cannot get enough experience to adapt the expiration. As a result, the sensitivity should be adapted based on the frequency of accessing.

During the annealing process, temperature is adjusted as follows:

Formula 2: Let 'T' be the temperature and minT be its lower bound. Let 'Foa' denote frequency of accessing by users, and 'theta' be the mean value of Foa. Then:

$$T = Max\{\min T, (Foa - \theta)^2\} \quad (2)$$

This formula states that temperature is increased when frequency of accessing goes extreme and decreased when it reaches its mean. However, the sensitivity of the expiration time cannot be too small, which leads to an extreme occasion. So we manually define a lower bound to limit it. The annealing process ends when temperature drops to minT. Foa's increase means that all the services will have more chance to be visited in the period compared to normal. In this case, if the expiration time is seriously influenced by hit_ratio, it might get a very large value like 1 year or fall to a very small value like 0.1 second, both of which are unreasonable. So we must increase temperature to decline the influence of hit_ratio. On the opposite side, if Foa is too small, it might cost the system a long time to collect enough information to adapt the expiration time to the best. In this case, we must decrease temperature to add the influence of hit_ratio. In a word, temperature should be positively affected by frequency of accessing.

V. EVALUATION

To demonstrate the contribution of this paper, we have conducted an experiment that highlights the usability and effectiveness of our framework. The first evaluation is done without the adaptive approach (so we set the expiration as a constant). Then, we incorporate the adaptive approach to see a further cache enhancement in action.

A. *Evaluation of the Scenario*

To come up with a comprehensive evaluation, we implement the case discussed in our motivating scenario with the following Web-delivered services: eBay Shopping API, Amazon and Google Map. The demonstration is run in Firefox 3.0.10.

Via our framework, we build the following data models: two LT models and two OT models for the two shopping services, and one OT model for Google Map. We assign the following cache cases for them: LT and OT models of the shopping are assigned with the cache cases of "Identical" and "List-Object $\{object_i | i = 0,1,2,\dots\} \ni object_k$". The OT model of Google Map is simply a marker service, thus is assigned with the cache case of "Identical". We also make our own cache strategies as discussed previously—refresh for all models is not necessary within thirty minute from last response, and the cached data should be validated automatically after that. We run the evaluation against the scale of requests to reflect the fact that a composition application could be published and accessed heavily.

The evaluation is done by writing a script which randomly invokes requests on behalf of a user. To better ponder the validation, our script would pause 5 seconds between every two requests. We record the outgoing requests versus the increase of requests with and without our framework. It is worth noting here the memory consumption of the framework is dependent upon the practical data of cases other than the framework mechanism, thus it is not included in the evaluation.

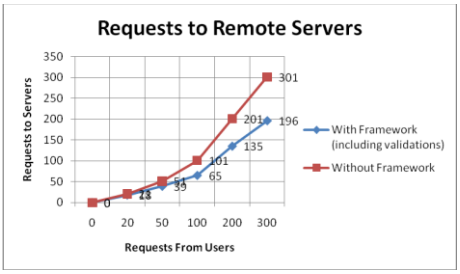

Figure 4. Requests to Remote Servers Comparison

The first evaluation is to evaluate the efficiency of the framework. The X axis refers to the total number of requests invoked from users, and the Y axis refers to the requests that are actually made to servers. More requests are replied from caches as the requests rise. This is due to both the ability to cache and data reuse in OT models of shopping. The same cases happen in reality when users prefer to quickly browse all available providers while going back and forth periodically. Moreover, if the strategy conducts the validation less frequently, requests to servers would further decrease.

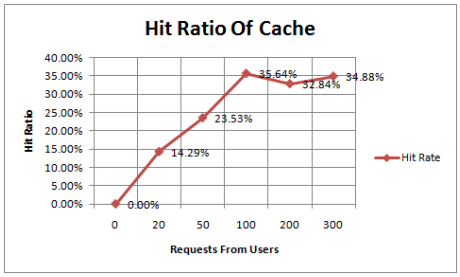

Figure 5. Hit Ratio of the Framework

The second evaluation is to evaluate the performance of the framework. The X axis takes the same meaning as above. The Y axis in the left chart refers to the number of hits to the total number of requests from users. The above results reveal the following observations.

- The hit rate starts highly for even small scale of requests. This is due to the concrete behavior of the script and the specific scope of this application. Since the descriptions of the providers seldom change over a long period of time, our framework incorporate obvious improvement over the otherwise;
- The hit rate rises slightly as the size of requests increases, while sliding down a bit around 32%. The increase of hit rate comes from the increase of available instances. And the decrease is because some instances become out-of-date and require validation, which would diminish the hit ratio by missing requests while validating these models.

From the evaluation above, we demonstrate that our framework has a sound performance in exerting the cache strategies from developers. And under appropriate strategies which balance the traffic and validity well, the corresponding composite applications would benefit from the framework without burdening too much on traffic.

B. *Evaluation of Adaptive Cache*

We have firstly conducted an independent experiment from the one above, in order to find out how our algorithms in IV.C changes the expiration time, so it would adapt to users' behavior. After that, we then add the adaptive approach to the scenario to gain a further improvement.

We have implemented a web application that calls only one web service. There is a script invoking the service on behalf of the users with irregular access frequencies and different parameters (from a parameter collection that is large enough). We have timed the process and recorded the following factors: frequency of access (Foa), hit-ratio, temperature and Vi. The records have the unit of 10 times, and the time internal is 2 minutes.

The first step of the experiment is done without the adaptive cache. We generate a series of requests (denoted as R0) irregularly, which means the frequency of invocations would change in some period of time.

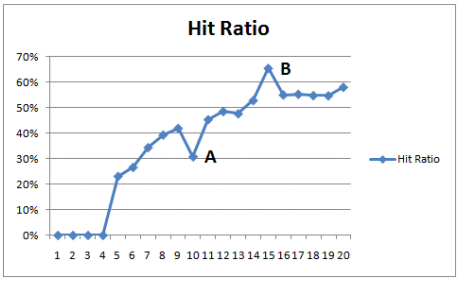

Figure 6. Hit Ratio against Time Without Adaptive Cache

From the figure, we could find out that there is a dramatic fall of hit-ratio at Point A and also an increase at Point B. The reason is around the period of A, users send too many requests with different parameters, which lead to significant cache miss. While in the period of Point B, users just send identical requests which increase the hit-ratio. As discussed previously, our approach should alleviate the dramatic expiration change at both Point A and B. And we should also allow expiration to change much more quickly around other time.

Then, we turn on the adaptive method, send the exactly same requests R0 in the same order and frequencies, and obtain the following data about temperature.

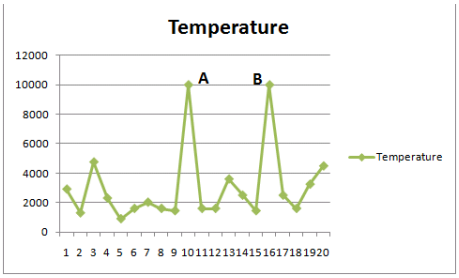

Figure 7. Temperature against Time

According to Formula 2, temperature is affected by Foa. From the graph we can observe that around the period of A and B, temperature rises significantly due to the large quantity of requests. And according to Formula 1, since high frequency results high temperature, which "slows down" the process of "annealing", we expect the Vi to drop smoothly at the Point A and increase smoothly at Point B. At other points, temperatures are rather low so Vi should change quickly.

Finally, we check the resulting expiration time (which has a linear conversion from Vi) from the impact of temperature.

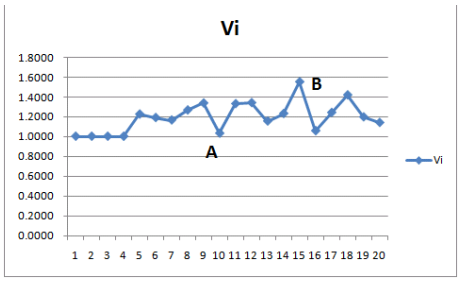

Figure 8. Vi against Time with Adaptive Cache

From Figure 8. and formula 2, we are generally more concerned about how Vi perform at A and B among all the dots. The relatively small decline at A shows a desirable result, which shows our approach keeps the expiration stable when too many cache miss occur. In this way, if we set up the threshold value of Va as level of A, the framework could turn off the cache to reduce the unnecessary memory occupation. The same effect could be observed on the smooth increase at Point B. On the other hand, the fluctuation on the left segment of the curve shows our annealing algorithm changes the expiration time relatively quickly at a lower temperature. This makes sense to find the desired expiration quickly without too many history data.

Consequently, we run section V-A again with the adaptive cache on. The comparison of the requests is as below:

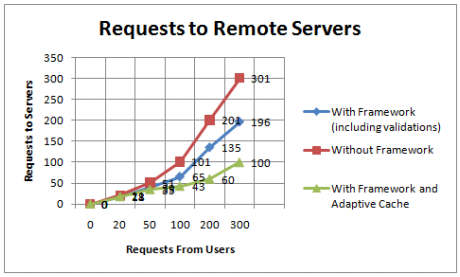

Figure 9. Requests to Remote Servers Comparison(2)

From the experiment above, we show that the adaptive cache and the annealing algorithm suit some scenarios rather well. It guarantees the stable changes of expiration time, and encourages quick adaption at the inception of scenarios.

VI. DISCUSSION

There are the following open issues of our framework.

First of all, though the framework is proposed to cache all kinds of Web-delivered services, it might be improper for some exceptions. For instance, the data size of some

service responses might be extremely large. And if these data is referenced frequently, the framework would place the data in the memory. This might lead to high memory consumption and slow reaction of the applications. For such cases, our framework might not be so effective in caching service responses.

Secondly, our framework is implemented on the browser-side. The use of it might be restricted for server-side applications because many developers compose services on the server-side and provide the final web pages to users. To enable caching for these applications, the framework should bear a server-side implementation. Since the mechanism and the run-time environment are loosely coupled in our framework, it might not be tough work to realize such an environment.

VII. RELATED WORK

The concept of semantic cache is proposed to reuse the overlapping cache data with the help of the semantic knowledge of the data [12]. The semantic cache focuses on how to exploit the semantic information within the query and how to use this information for the future reuse [13]. We commonly believe semantic cache is a very desirable reference for our framework. Because data models with semantics are similar with result sets in semantic cache, and the algorithms for classifying, reusing and replacing [11] could supplement our relatively simple models here. However, it is also worthy to note that semantic information in responses from Web-delivered services are more variable and complicated than the query-based responses (usually expressed as name-value pair) of semantic cache. Thus our framework differs from it in scope and approaches.

Depending on the hierarchy differences, there are two kinds of browser-side cache. The first kind is to cache on the data tier, such as the browsers' built-in cache or cache modules in the browser add-ins (e.g. Runtime Shared Libraries in Flash [15]). This sort of cache would map URLs to content according to HTTP/1.1. A number of elements in the HTTP HEADER are included to direct cache behaviors, which are specified by service providers. The second kind resides on the application tier, which are usually embedded in client side JavaScript frameworks [8]. Nevertheless, popular JavaScript frameworks such as DOJO[9] and DWR[10] merely cache the data on basis of the entire Web pages or responses, providing inadequate flexibility for developers.

VIII. CONCLUSION

The composite applications from Web-delivered services are proliferating on the Web. Most of them are employing a cache pattern with little support to customize developers' own cache strategies. Our framework fills such a gap and makes the following contributions.

We propose a cache framework to support developers to make cache strategies when developing composite applications. These strategies include the granularity of cache, expiration time and so on. We also propose an adaptive approach for strategies. Therefore, in terms of developers, they get more flexibility when developing composite applications. While in terms of end users, they get customized cache behaviors which improve the cache efficiency.

[9] DOJO, http://dojotoolkits.org

[10] DWR, http://directwebremoting.org/dwr/index.html

IX. ACKNOWLEDGEMENT

This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60821003, 60873060; the High-Tech Research and Development Program of China under Grant No. 2009AA01Z116; and the EU Seventh Framework Programme under Grant No. 231167.

REFERENCES

[1]. Eyhab Al-Masri and Qusay H. Mahmoud, Investigating Web Services on the World Wide Web, WWW 2009

[2]. Jin Yu et al. Understanding Mashup Development, IEEE Internet Computing, 2008

[3]. Yan Li et al. An Exploratory Study of Web Services on the Internet, IEEE International Conference on Web Services, 2007

[4]. Volker Hoyer and Marco Fischer. Market Overview of Enterprise Mashup Tools, International Conference on Service Oriented Computing, 2008, LNCS 5364, pp. 708–721.

[5]. E. Michael Maximilien et al. A Domain-Specific Language for Web APIs and Services Mashups. International Conference on Service Oriented Computing, 2007, LNCS 4749, pp. 13-26.

[6]. Ziyan Maraikar, Alexander Lazovik and Farhad Arbab. Building Mashups for the Enterprise with SABRE. International Conference on Service Oriented Computing, 2008, LNCS 5364, pp. 70-83.

[7]. Michael Pierre Carlson et al. Automatic Mash Up of Composite Applications. International Conference on Service Oriented Computing, 2008, LNCS 5364, pp. 317-330.

[8]. G. Barish and K. Obraczka: World Wide Web Caching. Trends and Techniques, IEEE Comm. Magazine, Internet Technology Series, 2000, pp. 178-185.

[9]. Wei-Guang Teng et al. Integrating Web Caching and Web Prefetching in Client-Side Proxies, IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 5, May 2005

[10]. Qi Zhao et al. A Web-based Mashup Environment for On-the-fly Service Composition. 4th International Symposium on Service-Oriented System Engineering (SOSE 2008), pp 32-37. Dec. 18-19.

[11]. Y. Arens and C.A. Knoblock. Intelligent caching: selecting, representing, and reusing data in an information server, Proc. CIKM'94 Conference, Gaithersburg, MA, 1994, pp. 433–438.

[12]. S. Dar, M.J. Franklin et al. Semantic data caching and replacement. Proc 22nd VLDB Conference, Bombay, India, 1996, pp. 330–341.

[13]. Boris Chidlovskii et al. Semantic cache mechanism for heterogeneous Web querying. The International Journal of Computer and Telecommunications Networking, Volume 31, pp. 1347 - 1360, 1999

[14]. S.J.Russsel and P. Norvig Artificial intelligence: a modern approach. Prentice-Hall, 1995

[15]. RSL, http://www.adobe.com/devnet/flex/articles/rsl.html