Sequential Pattern-Based Cache Replacement in Servlet Container

Yang Li, Lin Zuo, Jun Wei, Hua Zhong, and Tao Huang

Technology Center of Software Engineering, Institute of Software
Chinese Academy of Sciences, Beijing 100080, P.R. China
{fallingboat, martin_zl, wj, zhongh, tao}@otcaix.iscas.ac.cn

Abstract. Servlet cache can effectively improve the throughput and reduce response time experienced by customers in servlet container. An essential issue of servlet cache is cache replacement. Traditional solutions such as LRU, LFU and GDSF only concern some intrinsic factors of cache objects regardless of associations among cached objects. For higher performance, some approaches are proposed to utilize these associations to predict customer visit behaviors, but they are still restricted by first-order Markov model and lead to inaccurate predication. In this paper, we describe associations among servlets as sequential patterns and compose them into pattern graphs, which eliminates the limitation of Markov model and achieve more accurate predictions. At last, we propose a discovery algorithm to generate pattern graphs and two predictive probability functions for cache replacement based on pattern graphs. Our evaluation shows that this approach can get higher cache hit ratio and effectively improve the performance of servlet container.

Keywords: Servlet Cache, Sequential Patterns, Cache Replacement.

1 Introduction

Recently, Java EE has become mainstream middleware platform for large-scale enterprise applications. As the core part, servlet container [1] provides runtime environment for servlet components and plays a key role in the performance of Web applications. However, servlet performance report [2] presented by Web Performance Inc. in 2004 showed that challenges still exist to improve the performance of servlet container. Many solutions based on cache technology are proposed to gain performance improvement in web server and proxy server [4] [5]. But they are hard to adapt to servlet container because responses generated by servlets are dynamic and dependent on values of input parameters.

In servlet container, there exist lots of servlets which are frequently accessed, and their outputs or responses keep unchanged or stable in some period if their parameter values are unchanged, e.g. catalog pages within a shopping application or an events calendar on a university web site [5]. We call them as cacheable servlets. Therefore, the responses of these cacheable servlets can be cached and accessed by multiple clients to significantly improve response time experienced by customers. Cacheable servlets have been supported in Websphere [6].

L. Baresi, P. Fraternali, and G.-J. Houben (Eds.): ICWE 2007, LNCS 4607, pp. 105 – 120, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Cache replacement is an essential issue in servlet cache. Traditional cache replacement algorithms such as LRU [3], LFU [7] and GDSF [12] mainly utilize a combination of several intrinsic factors of cache objects (e.g. file size, the recentness and frequency) to design replacement cost functions, which can enhance cache hit ratio and performance of servers to a certain extent. However, they ignore business functionaries represented by servlets and business associations among them.

Some researchers have observed these associations among business components and adopted them in different areas. Access patterns [8] are firstly introduced to database system to represent associations among data items. Access patterns are also utilized to prefetch cached web objects by mining server logs to achieve the associations among web pages [9]. Qiang [14] also proposes one discovery algorithm for proxy server based on web access pattern mining and gives the n-gram replacement algorithm to improve cache hit ratio. In general, access patterns mainly includes association rule [9] and sequential pattern [9][10][11], and sequential pattern is widely used in Web system.

Markov model is often adopted to describe sequential patterns. Some researchers have made use of associations among web pages to predict customer visit behaviors for link prediction and path analysis [21][22], but most of them are under first-order Markov model, which assumes the future visit behaviors of customers are only influenced by their current behaviors. Therefore, first-order Markov model can not accurately reflect customer visit behaviors at all time because it brings some inexistent behaviors from multiple steps predictions and leads to invalid replacements. For example, suppose there exist five servlets S_1, S_2, S_3, S_4 , S_5, and customer visit behaviors are S_1->S_3->S_4(10 times) and S_2->S_3->S_5(10 times). The predictions based on first-order Markov model are described in Fig.1, where direct predictions are accurate (Figure1. a). However, the two steps predictions (Figure1.b) include inaccurate behaviors S_1->S_5 and S_2->S_4, which never happened in actual behaviors.

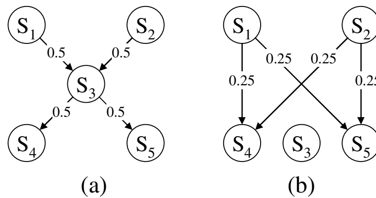


Fig. 1. The Predictions Based on First-order Markov Model

In this paper, we proposed a novel cache replacement approach to support servlet cache. We first describe the business associations among servlets as sequential patterns and compose them to form pattern graphs, which do not suffer the invalid predictions of first-order Markov model and reflect customer visit behaviors more accurately. Then, we present a discovery algorithm to discover above pattern graphs according to customer visit histories, and two predictive probability functions based on pattern graphs are presented, which can be combined with traditional cost functions to improve the effectiveness of cache replacement. We discuss this approach in the context of servlet container. However, we believe the principle idea can be applied

to other dynamic environments, such as ASP and PHP. Our evaluation shows that this approach greatly enhances cache hit ratio, reduces redundant reexecution of cacheable servlets and gains more attractive performance in servlet container.

In the next section, we briefly review related work. Some basic definitions and cache system model of servlet container are presented in section 3. The definitions of sequential patterns and sequential patterns discovery algorithm is described in section 4. Section 5 presents sequential pattern-based replacement cost functions. In section 6, we provide simulation and performance evaluation of the proposed approach and conclude the paper in section 7.

2 Related Works

Many cache replacement algorithms use a combination of several factors such as the visit time and visit frequency, the size, and the cost of fetching a document, which lead to a significant improvement in cache hit ratio and latency reduction in proxy server and web server. LRU [3] replaces cache object with the oldest visit time. LFU [7] evicts cache object with minimal visit frequency. LRU-threshold [13] extends LRU and only caches the object whose size is less than a threshold value. GDSF [12][14] is based on visit time, file size, visit frequency and loading cost etc, which evicts cache object with minimal replacement cost. However, all these algorithms only utilize some factors of cache objects itself (visit time, visit frequency, file size, loading cost, and so on) as criterion to replace cache objects, and ignore associations among cache objects which can be applied to cache replacement and greatly enhance cache hit ratio. In addition, they mainly suit for static files and are hard to handle dynamic servlet responses.

Some approaches are proposed to utilize sequential patterns to describe business association among cache objects. Agrawal [15] first adopts sequential patterns to describe the associations among data items in database and present three algorithms to discover them. Huang [16] analyzes user access sequential patterns and design a prediction-based proxy server to improve hit ratio of accessed documents. Sarukkai [21] uses Markov chain to solve probabilistic link prediction and path analysis in web server. To improve the efficiency of link prediction, Zhu [22] uses Markov model to construct the structure of a Web site based on past visitor behaviors. However, these approaches only adapt to static objects and can not support cacheable servlet responses which depend on the values of parameters. In addition, some of them are base on the first-order Markov model in which the objects that a client visits in future are only determined by its current position. Therefore, these approaches can not accurately reflect business association among servlets on multiple steps predictions.

Compared to current approaches, our contributions are summarized as follows: 1) we describe business associations among servlet as sequential patterns. 2) We utilize pattern graphs to compose above sequential patterns, which can reflect multiple steps associations and get rid of the inaccuracy of Markov model. 3) We design a discovery algorithm discoverTPG to gather pattern graphs at runtime. 4) Based on pattern graphs, we propose two sequential pattern-based replacement cost functions to makes a supplement for traditional replacement cost functions. To the best of our knowledge, this is the first attempt to utilize sequential pattern in servlet cache.

3 Servlet Cache Model

In this section, we present some basic definitions about servlet cache and discuss servlet cache model in servlet container.

3.1 Basic Definitions

Definition 1. Servlet s can be defined by *s=<id, name, parameters, type>* where *id* and *name* are identity and name of *s*, *parameters* is a group of parameters of *s*, *type* is the type of *s* and *type* ∈ *{cacheable, non-cacheable}*.

When *type* is *cacheable*, it means responses generated by *s* can be cached. On the contrary, responses of *s* can not be cached if *type* is *non-cacheable*. In servlet container we represent all servlets by S, all cacheable servlets by S^C and non-cacheable servlets by S^{NC}. For convenience, we put all cacheable servlet before non-cacheable servlets in S, which means s_i*.type=cacheable* when $1<i \leq L$ and s_i*.type=non-cacheable* when $L<i \leq N$ if $S=\{s_1,s_2,\ldots s_i,\ldots s_N\}$ and $|S|=N, |S^C|=L$.

Cacheable servlets are often defined in a configure file according to their business functionaries by application developers, which generally only return some query information such as product information or history of stock. They are generally visited frequently but generated responses are changed rarely when values of parameters are identical. We can cache their responses to service multiple customers, which makes that servlet container need not execute cacheable servlets to generate same responses for each customer at each time.

Definition 2. Servlet cache object is defined as a five-tuples: *scache =<id, s, values, response, factors >* where

- id denotes a unique identity of cache object
- s denotes a cacheable servlet
- values denotes the values of s.parameters
- response denotes the response generated by s when the values of s.parameters are values
- factors denotes factors of cost function

Comparing with static files, servlet cache object depends on not only cacheable servlet but also its parameter values. The *scache.factors* is different according to specific cache replacement algorithms. For LRU, the *scache.factors* is {age}, and the *scache.factors* for GDSF is {age, frequency, cost, size}. The meanings of the factors will be detailed in section 5.

Web Characterization Activity (WCA) defines a user session as the click-stream of page views for a single user across the entire web site [23]. In servlet container we also define a session just like what they do.

Definition 3. A **session** represents a sequence of servlets visited by a customer, which can be defined as $session=\{id_1,id_2, id_m\ldots,id_{length}\}$, where id_m is *id* of a servlet and *length* represents the length of *session*.

Next, we will detail the cache model of servlet container.

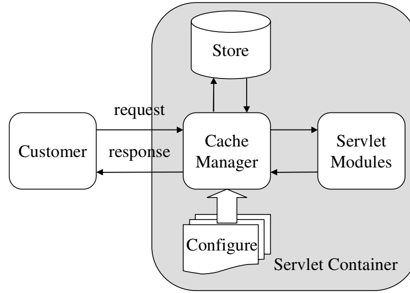


Fig. 2. Servlet Cache Model

3.2 Servlet Cache Model

Fig.2 describes the cache model of servlet container. Here **configure** defines the cacheable servlets; **cache manager** stores **servlet cache objects** into **store** and manages their consistency; **servlet modules** generate response to service customers.

Algorithm 1 explains how above cache model works. ServletContainer.execute(*s*, *values*) means execution of *s* with values of *s.parameters* (line 5). CacheManager.get (*s.name*, *values*) means to lookup corresponding *scache* in store according to *s.name* and its parameters values (line 9). CacheManager.updateFactors(*scache*) means to update some factors of *scache* such as visit time and frequency (line 11).

From Algorithm 1, we can see that the process flows are different according to servlet type. If s.type is non-cacheable, servlet container will directly execute s with the values of s.parameters (line 5) and return generated response to customer (line 6). But when s.type is cacheable, servlet container will first lookup corresponding scache from store (line 9). If scache exists, some factors of scache are updated, and scache.response is returned to customer (line 12). If scache does not exist, servlet container does what it does for non-cacheable servlet (line 15) and creates new scache object to store into store (line 16, 17). When the store overflows upper limit, some scaches are evicted according to replacement cost function rank(scache) (line 19).

Obviously, replacement cost function **rank (scache)** is the key of cache replacement, which determinates which scaches should be evicted. Next we will detail our approach, and describe two more effective replacement cost functions based on this approach in section 5.

```
 1.   s: servlet that client requests
 2.   values: the values of parameters of s
 3.   function request( s, values){
 4.    if( s.type == non-cacheable ){
 5.      response = ServletContainer.execute( s,values);
 6.      return response;
 7.    }
 8.    else if( s.type == cacheable ){
 9.     scache= CacheManager.get(s.name, values)
 10.     if(scache != null){
 11.       CacheManager.updateFactors(scache);
 12.       return scache.response;
```

```
13.      }
14.      else{
15.       response = ServletContainer.execute(s, val-
ues);
16.       scache= new scache(new(id), s, val-
ues,response);
17.       CacheManager.push(scache);
18.       while(Store.size > Store.maxsize ){
19.         Run cache replacement algorithm to evict the
            scache with minimal values of rank(scache).
20.       }
21.       return scache.response;
22.      }
23.     }
24.   }
```

Algorithm 1. The Principle of Servlet Cache Model

4 Sequential Patterns in Servlet Container

4.1 Basic Definitions

Srikant [8] defines sequential patterns in database as "5% of customers bought 'Foundation' and 'Ring world' in one transaction, followed by 'Second Foundation' in a later transaction. In servlet container, we use similar description to define sequential patterns as "15% of customers visited servlet 'searchBook' followed by servlet 'showBookDetail' in a later visit". Some basic definitions of sequential pattern are presented as follows.

Definition 4. Transition, if a *session* contains a sequence like $\ldots id_m,\ldots,id_{m+d}\ldots$and $id_m = s_{source}.id$, $id_{m+d} = s_{target}.id$, we think that there exists a transition from servlet *source* to servlet *target* and transition distance is *d*, which is defined as: $s_{source} \xrightarrow{d} s_{target}$, where s_{source} , s_{target} and d are source, target and distance of the transition, respectively.

A transition is called **self transition** if its source and target are the same servlet. For example, if a servlet s returns product information according to *productid*, a self transition happens when a customer visits s with different *productid* continuously.

Definition 5. Sequential pattern is defined as the probability that transition $s_{source} \xrightarrow{d} s_{target}$ happens in servlet container.

Sequential patterns in servlet container represent the business associations among servlets. We can use them to predict the probability that a cacheable servlet response is visited again and evict cache objects with minimal probability.

4.2 Pattern Graphs

Definition 6. Transition graph TG(d) is a directed weighted graph, which comes from a group of session *sessions*: session(1), session(2), …… , session(n). Vertex i of

TG(d) represents servlet s_i, edge e<i, j> represents transition $s_i \xrightarrow{d} s_j$ and weight of edges $W_{TG(d)}(i, j)$ represents the times that $s_i \xrightarrow{d} s_j$ happened in *sessions*.

Definition 7. P (i, j, d) denotes the probability that transition $s_i \xrightarrow{d} s_j$ happened in a group of session *sessions*. Obviously $P(i, j, k) = W_{TG(k)}(i, j) / \sum_{m \in EndS} W_{TG(k)}(i, m)$, where $W_{TG(k)}(i, j)$ represents the weight of edge e<i, j> of TG(k) and *EndS* is collections contained all end of transition $s_i \xrightarrow{k} s_m$ in *sessions*. P(i, j, d) is sequential pattern in servlet container according to definition 5.

Definition 8. Transition probability graph TPG(d) is also a directed weighted graph from a group of session *sessions*: session(1), session(2), …… , session(n). Its vertices and edges have the same meaning with TG(d), but the weight of edge $W_{TPG(d)}(i, j)=P(i, j, d)$.

TG(d) and TPG(d) are called pattern graphs (PG), they record customer visit behaviors from different viewpoints. The former records customer visit behaviors honestly and latter records the trends of customer visit behaviors. Before using them to design replacement cost functions, we first take an example to illustrate pattern graphs (d=1, 2) and compare them with Markov model.

Suppose there are 5 servlets and $S^C = \{s_1, s_2, s_3\}$, $S^{NC} = \{s_4, s_5\}$ in servlet container. The sessions of three customers are as follows:

session (1)=$\{s_1,s_2,s_3,s_5,s_5,s_2,s_3,s_4\}$
session (2)=$\{s_1,s_2,s_4,s_3,s_4,s_1,s_2,s_3,s_5,s_5\}$
session (3)=$\{s_2,s_3,s_5,s_1,s_2,s_4\}$

Obviously we can get N=5, L=3. We represent cacheable servlets by grey nodes and non-cacheable servlets by white nodes. Then, the corresponding pattern graphs can be illustrated in Fig.3 (a) ~ (d).

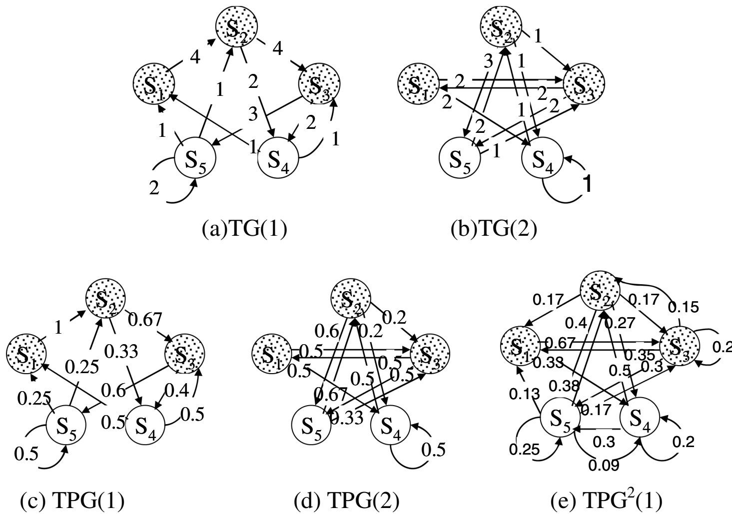


Fig. 3. Pattern Graphs

Current approaches mainly adopt first-order Markov model to predict the future customer visit behaviors according to visit histories [21][22]. First-order Markov model gets TPG (n) by calculating the power of TPG (1), namely TPG (n) =$TPG^n(1)$. However, these approaches have some defects comparing with ours. On the one hand, they base on first-order Markov assumption, in which those servlets that customer will visit in future are only determined by servlets that customer are visiting currently. This model can not accurately reflect customer visit behaviors because it will bring some inexistent multiple steps visit behaviors. For example, as shown in Fig.3(e), there exist some transitions such as $s_3 \xrightarrow{2} s_2$, $s_5 \xrightarrow{2} s_1$ and $s_5 \xrightarrow{2} s_4$ in $TPG^2(1)$, but they never happened in three customer sessions. The inexistent transitions make visit patterns from Markov model are imprecise and may lead to invalid predictions. Our approach obtains pattern graphs from original customer sessions, so it overcomes the inaccuracy of Markov model. In addition, computation of $TPG^n(1)$ will bring more time-complexity.

4.3 Sequential Patterns Discovery Algorithm

Based on above definitions, we present a discovery algorithm discoverTPG to discover pattern graphs. For convenience of discussion, we define operation: session [k] = id_k, which gets the identity of servlet at position k in a session.

Algorithm 2 describes our sequential patterns discovery algorithm. First we use the function discoverTG to compute all transition graphs TG(k) (line 5). Secondly we compute $\sum_{m \in EndS} W_{TG(k)}(i, m)$ for different distances (line 10). At last, we get all transition probability graphs with different distances (line 13).

```
1.    S: the set of  all servlets in servlet container
2.    sessions: sessions represented clients behaviors
3.    d: maximal transition distances
4.    function discoverTPG(S, sessions, d){
5.     int[][][] TG = discoverTG(S, sessions, d);
6.      double[][][] TPG = new double[|S|][|S|][d];
7.      int[][] Temp = new int[|S|][d];
8.      for( int m = 0; m < d ; m ++){
9.        for( int j = 0; j < |S|; j ++ )
10.         Temp[i][m] += TG [i][j][m];
11.       for( int j = 0; j < |S|; j ++) {
12.         for( int i = 0; i < |S|; i ++ ){
13.           TPG [i][j][m] = TG [i][j][m]/ Temp[i][m];
14.         }
15.       }
16.     }
17.     return TPG;
18.   }
19.   function discoverTG(S, sessions, d){
20.     int[][][] TG = new int[|S|][|S|][d];
21.     for( each session in sessions ){
22.      for( int i = 0; i < d; i ++ ){
23.       for( int j = 0; j < session.length -d; j ++){
24.       TG [session[j]][session[i + d]][d]+=1;
25.         }
```

```
26.         }
27.     }
28.    return TG;
29.    }
```

Algorithm 2. discoverTPG Algorithm

If there are *N* sessions and the average length of them is *avglength*, the time-complexity of $O(N*d*avglength)$ is required for the function discoverTG. On the other hand, the function discoverTPG needs $O(|S|*|S|*d)$ time-complexity to compute all transition probability graphs. Therefore this discovery algorithm has a time-complexity of $O(max(N*d*avglength, |S|*|S|*d))$. While there are a large number of sessions and servlets in servlet container, this algorithm is time-consuming and may occupy a lots of system resources. Therefore, we should choose suitable time to run discovery algorithm to avoid the influence on the normal running of servlet container.

In addition, Berkhin [17] has confirmed that there are no associations between two servlets if transition distance is higher than 6. Therefore, we only concern the transitions with less 7 distance.

5 Replacement Cost Functions

In this section, we first briefly review some traditional replacement cost functions. Secondly the concept of predictive probability function will be presented. At last two replacement cost functions based predictive probability function are introduced.

5.1 Traditional Replacement Cost Functions

Traditional replacement cost functions concern some factors such as visit time, visit frequency, file size and file loading cost in web server and proxy server. The same factors can also be concerned in servlet container, but they have different meanings for servlet cache object. Some factors of scache are as follows:

- age: the last visit time of scache.
- frequency: the visit times of scache.
- size: the size of scache.response
- cost: the time spending on executing scache.s to generate scache.response with scache.values.

Definition 9. Above factors are called as **original factors** because they belong to servlet cache object (scache) itself. Some typical replacement cost functions of traditional cache replacement algorithms are listed in Table.1, which contain one or more original factors.

In Table.1, the factor age and frequency make the cache objects with minimal revisited possibility evicted from the cache; the factor size and cost make the cache objects with minimal replacement cost removed from the cache. These replacement cost functions only consider visit history of single servlet regardless of associations among

Table 1. The Factors of Cache Replacement Algorithms

	age	frequency	size	cost	replacement cost function
LRU	√				age
SIZE			√		size
LFU		√			frequency
GDSF	√	√	√	√	cost*frequency/size +age

servlets. Next, we will consider some features of servlet container and utilize aforementioned transition probability graph TPG(d) to design predictive probability function, which makes a supplement for traditional replacement cost functions.

5.2 Predictive Probability Function

Definition 10. Predictive probability function represents the probability that a cacheable servlet is revisited in near future, which is regarded as a supplementary factor for original factors and comes from associations among servlets. In this section we design two predictive probability functions as follows, where K represents quantity of scache objects in cache system.

- Session-based predictive probability function

$$rank_{session}(j, D) = \sum_{d=1}^{D} \sum_{i=1}^{|sessions|} TPG(d)[session[length], j, d] \tag{1}$$

- Scache-based predictive probability function

$$rank_{scache}(j, D) = \sum_{d=1}^{D} \sum_{i=1}^{K} TPG(d)[scache(i).s.id, j, d] \tag{2}$$

Equations (1) and (2) predict the probability that a cacheable servlet s_j is revisited in near future according to the different historic information. The function $\text{rank}_{\text{session}}$(j, D) bases on session information and reflects realtime customer behaviors. The function $\text{rank}_{\text{scache}}$(j, D) only considers the transitions starting from cacheable servlets and reflects the status of cache system, which maybe makes its prediction lesss accurate than equation (1), but less time-complexity of computation will be gained.

It needs the time-complexity $O(L \times D \times |sessions|)$, $O(L \times D \times K)$ to compute the prediction probability of all cacheable servlets through equations (1) and (2), respectively. Although predictive probability functions induce some computation cost, they can evict cache objects with minimal cost and reduce the times of cache replacement and redundant reexecution of cacheable servlets. In result, a higher cache hit ratio and better performance of servlet container are achieved by adoption of predictive probability functions.

If we define traditional cost functions and predictive probability functions as $\text{rank}_{\text{tranditional}}$(scache) and $\text{rank}_{\text{predictive}}$(scache) respectively, we can represent new replacement cost function as:

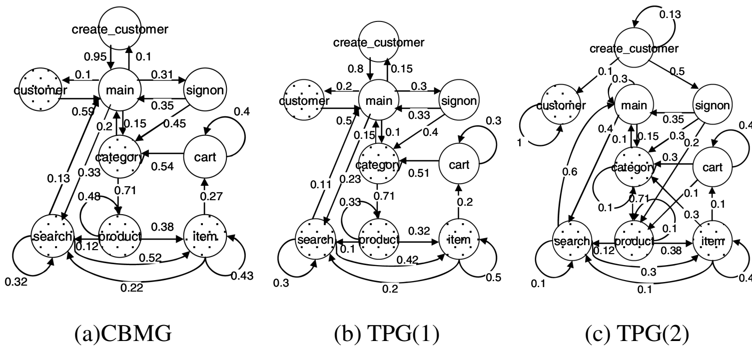
$$\text{rank}_{\text{new}}(\text{scache})=\text{rank}_{\text{traditional}}(\text{scache}) \times \text{rank}_{\text{predictive}}(\text{scache}). \tag{3}$$

In the section 6 we will confirm the effectiveness of new replacement cost function $\text{rank}_{\text{new}}(\text{scache})$ through evaluations.

6 Evaluation

The servlet cache model of servlet container and replacement cost functions have been implemented in the application server OnceAS [18] developed by Institute of Software, Chinese Academy of Sciences.

OnceAS application server runs on a PC with CPU of P4 2.8G, memory of 512M, and windows 2000 professional operating system. The simulating customers visit OnceAS through a 100M LAN. We adopt a Java EE blueprint program Pet Store Demo [19] as our web application. For experiment simulation, we add 10000 pets including 2000 cats, dogs, fish, birds and reptiles respectively into pet store as our simulation data.



(a)CBMG (b) TPG(1) (c) TPG(2)

Fig. 4. CBMG of Simulating Customers and Pattern Graphs

Daniel [20] proposes customer behavior model graph (CBMG) and gives different category CBMG for three kinds of customers to simulate the behaviors of real customers in TPC-W. We use Daniel's method to compute the CBMG of Pet Store for one category customer in Fig.4 (a) where white nodes denote non-cacheable servlets and grey nodes denote cacheable servlets. We regard Fig.4 (a) as the behaviors of simulated customers, every simulated customer visits different servlets continually according to the transition probability of Fig.4 (a).

Servlet cache system running in OnceAS has provided the solution on cache consistency, which will evict invalid cache objects in time. In addition, since the response of servlets search, categories, products and items are relatively stable unless

Table 2. The Cacheable Servlets of Pet Store

Servlet name	category	product	item	customer	search
Parameter	category_id	product_id	item_id	customer_id	keywords

we maintain the database of Pet Store, we define them as cacheable servlets. Their name and parameters are shown in Table.2.

The evaluation has been performed in two steps. First of all, 100 customers are simulated to visit Pet Store for six hours according to the CBMG in Fig.4 (a) so that we can collect the customer sessions and run discovery algorithm discoverTPG to achieve pattern graphs. In a second stage, we utilize the results from first stage to perform some experiments to confirm the efficiency of our approach.

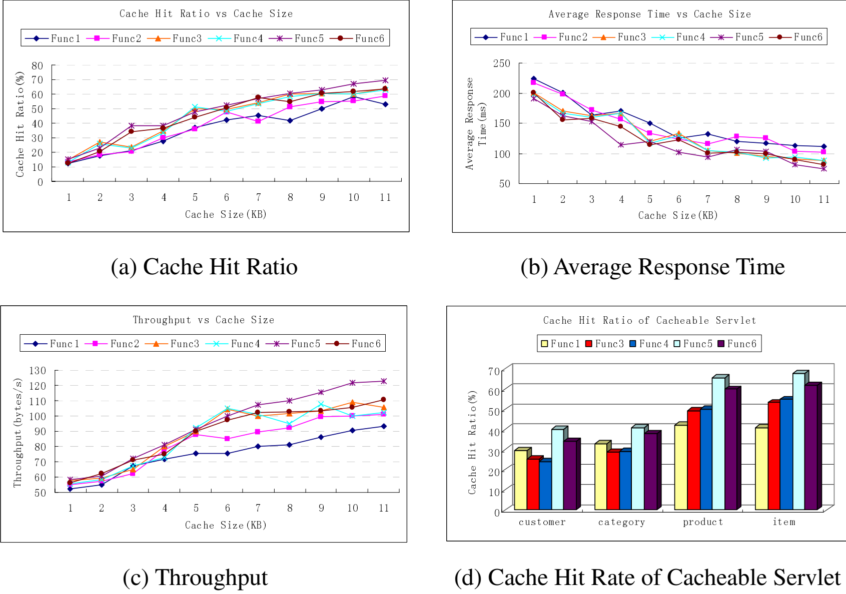
The results of first stage are presented in Fig.4 (b)(c). Obviously, TPG(1) is similar to Fig.4 (a), which has covered customer visit tendency. In addition, we can see that TPG(2) can discover some sequential patterns that can not be discovered by TPG(1), such as $customer \xrightarrow{2} customer$ and $item \xrightarrow{2} category$.

Experiment 1 is conducted with different cache sizes and replacement cost functions. We still simulate 100 customers with the behaviors described in Fig.4 (a) to visit Pet Store for two hours. The settings and results of experiment 1 are presented in Table 3 and Fig.5. (The function whose $rank_{predictive}$ =1 represents a traditional algorithm, and $mrank_{session}(j,1)$ and $mrank_{session}(j,2)$ represent the one step and two steps replacement cost functions from Markov model, respectively.) First, experiment 1 compares cache hit ratios of traditional LRU algorithm and sequential pattern-based cache replacement algorithms adopted equations (1) and (2) ,respectively. Secondly experiment 1 shows the influence on cache hit ratio from different transition distances. At last, experiment 1 also compares the prediction effect of our approach with Markov model at different distance.

Table 3. The Settings of Experiment 1

Cost function	$rank_{traditional}$	$rank_{predictive}$
Func1	age	1
Func2	age	$rank_{scache}(j,1)$
Func3	age	$rank_{session}(j,1)$
Func4	age	$mrank_{session}(j,1)$
Func5	age	$rank_{session}(j,2)$
Func6	age	$mrank_{session}(j,2)$

From Fig.5 (a) we can see that traditional LRU algorithm (Func1) has the worst cache hit ratio, only 52% when cache size is 1.2MB. Then, cache hit ratio has increased remarkably when predictive probability functions are applied. In addition, cache hit ratio increment gained from equation (1) is higher than that from equation (2) (Func3>Func2) just like what we expect in section 5. Although there exist several exceptions caused by additional computation, the cache hit ratio of Func5 is higher than that of Func3 at most time. It shows that the longer distance is considered in replacement cost function, the higher cache hit ration can be gained. At last, we can see that Markov model (Func4) has the almost same increment comparing to our approach (Func3) when transition distance is 1. However, our approach has the more attractive enhancement of cache hit ratio than Markov model when transition distance is 2, because the latter introduces some inexistent visit behaviors at this time.



(a) Cache Hit Ratio (b) Average Response Time

(c) Throughput (d) Cache Hit Rate of Cacheable Servlet

Fig. 5. The Result of Experiment 1

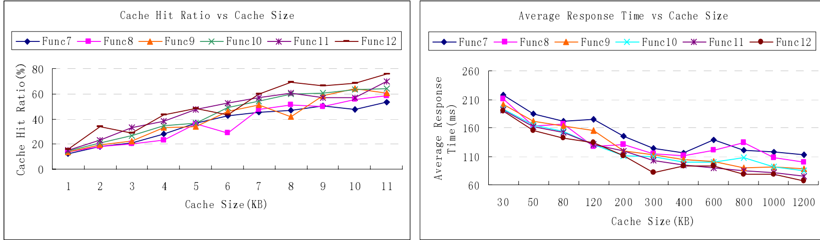
Fig.5(b) and (c) show that our approach also gains much less average response time and higher throughput of servlet container than traditional algorithms and those based on Markov mode. However, we notice that there exists an abnormal case where the cache hit ratio of Func3 is lower than that of Func1 (traditional LRU) for servlets customer and category, as shown in Fig.5 (d). The reason is that TPG(1) in Fig.4 (b) misses some customer behaviors, which makes cached responses of servlets *customer* and *category* are evicted improperly and impairs cache hit ratio. But we are glad to see that Func5 (TPG(2)) has erased these abnormalities successfully.

Experiment 2 is also conducted with different cache sizes, which compares cache hit ratio of original LRU, LFU and GDSF algorithms and the corresponding algorithms using predictive probability function $rank_{session}(j,2)$ respectively. Table.4 and Fig.6 show the settings and the result of experiment 2.

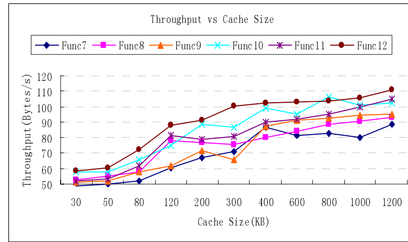
Table 4. The Cost Functions of Experiment 2

Cost function	$rank_{traditional}$	$rank_{predictive}$
Func7	age	1
Func8	age	$rank_{session}(j,2)$
Func9	frequency	1
Func10	frequency	$rank_{session}(j,2)$
Func11	cost*frequency/size +age	1
Func12	cost*frequency/size +age	$rank_{session}(j,2)$

From Fig.6(a) we can observe that cache hit ratios of three cache replacement algorithms have been improved after adopting predictive probability function ($rank_{session}(j,2)$). Although there are some inconsistencies in Fig.6 (a) because the computation of $rank_{session}(j,2)$ consumes some system resources such as CPU and memory, Fig.6 (b) and (c) still show that predictive probability functions have improved the performance of servlet container in response time and throughput.



(a) Cache Hit Ratio (b) Average Response Time



(c) Throughput

Fig. 6. The Result of Experiment 2

7 Conclusion

To resolve the servlet cache replacement problem and improve the performance of servlet container, in this paper, we first present pattern graphs to describe business associations among servlets as sequential patterns. Secondly we present our discovery algorithm to discover the pattern graphs and propose two predictive probability functions to mend traditional replacement cost functions according to pattern graphs. Finally, evaluation shows that our approach can effectively improve cache hit ratio and the performance of servlet container.

However, our approach still has some limitations. Firstly, the discovery algorithm discoverTPG has a higher time-complexity ($O(\max(N \times d \times avglength, |S| \times |S| \times d))$), which makes us have to choose suitable time to run discovery algorithm. Secondly pattern graphs maybe become stale with the change of customer visit behaviors so that we have to rerun discovery algorithm to capture these changes and adjust our predictive probability functions appropriately. In addition, to apply this approach into other environments such as EJB and Web service, it is need to be extended to solve the consistency of cached objects. We will solve above issues in the future.

Acknowledgments. The work was supported partially by the National Natural Science Foundation of China under Grant No. 60573126; the National Basic Research Program of China (973) under Grant No. 2002CB312005; the National High-Tech R&D Plan of China (863) under Grant No.2006AA01Z19B; the National Key Technology R&D Program of China under Grant No. 2006BAH02A01.

References

1. Coward, D.: Java TM Servlet Specification Version 2.4 Sun Microsystems Inc. 2003-11-24 http://jcp.org/aboutJava/ community/ process/final/jsr154/index.html
2. Christopher, L.: Servlet Performance Report: Comparing The Performance of J2EE Servers http://www.webperformanceinc.com/library/reports/ServletReport/index.html
3. Bonchi, F., Giannotti, F., Gozzi, C., Manco, G., Nanni, M., Pedreschi, D., Renso, C., Ruggieri, S.: Web Log Data Warehousing and Mining for Intelligent Web Caching. Data & Knowledge Engineer 39(2), 165–189 (2001)
4. Li, K., Shen, H., Tajima, K.: Cache Design for Transcoding Proxy Caching. In: Jin, H., Gao, G.R., Xu, Z., Chen, H. (eds.) NPC 2004. LNCS, vol. 3222, pp. 187–194. Springer, Heidelberg (2004)
5. Turner, D.: Web Page Caching in Java Web Applications. In: Proc. of International Conference on Information Technology Coding and Computing, Las Vegas (2005)
6. Shupp, R., Andy, C., Chuck, F.: Web Sphere Dynamic Cache: Improving J2EE application performance. IBM System Journal 43(2), 351–370 (2004)
7. Podlipnig, S., Böszörmenyi, L.: A survey of web cache replacement strategies. ACM Computing Surveys 35(4), 374–398 (2003)
8. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. In: Proc. of the Fifth Int. Conference on Extending Database Technology, Avignon, France: Springer-Verlag Berlin and Heidelberg GmbH, pp.18–32 (1996)
9. Cooley, R., Mobasher, B., Srivastava, J.: Web mining: information and pattern discovery on the World Wide Web. In: Proc. of the 9th IEEE International Conference on Tools with Artificial Intelligence, Newport Beach, pp. 558–567. IEEE Computer Society, Los Alamitos, CA, USA (1997)
10. Garofalakis, M., Rastogi, R., Shim, K.: Spirit, Sequential pattern mining with regular expression constraints. In: Proc. of the ICVLD, pp. 223–234. Morgan Kaufmann Publishers, Edinburgh (1999)
11. Han, J., Pei, J., Mortazavi-Asl, B.: PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: ICDE01, pp. 215–224. Springer, Heidelberg (2001)
12. Cherkasova, L.: Improving WWW Proxy Performance with Greedy-Dual-Size Frequency Caching Policy, HP Labs: Computer Systems Laboratory: HPL-98-69R1 (1998)
13. Abrams, M., Stanbridge, C., Abdulla, G., Williams, S.: Caching Proxies: Limitation and Potentials. In: Proc. of the 4th WWW Conference, pp. 119–133. O'Reilly, Boston (1995)
14. Yang, Q., Zhang, H.: Web-log mining for predictive web caching. IEEE Transactions on Knowledge and Data Engineering 15(4), 1050–1054 (2003)
15. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Yu, P.S., Chen, A.S.P. (eds.) ICDE. In: Proc. of the 11th ICDE, pp. 3–14. IEEE Computer Society Press, Washington DC (1995)
16. Yin-Fu, H., hao Min, J.: Mining Web Logs to Improve Hit Ratios of Prefetching and Caching. In: Proc. of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence, Compiegne France, Sep. 19-22, pp. 577–580 (2005)

17. Pavel, B., Becher, D.J., Randall, D.J.: Interactive Path Analysis of Web Site Traffic. In: Proc. of ACM SIGKDD Int. KDD01, San Francisco, CA, pp. 419–441. ACM Press, New York (2001)
18. Huang, T., Chen, N.J., Wei, J., Zhang, W.B., Zhang, Y.: OnceAS/Q: A QoS-enabled Web application server. Journal of Software 15(12), 1787–1799 (2004)
19. JavaTM Pet Store Demo, http://java.sun.com/developer/ releases/petstore/
20. Daniel Menascé, TPC-W:A Benchmark for E-commerce IEEE Internet Computing, 6(3), pp. 83–87 (2002)
21. Sarukkai, R.: Link Prediction and Path Analysis Using Markov Chains. In: Proc. of the 9th Intl. World Wide Web Conf. Amsterdam, May (2000)
22. Zhu, J., Hong, J., Hughes, J.: Using Markov Chains for Link Prediction in Adaptive Web Sites. In: Proc. of Software, Belfast, Northern Ireland, pp. 60–73 (2002)
23. Web Characterization Activity. http://www.w3.org/WCA