A New Distributed Caching Replacement Strategy

Bing Zhang, Hao Wu
College of Computer and Software
Shenzhen University
Shenzhen, China
zhangb@szu.edu.cn

***Abstract*— A new distributed caching replacement strategy is presented. The nodes in the distributed system concerned are connected according to topology of Petersen Graph. The strategy takes into account the system global characteristics in replacing cache objects. A mechanism called "dump and clear" is designed to solve the locality problem of most of the current replacement strategies. Experiments on cache hit rate and byte hit rate are conducted and results show that the presented distributed caching replacement strategy outperforms other similar algorithms.**

Keywords-: distributed caching protocol; cache replacement algorithm; Petersen Graph.

I. INTRODUCTION

Since beginning of 21st century, researches on large-scale parallel and distributed computing methodologies and their industrial applications have achieved great success, which not only boosts new applications of integration technology of parallel and distributed systems, but also stimulates revolution on their design theory and application infrastructures. As the frontier of information science, Internet has been carrying more and more information and serving more and more user requests. The access speed of Internet has never met with satisfaction of what user actually requires. Under such circumstances, it becomes more and more important and indeed has received more and more attentions from both academic world and industrial section to design new web caching protocols.

Replacement algorithm plays a key role in caching performance. Most caching replacement algorithms are designed to be used on standalone computers. While in large-scale distributed computing systems, traditional caching replacement algorithms can not take into account the fact that the data to be cached is distributed among the system. Therefore, it is of great research significance and high application value to design a caching replacement algorithm that is suitable for large-scale distributed systems.

II. RELATED WORK

Most of the current caching schemes emphasize on increasing nodes involved in caching and on designing coordination mechanisms for the nodes to increase caching capacity and performance. Malpani[1] presented a distributed caching protocol based on IP multicast; Wessels[2] presented a hierarchical caching protocol called ICP(Internet Cache Protocol); Some distributed caching protocols, like CARP[3], Squirrel[4] and Kache[5] etc, are based on hash functions.

Korupolu[6] converted the cooperative cache placement problem into a problem of the minimum cost flow, and presented an optimal solution to optimize the access latency between nodes in a distributed system that is based on hierarchical topology. Bhattacharjee[7] proposed a cache placement and replacement strategy called "Modulo", which is based on self-organizing networks, to reduce network congestion and access latency. Li[8] presented an optimal placement strategy based on a solution to the cost graph, which can effectively reduce cache redundancy and access delay through placing cache copies among nodes of the distributed system reasonably. However, most studies focused on decreasing redundancy of cache copies and reducing cache access delay (hops). The caching systems that studied are mostly based on overlapped networks, without considering the characteristics of the underlying physical network topology.

Current replacement algorithms, such as LRU(Least Recently Used), GDSF[9] and Hybrid[10], also have a common shortcoming that only web objects of local cache are replaced when the cache overflows, without putting cache objects of the other nodes in the system into the replacement set to make replacement decision globally, hence objects of high utility value are often replaced before that of the low utility value, and the overall system performance is degraded.

This paper presents a new distributed replacement strategy based on a network topology of dense graphs such as Petersen Graph with the purpose to solve the above mentioned problem and to increase cache space utilization and cache hit rate.

Petersen Graph[11], as shown in Figure 1, is a graph that is the most dense in the sense that it can accommodate the largest number of nodes among graphs of diameter *2* and degree *3*. Zhang[12] presented a parallel sorting algorithm that runs on a system of *10* nodes connected as Petersen's Graph. In 2004, Zhang[13] showed that the algorithm can be implemented in hardware using FPGA and can be used to find the minimum or maximum in single step. In 2007, Zhang[14] showed that the algorithm can be extended into a network connected according to Singleton Graph[15] which is also a special type of dense graph. These results show that using dense graph as the topology, computing load can be evenly distributed among the nodes in the system. Every node executes the same algorithm, which reduces the

978-1-61284-486-2/11/$26.00 ©2011 IEEE

communication and synchronization cost, hence a high parallelism can be achieved. Apart from using Petersen's Graph as the topology, the presented caching replacement strategy also designed a mechanism called "dump and clear".

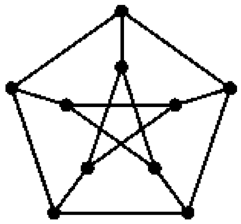


Figure 1. Petersen graph

III. THE NEW CACHING STRATEGY

When a new object is introduced into the cache of a node and the cache overflows, replacement has to take place. Assume the k objects which are going to be replaced by the new object are in set $O = \{obj_1, obj_2, ..., obj_k\}$. Given V as the set of neighbor nodes of the current node, the presented "dump and clear" strategy can be described as:

```
Procedure DUMP_CLEAR
begin
    for i :=1 to k do
    begin
        y := min_element(O);
        acceptable_nodes := lookup(V， y) ;
        lowest_cost_node := min_node(acceptable_nodes) ;
        if  lowest_cost_node <> Φ then
            (* let lowest_cost_node cache object y *)
            dump (lowest_cost_node,y) ;
        O.remove(y);
    end
end
```

Where *min_element(O)* returns the object y that has the minimum utility value in O. *lookup(V, y)* calls the *GET_H(y)* function in its adjacent nodes V and returns the nodes that can cache object y. *min_node()* returns the node with the minimum cost of caching object y. The presented algorithm to determine the dumping cost can be described as:

```
Function GET_H
begin
    c := CALC_H(y);
    if c <=  Δ then return c;
    if step > 1 then
    begin
        step := step -1;
        foreach e in [V - {parent}] do
            if c > e.GET_H (y) then c := e. GET_H (y) ;
    end
    return c;
end
```

Where *CALC_H(y)* returns the calculated cost of the current node to cache object y, which is denoted as c. If c is less than a threshold Δ, c is the value returned by function *GET_H*. Otherwise, the information about object y is sent to the neighbor of the current node (denoted now as *parent*), where the cost of caching y is calculated. This process iterates until *step* equals one, where *step* is the largest distance between any two nodes. In this way, the minimum cost of caching object y among nodes of a sub-tree is returned as the value of *GET_H*.

The algorithm of calculating the cost of caching object y is described as follows:

```
Function CALC_H
begin
    free_space := get_free_space();
    if free_space >= y. size then return 0.0;
    (* If size of object y is greater than that of the cache, then
return ∞ *)
    if get_space()  <  y.size then return ∞;
    h := 0.0;
    elem := priority_list.get_first() ;
    (* calculating the upper limit to space and cost required
    for caching object y *)
    while elem <> Φ do
    begin
            free_space := free_space + elem.size;
            h := h + elem.h;
            if free_space >= y.size then break;
            elem := priority_list.get_next(elem);
    end
    usable_space := free_space - y.size;
    (* calculating the lower limit to space and cost required
    for caching y *)
    foreach e : (elem, priority_list.get_first()] do
        if e.size < = usable_space then
        begin
            h := h - elem.h;
            usable_space := usable_space - e.size;
        end
    if h >= y.h then return ∞ else return h;
end
```

Where *free_space* is the size of unused cache space, and *priority_list* is a two-way linked list of cached objects in ascending order according to utility value, each of its elements stores information about a cached object.

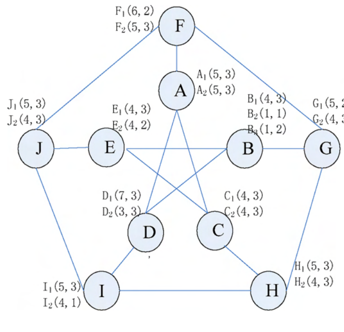
IV. AN EXAMPLE

This section presents an example showing how the dump and clear strategy works. Assuming the capacity of a single cache is "6", and the initial state of all caches of the 10 nodes connected according to topology of Petersen Graph is shown in Figure 2(a). The parameter *step* now equals two and the threshold Δ is set as zero. Let $u(v, w)$ represent an object u with utility value of v and size w. When F gets a new object $x(6,3)$ from the server, three dumping operations will be taken place:

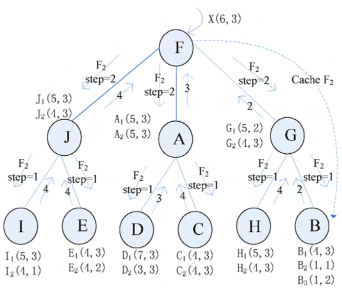
1. After receiving x, F decides to replace F_2, so it sends information about F_2 to A, G and J. A, G and J calculates in parallel the cost of caching object F_2. The results are ∞, 4 and 4. Since the results are all greater than $\varDelta$, the node B, H, C, D, E and I (F now is the parent and will not be involved) will calculate in parallel the cost of caching F_2, which are 2, 4, 4, 3, 4 and 4 respectively. Now *step* number of iterations has reached, and the searching stops. B, D and E are the three nodes that have the minimum cost of caching object F_2 determined in parallel from three branches of F: G, A and J. Next, node B with the minimum cost of 2 is chosen as the destination node, and object F_2 is sent to B via G, as is shown in Figure 2 (b).

2. As shown in Figure 2(c), after receiving F_2, B decides to replace B_3 and B_2, so it first sends information about B_3 to D, E and G, to calculate their costs of caching B_3, using the same algorithm as mention in operation 1. The results are 0, ∞ and ∞ respectively. Then, I is chosen as the destination node, and object B_3 is sent to I via D. Since enough space is available in the cache of node I, B_3 is inserted directly into this cache without making any replacement.

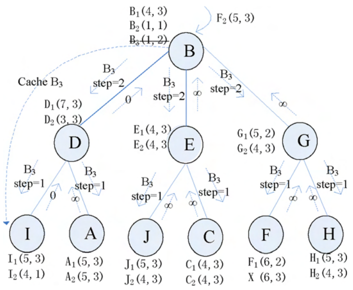
3. As shown in Figure 2(d), to replace B_2, B first sends information about B_2 to D, E and G to calculate in parallel the cost of caching object B_2. The results are ∞, 0 and 0. Since the results of E and G equal to 0 which is less than $\varDelta$, E and G will not send the information to their child nodes anymore, and 0 are returned by E and G to B. Then B chooses E or G by random (assumed to be G) as the destination node without waiting for the response from node D, and sends B_2 to G. Since enough space is available in the cache of node G, B_2 is inserted directly into this cache.



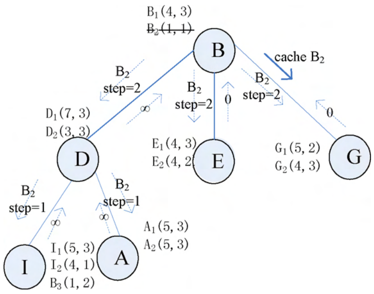
(a) The system initial state



(b) Operation one



(c) Operation two



(d) Operation three

Figure 2. An example illustrating the presented dump and clear strategy

V. EXPERIMENTS

In order to verify the feasibility and effectiveness of the presented "dump and clear" replacement strategy, experiments on hit rate and byte hit rate which are two important criteria to evaluate caching protocols are conducted. Experiments focus on performance comparison among the presented dump and clear replacement strategy, LRU and GDSF algorithms. Data used in the experiments is taken from trace records from DEC's proxy cache server[16], which contains a total of *323,890* web access requests. A program that emulates *10* nodes connected according to topology of Petersen's graph is written and runs on an Intel Pentium D CPU 3.20GHz, 2GB RAM computer. In calculating dumping cost, the parameter *step* is set to 2 and the threshold $\varDelta$ is set to half of utility value of the current dumped object. Each node calculates $H(f)$ according to (1), which is the utility value of each object and is used as an important criteria to determine whether the object should be cached or not.

$$H(f) = L + F(f) * /\log_2[S(f)] \qquad (1)$$

In Equation (1), $F(f)$ is reference frequency for the object f, $S(f)$ is size of object f, and L is inflation factor.

A two-way priority list is used to maintain cached objects. The elements of the list are sorted in ascending order according to utility value. In addition, each node maintains a hash map in order to quickly find the object information, where the hash key is the object identifier and the value is the structure reference. During simulation, the request data is sent randomly in sequence to every node by transmitter.

Figure 3 and Figure 4 show the experiment results on comparing the presented dump and clear replacement strategy, LRU and GDSF in terms of hit rate and byte hit ratio.

Results shown in Figure 3 indicate the effect of cache size on hit rate performance of the three schemes. Results shown in Figure 4 illustrate the effect of cache size on byte hit rate performance of the three schemes. As far as hit rate is concerned, the presented scheme is always better than the other two schemes, especially when the cache size is relatively small. When cache size increases, the hit rate performance gap between the three schemes becomes narrower. This is because smaller cache space will be overflowed more often, and replacement operations will take place more frequently, hence the advantage of the presented scheme on replacing objects with globally smaller utility values is more obviously shown.

In terms of byte hit rate, as shown in Figure 4, dump and clear scheme is better than the other two schemes. This is because the presented dumping operation makes high utility objects of large size have more chance to remain in the cache and be reloaded into the cache when they are removed temporarily from the cache. Besides, the presented scheme tends to accommodate more objects of smaller size. These two factors result in more effective cache space utilization and improvement on byte hit rate.

VI. CONCLUSION

This paper presents a new distributed cache replacement strategy called “dump and clear” to solve the cache replacement problem when cache overflows. Unlike other cache replacement algorithms, the presented cache replacement strategy has the characteristics that replacement decision is made on a global basis, hence objects with globally smaller utility value can be replaced before objects of higher value, making cache replacement more rational and resulting in more effective use of limited cache space. Experiments on hit rate and byte hit rate, which are two important factors evaluating caching performance are conducted. Simulation experiment results show that the presented cache replacement strategy outperforms LRU and GDSF algorithms, and is suitable for high-speed distributed cache system with widely distributed nodes and limited cache space. Experiments on introducing distributed sorting into the caching scheme will be conducted in future work.

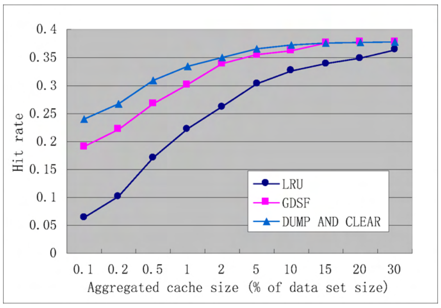


Figure 3. Experimental result on hit rate

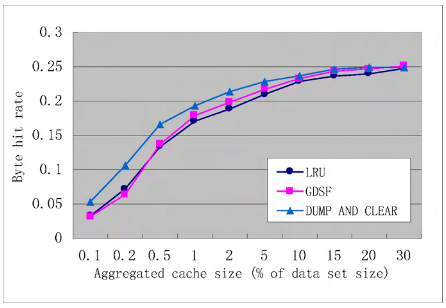


Figure 4. Experimental result on byte hit rate

REFERENCES

[1] R. Malpani, J. Lorch, and D. Berger, Making World Wide Web caching servers cooperate, Proceedings of the 4th International WWW Conference, Boston, MA, Dec. 1995.

[2] D. Wessels and K. Claffy, ICP and the Squid Web cache [J] .IEEE Journal on Selected Areas in Communication, 1998, 16 (3) :345 - 357.

[3] V. Valloppillil and K. W. Ross, Cache array routing protocol v1.0, Internet Draft _ draft- vinod-carp-v1-03.txt.

[4] Sitaram Iyer, Antony Rowstron, and Peter Druschel, Squirrel: a decentralized peer-to-peer web cache, Proceedings of the twenty-first annual symposium on Principles of distributed computing, July 21-24, 2002.

[5] P. Linga, I. Gupta, and K. Birman, Kache: Peer-to-peer web caching using Kelips. ACM Transaction on Information System, 2003, 5(9):1-29.

[6] Korupolu MR, Dahlin M. Coordinated placement and replacement for large-scale distributed caches. In: Proc. of the 1999 IEEE Workshop on Internet Applications (WIAPP’99). San Jose: IEEE Computer Society, 1999. 62.

[7] Bhattacharjee S, Calvert KL, Zegura EW. Self-Organizing wide-area network caches. In: Proc. of the 17th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM’98). San Francisco: IEEE Computer Society, 1998. 600−608.

[8] Wenzhong Li, Daoxu Chen, Sanglu Lu, Graph-Based Optimal Cache Deployment Algorithm for Distributed Caching Systems, Journal of Software, Vol.21, No.7, July 2010, pp.1524−1535.

[9] L. Cherkasova, Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy, Technical Report HPL-98-69R1, Hewlett-Packard Laboratories, November 1998.

[10] R. P. Wooster and M. Abrams, Proxy caching that estimates page load delays, Proceedings of the 6th International WWW Conference, April 1997.

[11] D.A.Holton and J.Sheehan,The Petersen Graph,Cambridge University Press,New York,1993.

[12] Bing Zhang and Mingcheng Zhu, A Parallel Sorting Scheme Based on Interconnectec Processor networks,Computer Applications, 2003, 23(7):9-12.

[13] Bing Zhang, Yuan Xu, Mingcheng Zhu, A parallel sorting algorithm and its hardware implementation based on FPGA, Journal of Information and Computational Science, 2004,1(3):417-422.

[14] B. Zhang, J.G. Shi, M.C. Zhu. A Parallel Sorting Scheme of 50 Numbers and its Hardware Implementation on FPGA [C]. HUBEI, CHINA: Proceedings of Distributed Computing and Algorithms for Business, Engineering, and Sciences,2007:1213-1216.

[15] A. J. Hoffman, R. R. Singleton, On Moore graphs with diameters 2 and 3. IBM Journal of Research and Development, 64:15-21(1960).

[16] Digital Equipment Cooperation, Digital's Web Proxy Traces ftp://ftp.digital.com/pub/DEC/traces/proxy/webtrace.