A Multi-Resource Load Balancing Algorithm for Cloud Cache Systems

Yu Jia[1, 2], Ivan Brondino[1], Ricardo Jiménez Peris[1], Marta Patiño Martínez[1], Dianfu Ma[2]

[1]The Distributed Systems Lab, Universidad Politécnica de Madrid, Spain
[2]Advanced Computing Technology Institute, Beihang University, China
[1]{yjia, rjimenez, mpatino, ibrondino}@fi.upm.es
[2]{jiayu, madianfu}@act.buaa.edu.cn

ABSTRACT

With the advent of cloud computing model, distributed caches have become the cornerstone for building scalable applications. Popular systems like Facebook [1] or Twitter use Memcached [5], a highly scalable distributed object cache, to speed up applications by avoiding database accesses. Distributed object caches assign objects to cache instances based on a hashing function, and objects are not moved from a cache instance to another unless more instances are added to the cache and objects are redistributed. This may lead to situations where some cache instances are overloaded when some of the objects they store are frequently accessed, while other cache instances are less frequently used.

In this paper we propose a multi-resource load balancing algorithm for distributed cache systems. The algorithm aims at balancing both CPU and Memory resources among cache instances by redistributing stored data. Considering the possible conflict of balancing multiple resources at the same time, we give CPU and Memory resources weighted priorities based on the runtime load distributions. A scarcer resource is given a higher weight than a less scarce resource when load balancing. The system imbalance degree is evaluated based on monitoring information, and the utility load of a node, a unit for resource consumption. Besides, since continuous rebalance of the system may affect the QoS of applications utilizing the cache system, our data selection policy ensures that each data migration minimizes the system imbalance degree and hence, the total reconfiguration cost can be minimized. An extensive simulation is conducted to compare our policy with other policies. Our policy shows a significant improvement in time efficiency and decrease in reconfiguration cost.

1. INTRODUCTION

Platform as a Service (PaaS) has been proposed in the context of cloud computing as a solution to deliver computing platforms that facilitate application deployment by reducing the cost and complexity of managing underlying hardware and software for users [2]. One of the most significant requirements for PaaS providers is to deal with large number of users and considerable amounts of data. Handling large amount of data is also a big issue to tackle in Infrastructure as a Service (IaaS) and Software as a Service (SaaS). According to Forrester, Facebook employs a petabyte of storage to manage 40 billion photos of its millions of users. Some other web giants, such as Amazon and eBay, face similar scale [3]. In order to cope with this scale problem, multi-tier architectures become an important building block for many applications in the cloud [4]. A fast, scalable, and fault-tolerant data-access layer, known as cache system (platform) or in-memory data grid is added on top of the database to provide a fast access to data, and meet the scalability requirement of cloud environments. For example, Google App Engine uses memchached [5] for caching popular pages, frequently updated data, etc.[6].

Application workloads in the cloud are dynamic and change over time, thus, data stored in a cache platform are retrieved and updated in a varying manner, which means workloads become imbalanced over cache nodes and can be skewed to small fraction of data partitions [7]. Large number of concurrent data accesses to certain partitions of cached data (hotspots) may cause a high consumption of resources such as CPU and Memory at the cache node where these data are stored in, possibly leading to system bottlenecks [8]. In order to eliminate these bottlenecks, the system must be monitored and provide mechanisms for dynamically balancing the load. Figure 1 shows a possible scenario where our algorithm may be applied in. In the figure there are eight cache nodes in a single data center. Each one stores some objects that are grouped into *"buckets"*. A bucket is the unit of data migrated between cache nodes. The figure shows two overloaded cache nodes in red and dashed lines link the nodes involved in the data migration.

In this paper, we propose a dynamic load balancing strategy to balance multiple resources (namely, CPU and Memory) among

cache nodes in a single data center at runtime. Our main contributions are:

(1) A load balancing algorithm for redistributing stored data among cache nodes and hence, mitigating the imbalance degree. Since optimizing multiple resources at the same time may lead to contradicting goals, our algorithm firstly identifies which are the scarcer resources based on the runtime load distributions, and then assigns resources different priorities based on this diagnosis. As a result of the load balancing algorithm, some data partitions must be moved from one cache node to another to rebalance the load. This data migration incurs in extra consumption of resources such as CPU and network bandwidth, thus, reconfiguration cost is considered by the load balancing algorithm. Our proposed greedy algorithm seeks a local optimal solution to minimize the system imbalance degree in each step, a *nearest-neighbor-search* based policy is proposed to select the bucket to be migrated. Besides, since we avoid linear scanning to find this bucket, the time efficiency is improved compared with some other approaches.

(2) A detailed evaluation of the proposed algorithm is presented.

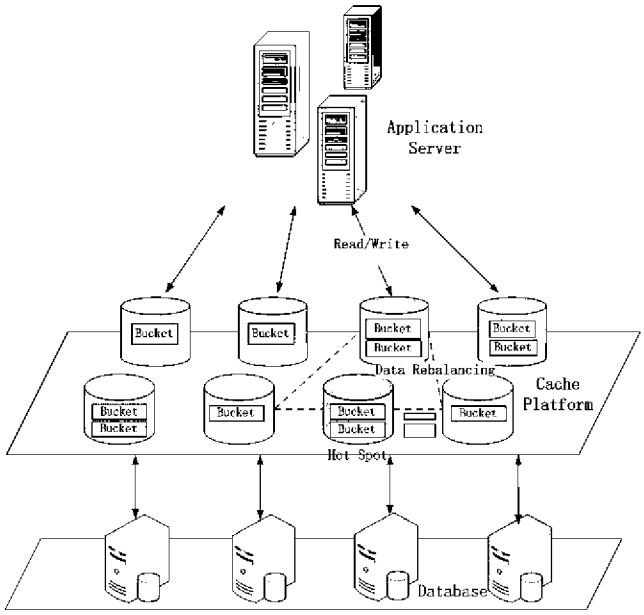


Figure 1. Scenario of Load Balancing in Cache Platform

The rest of the paper is organized as follows: related work is presented in Section 2. Section 3 provides some necessary definitions and goals. Section 4 illustrates the load balancing algorithm. Section 5 evaluates the performance of the load balancing algorithm and compares it with algorithms using other data selection policies. Conclusions and future work are presented in Section 6.

2. RELATED WORK

The search of optimal load balancing of multiple resources is a variant of bin-packing problem, namely multidimensional-vector packing problem, which is proved to be NP-hard [9]. The problem consists in packing a certain number of m-dimensional items into a minimum number of m-dimensional bins with independent capacity in each of the m dimensions [10]. In the context of load balancing for a distributed cache, a node can be treated as a bin with multidimensional capacities, each of which should be a value close to the average utilization of a resource in the cache cluster. Each data bucket (set of objects and migration unit) is an item to be packed with multiple dimensional weights, each of which should be the usage of a certain resource. Load balancing of multiple resources can be regarded as packing all the data buckets into the available nodes while ensuring that all the nodes are as "full" as possible. Papers [9][10][11][12] propose algorithms for multidimensional vector problems. However, all these algorithms obtain the approximate optimal solution off-line. Off-line approaches compute new configuration without considering the current configuration, which may cause a big change to the previous configuration, resulting in a very high reconfiguration cost.

Compared with off-line approaches, on-line load balancing computes a new configuration taking into account the current configuration of the system and tries to minimize the reconfiguration cost. Paper [13] describes how to eliminate bottlenecks in a P2P system by redistributing data among nodes. The algorithm uses an "extract and reallocate" pattern, it firstly extracts some lightest loaded data buckets from all nodes so that the load of each node is below a given threshold ordered from heaviest to lightest, it then reallocates these buckets back to system using a best-fit strategy. Greedy and evolutionary algorithms are proposed to mitigate the imbalance as well as minimize reconfiguration cost in [14]. Greedy algorithms for load balancing have also been studied in the context of data streaming and replicated databases [15][16]. In these algorithms, a bucket is always moved from the heaviest loaded node to the lightest one. The heaviest bucket that will decrease the system imbalance will be picked. Paper [7] has proposed a strategy to eliminate hot-spots as well as to minimize the reconfiguration cost for a large scale system, such as the Hotmail server cluster. However, all of these studies focus on balancing one single type of load, CPU for instance. Balancing only one resource may inadvertently produce bottlenecks in other resources.

Multi-resource load balancing has been widely studied in the context of job scheduling where tasks (jobs) are migrated. In such scenarios, each task being executed on a server has different requirements for multiple resources, such as CPU, Memory and bandwidth. Backfill Lowest (BL) and Backfill Balance (BB) [17] are two popular policies for job selection. Once the sender node and receiver node have been picked, BL determines which resource is most available in the receiver node, and then migrates the job that consumes most of that resource in sender node. BB migrates the job which can minimize the (maximum load/average load) measure for the receiving server. A Market Mechanism (MM) policy is presented in [18] to balance multiple resources among nodes with heterogeneous resource capacities. MM uses a pricing model($\sum_1^K(J_i * L_i)/\sum_1^K J_i$) to calculate the cost per resource of a job. J_i denotes the job's consumption of resource i, L_i is the load of resource i in receiver node before job migration, K is the number of resource types. The job with the lowest cost is selected. According to [18], MM improves both BL and BB, and it is more suitable for load balancing in heterogeneous systems. However, all of these three algorithms do not take the reconfiguration cost into account. Moreover, all of them need to find the target jobs using iteration among all jobs. Since the amount of data buckets in a cache node can be much higher than amount of jobs in a server, thus, iterating all the possible candidates will limit the efficiency of load balancing in a cache system.

3. PROBLEM FORMULATION

In cache systems, data are stored in memory due to its lower latency than persistent storage. The memory usage among cache nodes can be different because buckets may have different sizes (objects of different size) and the number of stored buckets among cache nodes may also be different. Besides, the memory usage of a single node changes dynamically. On the other hand, CPU usage among cache nodes can also be different since applications may have different access patterns to the data. Considering that both CPU and Memory are limited in cache systems, the load balancing algorithm must strive to balance both resources to eliminate hot spots. In this paper, we aim to achieve the following goals:

(1) Minimize the load imbalance for multiple resources among cache nodes but with different priorities based on the runtime load diagnose. Specifically, we consider CPU and Memory resources.

(2) Minimize the system reconfiguration cost; here we consider the number of data buckets to be migrated.

Table 1. Definitions

Definition	Description
$N = \{n_1, n_2, \dots, n_k\}$	Set of cache nodes, k is the number of cache nodes in system.
$Q = \{q_1, q_2, \dots q_i, \dots q_k\}$	Number of buckets in each node. q_i is the number of buckets in node n_i.
$B_i = \{b_{i1}, b_{i2}, \dots b_{iq_i}\}$	Set of buckets in node n_i.
$NL = \{(c_1, m_1), (c_2, m_2) \dots (c_k, m_k)\}$	Resource usage at each node, (c_i, m_i) is the average (CPU, Memory) resource usage in % at node n_i in a time interval.
$BL_i = \{(r_{i1}, s_{i1}) \dots (r_{iq_i}, s_{iq_i})\}$	Statistic info of each bucket in node n_i. (r_{iq_i}, s_{iq_i}) denotes the average data request rate and size of bucket b_{iq_i}, respectively.
$NL^* = \{l_1, l_2, \dots, l_k\}$	Utility load of all nodes. l_i is the average utility load of node n_i in a time interval.
$BL_i^* = \{(\upsilon_{i1}, \omega_{i1}) \dots (\upsilon_{iq_i}, \omega_{iq_i})\}$	Load of each bucket in node n_i. $(\upsilon_{iq_i}, \omega_{iq_i})$ represents the average (CPU, Memory) resource consumption (%) of bucket b_{iq_i} in a time interval.

We assume that all cache nodes are homogeneous with equal capacities of CPU and Memory. Table 1 shows some required definitions for later discussions and algorithm design. The definitions cover several sets, including the number of cache nodes N, and data buckets $\{B_1, B_2, \dots, B_k\}$. It also introduces the load information we will use for load balancing, such as resource usage at each node, denoted by NL, and some statistic information about buckets in a time interval, denoted by $\{BL_1, BL_2, \dots, BL_k\}$. In order to get these values, a monitoring mechanism periodically reports the resource usage at each node, as well as data request rate and size of each bucket. We will discuss sets NL^* and $\{BL_1^*, BL_2^*, \dots, BL_k^*\}$ (BL_i^* is the buckets load set at node n_i) later.

The load of each bucket is defined by its storage size and its associated CPU consumption. We use the request rate of a bucket to calculate its CPU consumption. The (CPU, Memory) load of a bucket b_{ia} is normalized as:

$$(\upsilon_{ia}, \omega_{ia}) = (c_i * (r_{ia} / \sum_{j=1}^{q_i} r_{ij}), m_i * (s_{ia} / \sum_{j=1}^{q_i} s_{ij}))$$

We have defined (c_i, m_i) as the (CPU, Memory) usage of a cache node. However, a measure is necessary to score a node's *overall load (utility load)*. This will be helpful for us to choose a node pair for data migration. We assume that the CPU and Memory usage are not related to each other. A utility function below defines the utility load l_i of a node n_i.

$$l_i = \lambda * c_i + (1 - \lambda) * m_i$$

$$\lambda = (\sum_{i=1}^{k} c_i) / \sum_{i=1}^{k} (c_i + m_i)$$

Dynamic weights λ and $1 - \lambda$ are used for CPU and Memory in order to capture the runtime load of a node. If CPU is a scarcer resource than memory at a particular instant, then a node with higher CPU usage is more likely to get a higher utility load. This will give the node a higher priority to be balanced.

Standard deviation σ_{cpu} and σ_{mem} are used to evaluate the imbalance degree of CPU and Memory usage among cache nodes, respectively. $\sigma_{cluster}$ captures the cluster's load imbalance degree. Since balancing both CPU and Memory resources may lead to conflictive goals, the formula uses dynamic weights μ and $1 - \mu$ for σ_{cpu} and σ_{mem}, respectively. If CPU is scarcer than memory, it is more likely that the CPU becomes the system bottleneck. In this case, $\lambda > 1 - \lambda$ and $\mu > 1 - \mu$, then CPU is prioritized over memory in the load balancing.

$$\sigma_{cluster} = \sqrt{\mu \sigma_{cpu}^2 + (1 - \mu) \sigma_{mem}^2}$$

$$\sigma_{cpu} = \sqrt{\sum_{i=1}^{k} (c_i - \overline{c})^2 / k}$$

$$\sigma_{mem} = \sqrt{\sum_{i=1}^{k} (m_i - \overline{m})^2 / k}$$

$$\mu = \lambda^2 / (\lambda^2 + (1 - \lambda)^2)$$

4. LOAD BALANCING ALGORITHM

Computing an optimal solution for balancing multiple resources as well as minimizing the reconfiguration cost is an NP-complete problem. Therefore, we use a greedy algorithm to obtain an approximate solution. The algorithm consists of a number of steps $\{step_0, step_1, \dots, step_m\}$. In each step, the algorithm migrates a bucket from the node with heaviest utility load defined in Section 3 to the one with the lightest load. We use a ***nearest-neighbor-search (NN)*** policy in order to select the bucket to be migrated. This policy ensures that at each step of the algorithm, the bucket that can minimize the system imbalance after migration is selected from the heaviest loaded node. The NN policy is derived as follows:

Let us assume that at the beginning of $step_x$, the system imbalance degree is $\boldsymbol{\sigma_{cluster}}$, the most loaded node and lowest

loaded node are $\boldsymbol{n_a}$ and $\boldsymbol{n_b}$, respectively. After moving a bucket from the most loaded to the lowest loaded, the system imbalance degree becomes $\boldsymbol{\sigma^*_{cluster}}$. Our goals are that this imbalance degree is decreased (1) and this imbalance degree is minimized (2).

$$(1)\ \sigma^*_{cluster} < \sigma_{cluster}$$

$$(2)\ \sigma^*_{cluster}\ \textit{is minimal}$$

Suppose buckets $\boldsymbol{b_{aj}}$ with load $(\upsilon_{aj}, \omega_{aj})$ is the bucket we are looking for, according to Section 3, $\sigma^2_{cluster}$ and $\sigma^{*2}_{cluster}$ can be computed as follows:

$$\boldsymbol{\sigma^2_{cluster}} = \mu[(c_1-\bar{c})^2 + (c_2-\bar{c})^2 + \cdots + (c_k-\bar{c})^2]/k +$$
$$(1-\mu)[(m_1-\bar{m})^2 + (m_2-\bar{m})^2 + \cdots + (m_k-\bar{m})^2]/\mathrm{k}$$

$$\boldsymbol{\sigma^{*2}_{cluster}} = \mu\left[\cdots + \left(c_a - \upsilon_{aj} - \bar{c}\right)^2 + \cdots + \left(c_b + \upsilon_{aj} - \bar{c}\right)^2 + \cdots\right]/k +$$
$$(1-\mu)\left[\cdots + \left(m_a - \omega_{aj} - \bar{m}\right)^2 + \cdots + \left(m_b + \omega_{aj} - \bar{m}\right)^2 + \cdots\right]/k$$

In order to get the minimal value of $\sigma^*_{\mathrm{cluster}}$, let us define:

$$\boldsymbol{t} = (\sigma^{*2}_{cluster} - \sigma^2_{cluster}) * k$$

Apparently, finding the minimal value of $\sigma^*_{\mathrm{cluster}}$ can be transformed into finding the minimal value of t.

$$\boldsymbol{t} = \mu\left[\left(c_a - \upsilon_{aj} - \bar{c}\right)^2 - (c_a-\bar{c})^2 + \left(c_b + \upsilon_{aj} - \bar{c}\right)^2 - (c_b-\bar{c})^2\right] +$$
$$(1-\mu)\left[\left(m_a - \omega_{aj} - \bar{m}\right)^2 - (m_1-\bar{m})^2 + \left(m_b + \omega_{aj} - \bar{m}\right)^2 - (m_k-\bar{m})^2\right]$$
$$= 2\mu\left[{\upsilon_{aj}}^2 - (c_a - c_b)\upsilon_{aj}\right] + 2(1-\mu)\left[{\omega_{aj}}^2 - (m_a - m_b)\omega_{aj}\right]$$
$$= 2\mu\left[\upsilon_{aj} - (c_a - c_b)/2\right]^2 + 2(1-\mu)\left[\omega_{aj} - (m_a - m_b)/2\right]^2 -$$
$$[\mu(c_a - c_b)^2 + (1-\mu)(m_a - m_b)^2]/2$$
$$= 2\{[\lambda\upsilon_{aj} - \lambda(c_a - c_b)/2]^2 + [(1-\lambda)\omega_{aj} - (1-\lambda)(m_a - m_b)/2]^2\}/$$
$$[\lambda^2 + (1-\lambda)^2] - [\mu(c_a - c_b)^2 + (1-\mu)(m_a - m_b)^2]/2$$

Since $\lambda, \mu, c_a, c_b, m_a, m_b$ are all constants within a certain step, thus, when $\boldsymbol{t}$ is minimal, the following measure should be minimal:

$$[\lambda\upsilon_{aj} - \lambda(c_a - c_b)/2]^2 + [(1-\lambda)\omega_{aj} - (1-\lambda)(m_a - m_b)/2]^2$$

Therefore, finding the local optimal solution for $step_x$ can be transformed as a *nearest-neighbor-search* problem. In our scenario, each bucket b_{ai} in node n_a can be treated as a point in metric space of dimension 2, since it has two associated load values$(\upsilon_{ai}, \omega_{ai})$. The location of this point (bucket) in space is:

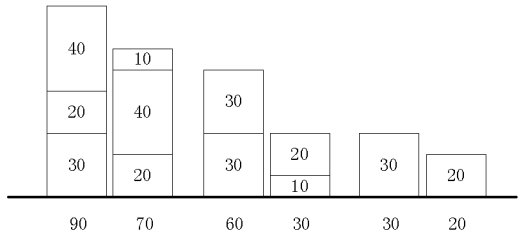
$$b_{ai}: (\lambda\upsilon_{ai}, (1-\lambda)\omega_{ai}), (\upsilon_{ai}, \omega_{ai}) \in BL^*_a$$

Given a point $\boldsymbol{h}$ with location as follows, our target point to search is the one nearest to point $\boldsymbol{h}$, point (bucket) $\boldsymbol{b_{aj}}$ above should be the one closest to point h in node n_a.

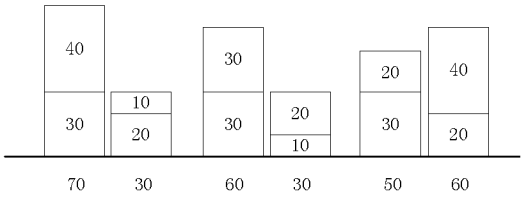
$$h: (\lambda(c_a - c_b)/2, (1-\lambda)(m_a - m_b)/2)$$

Figure 2 shows an example of bucket selection using different policies. Let us assume we have 3 cache nodes in the system, the (CPU, Memory) loads (%) of the nodes are (90, 70), (60, 30), (30, 20), respectively, before data migration. The load of each bucket at each node is shown in Figure 2 (a). Take node (90, 70) as an example, there are 3 buckets, each bucket's CPU and memory load are (30, 20), (20, 40), (40, 10), respectively. According to Section 3, the weight λ = (90+60+30) / (90+60+30+70+30+20) = 0.6. The utility load of each node is calculated as 0.6*90 + (1-0.6)*70 = 82, 0.6*60 + (1-0.6)*30 = 48, and 0.6*30 + (1-0.6)*20 = 26, respectively. Since node (90, 70) is the one with highest utility load, and node (30, 20) is the one with lowest, therefore, a bucket will be migrated from node (90, 70) to (30, 20). If BL was adopted, since Memory is more available than CPU with 20 versus 30 in node (30, 20), bucket (20, 40) demands most of the Memory resource among all the three buckets in node (90, 70), thus, bucket with load (20, 40) would be selected. If BB was applied, in case bucket (40, 10) was selected, the measure (maximum load/average load) on node (30, 20) would be calculated as *max* (40+30, 10+20) / [(40+30+10+20)/2] = 1.4. In the same way, the measure would be 1.1 and 1.2 if bucket (20, 40) and (30, 20) was selected, respectively. Since bucket (20, 40) minimizes this measure, thus, it would be selected for migration. In the case that MM was used, the cost of migrating bucket (40, 10) would be (40*30 + 10*20) / (40+10) = 28. Similarly, the cost would be 23 and 26 if bucket (20, 40) and (30, 20) was selected, respectively. Since bucket (20, 40) has the lowest cost, therefore, it would be selected. However, if our policy NN was applied, we firstly identify the "location" of "point" $\boldsymbol{h}$ introduced previously, which is (0.6*(90-30) / 2, 0.4*(70-20) / 2), namely (18, 10). The "location" of bucket (40, 10) is (0.6*40, 0.4*10), namely (24, 4). In the same way, the "location" of bucket (20, 40) and (30, 20) is (12, 16) and (18, 8) respectively. Bucket (30, 20) would be selected since it is closest to "point" $\boldsymbol{h}$.

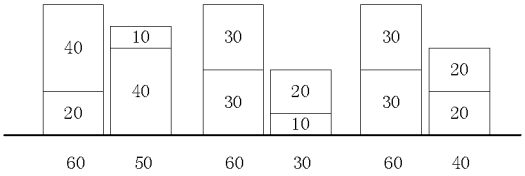
Figure 2(b) shows the result of data migration if BL, BB, MM policy is applied. Figure 2(c) shows the result of data migration of our policy NN. If we migrate bucket (20, 40) as Fig. 2(b) shows, the deviations of CPU and Memory would be 8.2 and 14.1 respectively after migration. However, if bucket (30, 20) is migrated as Figure 2(c) shows, the deviations would be 0 and 8.2, which achieves a better balancing result for multiple resources.



(a). Before data migration



(b). Results of BL, BB, MM



(c). Result of NN

Figure 2. Example of different policies

Our algorithm can be easily transformed to balance any number of resources (multiple dimensions). The above goals must be transformed into finding the nearest neighbor in a metric space *N*, where *N* is the amount of resource dimensions (types). There are many previous studies on *nearest-neighbor-search* problem. Kd-

tree of Friedman and Bentley [20] has been widely adopted to solve this problem. Kd-tree is a space-partitioning data structure for organizing points in a k-dimensional space. We applied kd-tree and nearest neighbor search to find the target bucket in the most loaded node, this achieves a better time efficiency than a linear search. Figure 3 shows the load balancing algorithm we use. Threshold ***IT*** (Improvement Threshold) in % is used in algorithm to trade off the balancing result versus the migration cost. That is, the gains of the "best" load balancing may trade-off the computing and reconfiguration cost needed to achieve this load balancing. The algorithm stops when the system imbalance degree $\sigma_{cluster}$ cannot be decreased anymore or balance improvement has reached the threshold *IT*.

Input: 1. NL 2. $\{B_1, B_2, \dots, B_k\}$

1. **Begin**
2. initialize bucket movements $m = \{\}$;
3. compute *utility load* $\boldsymbol{l_i}$ of each node n_i;
4. compute "*location*" of each bucket b_{ij} in each node n_i;
5. sort all the nodes by *utility load*;
6. build *kd-tree* for each node according to "*location*" of each bucket in the node ;
7. let n_a and n_b be the nodes with maximum and minimum *utility load*;
8. compute initial *cluster standard deviation* $\sigma_{cluster}$;
9. initialize cluster's *balance degree improvement* $\Delta = 0$;
10. set *running* = true;
11. **while**($\Delta <$ *IT* && *running*)
12. **begin**
13. compute "*location*" of point *h;*
14. get the bucket b_{aj} closer to point *h* via *nearest neighbor search* in *kd-tree* of n_a;
15. compute $\sigma^*_{cluster}$ if $b_{aj}$is moved from n_a to n_b;
16. **if**($\sigma^*_{cluster} < \sigma_{cluster}$)
17. add$(b_{aj}, n_a \rightarrow n_b)$ to m;
18. update *kd-tree* in n_a and n_b
19. re-sort all the nodes by *utility load*;
20. identify n_a and n_b;
21. $\Delta = (\sigma_{cluster} - \sigma^*_{cluster})/\sigma_{cluster}$;
22. $\sigma_{cluster} = \sigma^*_{cluster}$
23. **else:** set *running* = false;
24. **end**
25. **return** *m*

Figure 3. Load Balancing Algorithm

5. EVALUATION

In here we evaluate the effectiveness of our dynamic load balancing algorithm with an exhaustive evaluation. We generated programs to generate the various load distributions. BL, BB, MM and NN are separately applied as selection policy and compared of their effects on multi-resource load balancing. For each bucket in a cache node, its size is generated randomly in range]0-1MB] and the data request rate in range [0-1,000/s]. For a cache node, its CPU and memory usage are also generated randomly in the range (10-100%). The number of cache nodes is varied in [20,120]. The number of buckets per node is set to 2,000 and the system balance improvement threshold (*IT*) to 100%. We set the improvement thresholds as 100%, so that the algorithm keeps running until it cannot decrease the imbalance degree anymore, this will allow the comparison of policies for their best case. We evaluate the effectiveness of load balancing in Section 5-A, the reconfiguration cost in Section 5-B and the execution time of the algorithm for different configurations in Section 5-C. All values shown are obtained by an average on 20 runs.

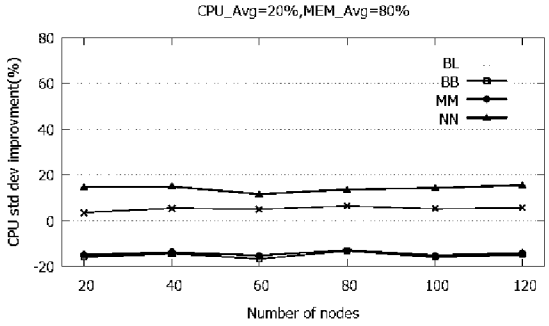


Figure 4. Balancing effect of policies on CPU when avg. (CPU, MEM) loads are (20%, 80%)

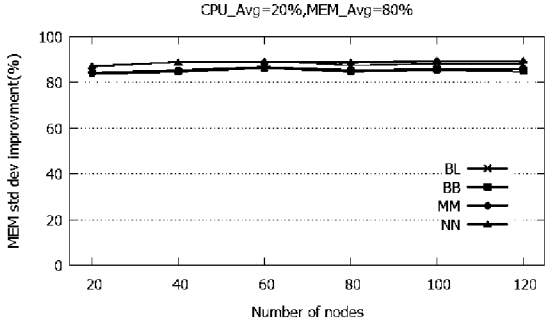


Figure 5. Balancing effect of policies on Memory when avg. (CPU, MEM) loads are (20%, 80%)

A. Effectiveness of load balancing

Since weights λ and μ introduced in Section 3 are dependent on CPU and memory load distributions in system, therefore, our load balancing result is closely related to the CPU and memory distributions. E.g., when the average CPU usage is higher than memory, the CPU is given a higher weight than memory for calculating the utility load l_i and the system imbalance degree $\sigma_{cluster}$, hence, a node with higher CPU usage is more likely to be balanced first and will be given a higher priority to be balanced than memory. In order to validate the effect of this prioritized strategy, we recorded the load balancing result when system's average CPU usage and average memory usage are in different magnitudes. Specifically, we analyze the balancing result when average CPU (CPU_Avg) and Memory usages (MEM_Avg) are (20%, 80%), (35%, 65%) and (50%, 50%), respectively, which are shown in Fig. 4 to 9. The attained balance is measured as the CPU (%) and memory standard deviation (SD) improvements (%). For a CPU usage SD of 30 that becomes 10 after load balancing, the CPU standard deviation improvement is (30-10)/30= 67%.

Figure 4 and Figure 5 show the algorithm effectiveness on

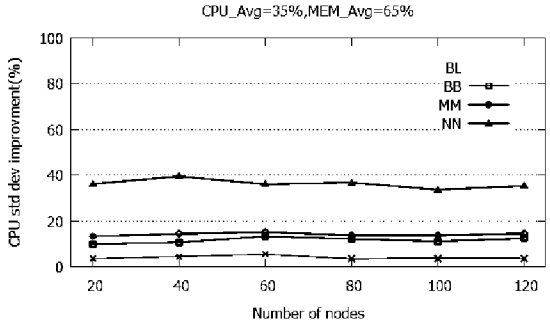


Figure 6. Balancing effect of policies on CPU when avg. (CPU, MEM) loads are (35%, 65%).

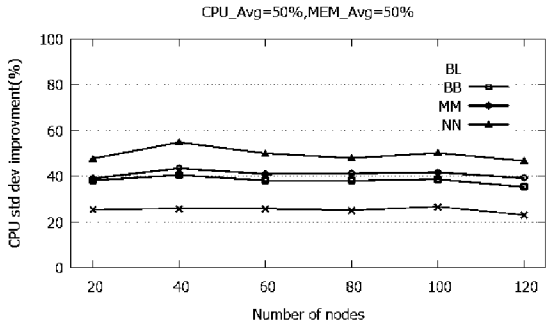


Figure 8. Balancing effect of policies on CPU when avg. (CPU, MEM) loads are (50%, 50%)

balancing CPU and Memory respectively when the average usage of CPU and Memory are (20, 80). Memory is a much scarcer resource in this case, thus Memory will be given a higher priority to be balanced in our policy NN; all selection policies show significant effects on improving the balance degree of memory among cache nodes, our policy NN is slightly better than the others as shown in Figure 5. However, the effectiveness of the different policies are quite different if we look at the CPU balance (Figure 4). BL improves CPU balance, while BB and MM show a negative impact on CPU balance since the CPU standard deviation have increased after load balancing. Our algorithm, NN, provides the best results for CPU balance.

When CPU and memory usages are close to each other, (35, 65) and (50, 50) respectively, BL shows a weaker balancing effect than the others, while our policy NN shows best results for balancing both CPU and memory (Figures 6-9). In case CPU and memory usages are (50, 50), since λ and μ are both 0.5, the two resources have equal weight to be balanced, Figure 8 and 9 shows that the balancing result for CPU and memory are roughly the same with approximately 50% improvement using our policy NN.

Figures 4-9 show that NN can achieve prioritized balancing results for CPU and memory based on the load distributions of both resources. Besides, compared with other policies, our policy NN demonstrates a noticeable improvement on balancing multiple resources. Note that since we randomly generate loads for buckets, there might be relatively light buckets co-existing with heavy buckets in the cache system. The load balancing algorithm will migrate the heavier loaded buckets at the beginning. At a later phase the migration, migrating less loaded buckets will provide a finer grain load balancing.

B. Reconfiguration cost

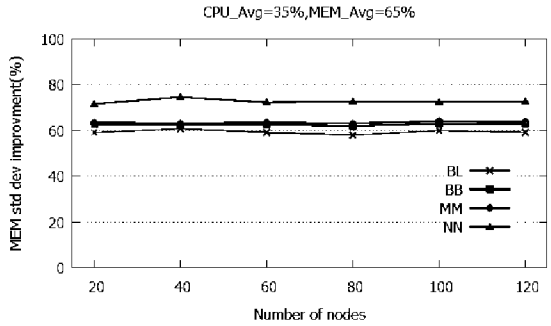


Figure 7. Balancing effect of policies on Memory when avg. (CPU, MEM) loads are (35%, 65%)

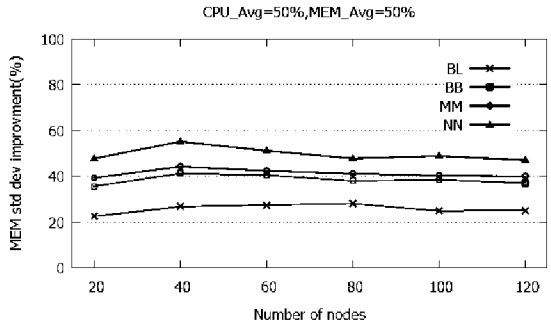


Figure 9. Balancing effect of policies on Memory when avg. (CPU, MEM) loads are (50%, 50%)

In this section we evaluate the reconfiguration costs using different policies. We count the number of buckets to migrate among all the buckets and obtain the percentage value. Figure 10 shows the proportion of migrated buckets using different policies. BL shows the lowest reconfiguration cost; our policy, NN, shows a slightly higher reconfiguration cost compared with BL, while MM shows highest reconfiguration cost followed by BB. Although in terms of reconfiguration cost, BL is better than the others, the result of balancing multiple loads using BL is the worst among all policies.

On the other side, reconfiguration cost can also be affected by the system balance improvement threshold *IT* set in algorithm. We recorded the balancing result when average CPU and memory usage are (50%, 50%). We evaluate the reconfiguration cost when *IT* is set at 50%, 40%, 30%, respectively, using our policy NN. Figure 11 demonstrates that this threshold plays an important role in the cost of reconfiguration and can avoid reconfigurations that are costly but provide a too small improvement in the balance.

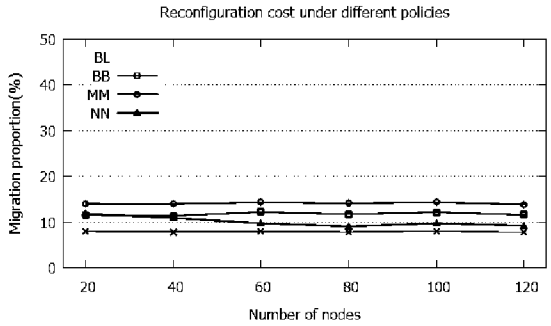


Figure 10. Reconfiguration cost under different policies

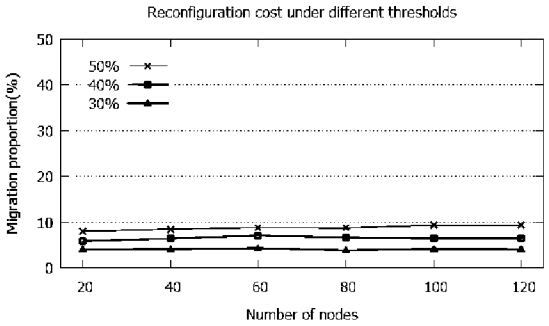


Figure 11. NN reconfiguration cost under different thresholds

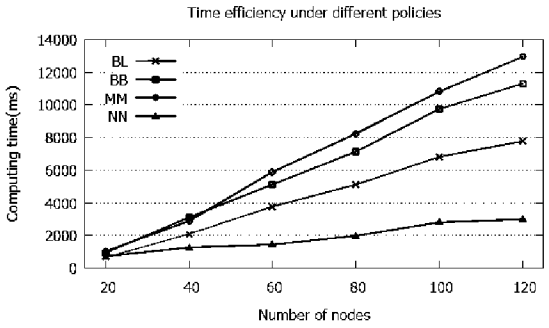


Figure 12. Time efficiency under different policies

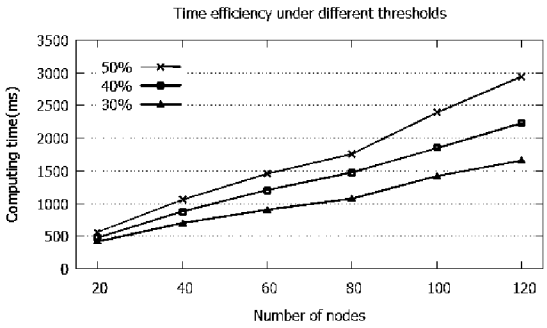


Figure 13. NN time efficiency under different thresholds

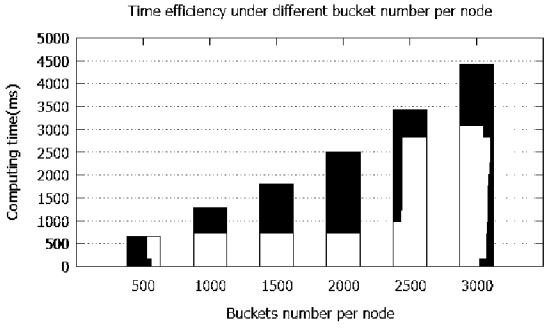


Figure 14. Time efficiency under different buckets number per node

C. *Time efficiency*

We also evaluate the execution time of algorithm using each policy. The result is measured when all policies reach their best load balancing result. Figure 12 shows that our policy, NN, is much better than the others with the lowest execution time. The time to compute the reconfiguration is around 3 seconds in a system with 240,000 buckets; MM takes around 13 seconds to finish, while BB and BL need 11 and 8 seconds, respectively. This is due to the adoption of kd-tree to avoid linear scans to find the optimal solution in each step of algorithm. Kd-tree needs a O (log n) time complexity on average to find the nearest neighbor among buckets, which is much better than scanning linearly all the buckets as BL, BB and MM do, which has an O(n) time complexity. The figure also shows that execution time increases with the number of cache nodes using all policies, but our policy NN has the slowest growing pace.

As it happens with the reconfiguration cost, execution cost may also be affected by the threshold set in algorithm. Nowwe evaluate the execution time using different *IT* values, namely 0.5, 0.4, and 0.3, and values are also recorded when the average load for CPU and Memory are (50%, 50%). Figure 13 shows the trade-off effect of *IT* for balancing result vs. execution cost using policy NN. Note that the execution time shown for NN also includes the time spent in building the k-d trees.

Besides, execution time of policies can also be affected by the number of buckets. In previous simulations, we fixed the buckets number in each node to 2,000. In this experiment we measure the execution time of the NN policy changing the number of buckets per node. Nodes number is set to 100, time cost has been counted when buckets number per node increase from 500 to 3,000. Figure 14 shows that the execution time cost growth with the number of buckets per node.

6. CONCLUSIONS AND FUTURE WORK

We have proposed a multi-resource load balancing algorithm targeting at large cloud cache systems. The algorithm aims at balancing both CPU and memory usage among cache nodes. This is accomplished by migrating data partitions among cache nodes. Our algorithm gives different weights to the resources based on the system load distribution. The scarcer a resource is, the higher its weight is. Simulation shows that compared to previous work, our algorithm attains a better balance for both CPU and memory,

and it also reduces the reconfiguration cost, that is, the amount of data buckets to be migrated, exhibiting the lowest execution time.

In future work, we plan to extend this algorithm to balance multiple resources in a heterogeneous environment, that is, capacities of resources in different cache nodes are different. We would also like to test the effect and efficiency of our proposed algorithm in a real cache system.

7. ACKNOWLEDGMENTS

This work has been partially funded by the European Commission under project CumuloNimbo (FP7-257993) [19], the Madrid Research Council under project CLOUDS (S2009TIC-1692) with funding from ESF and ERDF and the project CloudStorm funded by the Spanish Science Foundation (TIN2010-19077).

8. REFERENCES

[1] Lucas Nealan. Caching & Performance: Lessons from Facebook. Retrieved May 10,2012, from http://www.scribd.com/doc/4069180/Caching-Performance-Lessonsfrom-Facebook

[2] Roebuck, K. *Platform as a Service(PaaS):High-Impact Emerging Technology - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors.* Tebbo Publishers, 2011.

[3] Gualtieri, M. Rymer, J. R. The Forrester Wave™: Elastic Caching Platforms, Q2 2010. Retrieved May 14, 2012,from Forrester Companies: http://www.forrester.com/The+Forrester+Wave+Elastic+Caching+Platforms+Q2+2010/fulltext/-/E-RES55505?docid=55505

[4] Perez-Sorrosal, F., Patiño-Martínez, M. Jiménez-Peris, R. and Kemme, B. Elastic SI-Cache: consistent and scalable caching in multi-tier architectures. *The VLDB Journal — The International Journal on Very Large Data Bases, 20* (6).841-865

[5] Memcached. Retrieved March 15, 2012, from Dormando Companies: http://memcached.org/

[6] Scudder, J. Effective Memcache. Retrieved June, 2012, from https://developers.google.com/appengine/articles/scaling/memcache

[7] You, G. W., Hwang, S. W. and Jain, N. Scalable Load Balancing in Cluster Storage Systems. In *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, (Lisboa, Portugal, 2011), Springer Publishers, 101-122.

[8] Sung Goo Yoo, Kil To Chong. Hot Spot Prediction Algorithm for Shared Web Caching System using NN. In *2007 International Symposium on Information Technology Convergence*, (Jeonju, Korea, 2007), Springer Publishers, 125-129.

[9] Chandra Chekuri. On multidimensional packing problems. *SIAM Journal on Computing.33* (4).837-851.

[10] Csirik, J., Frenk, J. B. G., Labbé, M and Zhang, Shu. On the multidimensional vector bin packing. Acta Cybern Publishers, 1990.

[11] Chekuri C., Khanna S. On multi-dimensional packing problems. In *Proceedings of the 10th annual ACM-SIAM symposium on discrete algorithms*. (Baltimore, Maryland, 1999), Society for Industrial and Applied Mathematics Philadelphia Publishers, 185-194

[12] Patt-Shamir, B., Rawitz, D. Vector bin packing with multiple-choice. In *Proceedings of the 12th Scandinavian conference on Algorithm Theory*.(Bergen, Norway, 2010), Springer Publishers, 248-259.

[13] Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I. Load balancing in dynamic structured peer-to-peer systems. *Performance Evaluation - P2P computing systems,* 63(3). 217-240.

[14] Kunkle D., Schindler J. A load balancing framework for clustered storage systems. In *Proceedings of the 15th International Conference on High Performance Computing,* (Bangalore, India, 2008). Springer Publishers, 57−72.

[15] Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., Soriente, C., Valduriez, P. StreamCloud: An Elastic and Scalable Data Streaming System. In *Proceedings of the 2012 Parallel and Distributed Systems*. Page(s): 1

[16] Milan-Franco, J. M., Jiménez-Peris, R., Patiño-Martínez, M. and Kemme, B. Adaptive Middleware for Data Replication. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*,(Toronto, Canada, 2004). ACM Publishers, 175-194.

[17] Leinberger, W., Karypis, G., Kumar, V. Job scheduling in the presence of multiple resource requirements. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, (Portland, Oregon, USA, 1999) ACM Publishers.

[18] Chih-Chiang, Y., Kun-Ting, C., Jing-Ying, C. Market-Based Load Balancing for Distributed Heterogeneous Multi-Resource Servers. In *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems* (Shenzhen, China, 2009) IEEE Computer Society Publishers, 158-165.

[19] Jiménez-Peris, R., Patiño-Martínez, M., Magoutis, K. Bilas, A. and Brondino, I. CumuloNimbo: A highly-scalable transaction processing platform as a service. *ERCIM NEWS n° 89, Special theme Big Data*, 34-35. April 2012.

[20] Friedman, J. H., Baskett, F. and Shustek, L. J. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, C-24(10). 1000-1006.