A Client-Side Cloud Cache Replacement Policy

Thepparit Banditwattanawong [1] and **Putchong Uthayopas** [2], Non-members

ABSTRACT

To deliver data-centric cloud computing services heavily relies on data networks between cloud providers and consumer premises. The continuous and rapid growth of data hosted in external private clouds accelerates downstream network-bandwidth saturation and public cloud data-out overspends. The consumer-initiated replication of cloud data to consumer locality is a solution and known as client-side cloud caching. This paper presents the core mechanism of the cloud caching, called Cloud cache replacement policy. Simulations shown that 1) Cloud saved network bandwidth, data-out charge and data loading time; 2) even Clouds performance minima outperformed three well-known web cache replacement policies across all performance metrics for almost all test cases; 3) Cloud importantly attain optimal hit and byte-hit ratios without sacrificing one to the other.

Keywords: Cloud Computing, Cloud Cache, Cache Replacement Policy, Cost Saving Ratio, Window Size, Contemporaneous Proximity

1. INTRODUCTION

Network scalability, economy and responsiveness are the principal requirements of data-intensive cloud computing. In presently digital cultures, data has been proliferating in various forms such as instant messaging texts, high definition images and voices, massively streaming videos and SaaS applications. Not only WWW and social media contents but also big data (e.g., archive of videos gathered via ubiquitous information-sensing devices) have been hosted in cloud and being shared in a distributed fashion in an unprecedented volume. These will lead to several problems from a consumer standpoint; the bandwidth saturation of network connection between external cloud and consumer premise, increases in external-private-cloud data-out charge imposed by public cloud provider (such as [1], [2] and [3]) and long-delayed cloud service responsiveness.

While allocating more organizational budget in long term is a possible solution to these problems, cloud economy is however questionable. Improving both cloud scalability, economy and service responsiveness simultaneously can be attained instead by deploying client-side cloud cache system, which is located nearby the consumer premise, such as Amazon CloudFront [4].

The cloud cache basically inherits the capability of traditional forward web caching proxy since cloud data is also delivered by using the same set of HTTP/TCP/IP protocol stack as in WWW. Unavoidably, an issue in WWW caching is also inherited that is caching entire remote data in local cache is not economically plausible, thus cache replacement policy is also mandatory for cloud cache.

This paper presents a cache replacement policy, Cloud (named so for its intended application domain), along with a set of technical and economical performance results and findings based on half- and one-month HTTP traces.

2. RELATED WORK

Reference [4] has originally proposed Cloud strategy as an early attempt of client-side cloud cache replacement policy, and evaluated its performance by using short 5 day traces. In [6], we have made the minor algorithm enhancement of Cloud and extended its performance investigation by using 15 day traces in conjunction with 5 day traces. The 15-day results has led to an important finding in the realm of heterogeneous multicomputer caching that is Cloud could deliver optimal byte-hit and hit performances at the same time. In [7], a new set of even longer one-month traces was used in comparison with 15-day traces to observe Clouds behaviors in longer runs. This was to ensure its performance practicality and stability in more realistic environments to provide user confidence in employing cloud caching services. The main finding in [6] was also affirmed by the one-month results. Moreover, both performance maxima and minima of Cloud were presented in comparison with three well-known strategies, LRU, GDSF and LFUDA [8], which are all supported by popular Squid caching proxy [9]. This paper incorporates [5-7] with better content clarification.

We have investigated a number of other web cache

Manuscript received on July 12, 2013 ; revised on December 6, 2013.

Final manuscript received December 23, 2013.

[1] The author is with School of Information Technology, Sripatum University, Bangkok, Thailand., E-mail: thepparit.ba@spu.ac.th

[2] The author is with Computer Engineering Department, Faculty of Engineering, Kasetsart University, Bangkok, Thailand., E-mail: pu@ku.ac.th

replacement policies in terms of their characterizing parameters as follows.

2.1 Object Size, Loading Cost and Access Frequency

A number of policies surveyed in [8]: LRU, LFU-DA, EXP1, Value-Aging, HLRU, LFU, LFU-Aging, -Aging, swLFU, SLFU, Generational Replacement, LRU*, LRU-Hot, Server-assisted cache replacement, LR, RAND, LRU-C, Randomized replacement with general value functions, including policies ARC [10], CSOPT [11], LA2U [12], LAUD [12], SEMALRU [13] and LRU-SLFR [14] do not parameterize object sizes. If big objects are requested frequently but often evicted by these policies (as blind to object sizes), caching proxy will have to frequently reload the big objects from their original servers. Therefore, object-size ignoring schemes permit the poor economies of data-out charges.

Another group of policies surveyed in [8]: GDSF, LRU-Threshold, LRU-Min, SIZE, LOG2-SIZE, PSS, LRU-LSC, Partitioned Caching, HYPER-G, CSS, LRU-SP, GD-Size, GD*, TSP, MIX, HYBRID, LNC-R-W3, LUV, HARMONIC, LAT, GDSP, LRU-S, including LNC-R-W3-U [15], SE [16], R-LPV [17], Min-SAUD [18], OPT [19], LPPB-R [20], OA [21], CSP [22], GA-Based Cache Replacement Policy [23], improved GD* [24] and SzLFU(k) [25] consider object sizes in such a way that replacing bigger objects first, thus not aiming for cost-saving performance. The other policies LRV [7], M-Metric [8], NNPCR-2 [26] and Bolot and Hoschkas [27] favor bigger objects like Cloud. In particular, LRV supports the factorization of object size, downloading latency or loading cost but only one of them at each time; M-Metric allows bigger objects to stay longer in cache but does not incorporate loading cost parameter; NNPCR-2 applies neural network to decide the evictions of small or big objects but the perceptron does not input monetary cost parameter to control data-out charge; Bolot and Hoschka's policy replaces bigger objects first but ignores spatial locality by not considering access frequencies and does not support the nonuniform costs of object downloadings.

2.2 Access Recency

All known policies prioritize the recencies of object requests either implicitly or explicitly. By implicitly, every policy always accepts a newly loaded missing object (i.e., the most recently used object) into cache rather than rejects it. By explicitly, several policies such as LRU, LRU-Threshold, SIZE, LRU-Min, EXP1, Value-Aging, HLRU, PSS, LRU-LSC and Partitioned Caching parameterize elapsed times since the last requests to objects. Cloud policy explicitly regards the recency property of objects in its model.

2.3 Object Loading Latency

Several policies: GD-Size, GDSF, GD*, improved GD*, GDSP, HYBRID, LAT, LUV, MIX, LNC-R-W3, LNC-R-W3-U, LRU-SLFR and GDSP take object loading latencies into account. All of them replace objects with shorter latencies first. Cloud policy also follows such a design approach.

2.4 Object Expiration

Very rare policy considers object expiration. LA2U, LAUD, improved GD* and LNC-R-W3-U replace frequently updated objects first. The former three do not describe how update frequencies are derived. The last policy estimates update frequencies from changes detected in HTTP's 'Last-Modified' header fields; however, if frequently updated objects are seldom requested, updated 'Last-Modified' values will be rarely perceived by the policies resulting in underestimated update frequencies. This problem can be solved by using explicit expiration times or TTL as in Bolot and Hoschka's policy even though this parameter had not yet been implemented in their presented empirical studies.

To recap, no explicit policy aims for cloud computing paradigm for either of two main reasons. First, they evict big objects to optimize hit rates rather than byte-hit and delay-saving ratios, which are important to the scalability of cloud-transport infrastructures and cloud computing services. Second, they are not optimized for cloud data-out charges, thus fail to improve consumer-side cloud economy.

3. CLOUD CACHE REPLACEMENT POLICY

Designing cache replacement policies for cloud cache systems requires the new perspective of problem understanding to establish design goals, which truly suites cloud computing paradigm.

3.1 Design Goals

The design goals of Cloud policy would be described in terms of standard performance metrics borrowed from the realm of traditional web caching and newly invented as follows.

- *Cloud aims for optimal byte-hit ratio:* As described earlier, the scalability of cloud network infrastructure is vital due to the continual and rapid growth of data in cloud computing era that is reinforced by at least the two recent emergences of social media contents and big data.
- *Cloud aims for cost-saving ratio:* Cost-saving ratio is an economical performance metric originally proposed in [4]. As mentioned formerly, public cloud providers charge the volume of data transferred out of their cloud infrastructures down to consumer premises. This incurs financial difficulty

for consumers in meeting cost-effectiveness, acceptable pay-back period, or even a break-even point.

- *Cloud aims for improving delay-saving ratio:* Cloud computing paradigm is heavily network-driven because both enterprise data and services must be made available to users through the network. It is intolerable for cloud consumers to experience unresponsive SaaS owing to unpredictable or congested network connectivity.

It should be worth to clarify why hit rate is not included in the designs goal of Cloud unlike those found in the classic world of web caching. Date back to the pre-broadband communication era, fetching even small data object was delayed due to slow Internet connection. It was unacceptable for users to experience such delays on every request. For this reason, the web cache replacement policies had been optimized for hit rate to serve small objects as fast as possible [8, 28]. However, these days with globally available large-bandwidth network infrastructures, it is perceived that loading remote small objects is fast as if they were loaded from user locality. This fact has advocated several present enterprises to overlook the employment of web caching proxies. In other words, hit rate is considered no longer important for the broadband era. This evolutionary phenomenon also applies to cloud computing network infrastructures, and therefore Cloud excludes hit rate from its design goals.

3.2 Design Rationales

Cloud strategy mandates several parameters to make the efficient decisions of cached object replacements. The utilizations of the parameters is justified one by one as follows.

- Cloud strategy relies on contemporaneous proximity principle [29], which combines both spatial locality and temporal affinity together. Thus, Cloud parameterizes both the access frequency and recency of objects. Popular objects and most recently used objects are preserved in cache.
- One of the problems that cloud consumers experience for now is the loadings of big objects through IaaS, PaaS and SaaS take long time, whereas the loadings of small objects are no longer a problem in current broadband era as mentioned earlier. Therefore, Cloud parameterizes object size by favoring big objects in cache and evicting small ones. By retaining big objects in cloud cache, the bandwidth congestion of cloud transport infrastructures could be alleviated.
- Economy is highlighted as the major benefit of cloud computing paradigm. By solely avoiding the repeated loadings of big objects might not optimize data-out charges. If an enterprise employs multiple public cloud providers offering the different pricing rates of data-out charges, small objects can be more expensive than big ones. To control economical aspect, Cloud parameterizes data-out charge rate by preserving expensive objects in cache.
- Responsiveness becomes more important characteristic to cloud services as they are run via the network while consumers expect their qualities of services comparable to those of local applications. Cloud parameterizes data loading latency by replacing fast loaded objects before slowly loaded ones in cache.
- When objects in cache become stale or nearly expire, they remain fewer chances to get accessed. Thus, Cloud parameterizes the remaining lifespans of objects by evicting those expired or nearly stale from cache.

Cloud policy functions as follows. Whenever Cloud cache replacement mechanism is activated, Cloud first formulates a cluster of in-cache least-recently-used objects as many as instructed either by a window size parameter or the size of a newly missing object if the window size yields smaller total object size than the missing object. Cloud policy also calculates a profit for each object inside the cluster by using Eq. (1). For an object i,

```
algorithm: Cloud

inputs:
        ro      /*a requested object*/,
        cd      /*a cache database (recency-keyed min-
                  priority queue)*/,
        ws      /*a window size*/
local variables:
        ecd     /*empty cache database (recency-keyed min-
                  priority queue)*/
        rs      /*required cache space*/,
        ecd     /*an empty cache database (recency-keyed
                  min-priority queue)*/,
        oc      /*object cluster of LRU objects (profit-keyed
                  min-priority queue of evictable objects)*/,
        co      /*a candidate object to be included in a
                  cluster*/,
        ts ←0 /*total size of objects within ws initialized to
                  zero*/,
        eo      /*an evicted object*/,
        c ←0 /*a counter for objects in a cluster initialized
                  to zero*/
begin
        rs ← ro.getObjectSize();
        if      cd.getTotalNumberOfObjects() < ws
        then  ws ← cd.getTotalNumberOfObjects();
        ecd ← cd;
        do{
                co ← ecd.removeLeastRecentlyUsedObject();
                ts  ← (ts + co.getSize());
                oc.addObject(co);
                c ← (c + 1);
        }while  (c < ws) ∨ (ts < rs);
        do{
                eo ← oc.removeMinProfitObject();
                fs ← fs + eo.getSize();
                cd.evict(eo);
        }while  cd.getFreeSpace() < rs;
return cd.getFreeSpace();
```

Fig.1: *Cloud Algorithm.*

$$profit_i = s_i \cdot c_i \cdot l_i \cdot f_i \cdot TTL_i \quad (1)$$

where s_i is the size of i, c_i is data-out charge rate for loading i, l_i is latency in loading i, f_i is the access frequency of i, and TTL_i is the remaining lifespan of i. An object with least profit is first evicted from cache. This eviction decision process is repeated on a next least profitable object in the cluster until gaining enough room in cache that fits the missing object. The detailed pseudo code of Cloud algorithm is shown in Fig. 1.

3.3 Algorithmic Practicality

As for the time complexity analysis of Cloud illustrated in Fig. 1, the statements that take significant part in processing time are: Replicating *cd* into *ecd* takes O(N); the first *do-while* loop takes O(NlogN) as the window size can be set via the preceding *if-then* statement to as many as N, removing each object from *ecd* is O(logN) just like adding each object into *oc*; the second *do-while* loop has the worst-case running time of O(NlogN) because the number of evicted objects is bounded by N, while removing each object from *oc* takes O(logN) and deleting an object from cd is O(logN). The other statements are all identically O(1). Therefore, the algorithm is O(NlogN). In other words, Cloud strategy can be implemented.

4. PERFORMANCE EVALUATION

4.1 Input Data Sets

In the realm of client-side caching, simulation was a popular method for performance assessment as done in almost all of the related works. Hence, HTTP trace-driven simulations were conducted using four traces provided by [30] that represent two different user-community behaviors (for result crosschecking purpose): 31-day BO trace gathered from user community in Boulder from 16$^{\text{th}}$ August to 15$^{\text{th}}$ September 2012 and 31-day NY trace collected from user community in New York from 16$^{\text{th}}$ July to 15$^{\text{th}}$ August 2012. The other two 15-day traces will be described later.

Each of these raw one-month traces was preprocessed to extract only the object references of 50 most popular domain names (emulating the total number of intranet domains administrated within the first author's university). Omitting this preprocessing step means that a single consumer organization owns an impractically large number of domain names hosted on its cloud(s). The extracted references reflected both WWW and cloud service requests to, for example, Facebook, Youtube, and Twitter SaaSes as well as Mediafire and 4shared IaaSes. The WWW requests were not excluded as the majority of them were found to go to cloud-hosted web servers. We thus regarded the entire preprocessed traces as cloud service request streams. Furthermore, unused fields were removed, and object expiration time was appended as a new field to every reference inside all the traces. These field values were figured out by using the following heuristic rules based on an assumption that object creators set objects' expiration times deliberately.

- *Rule 1:* By scanning down all references within a trace in timestamp order, object expired immediately once its size was found changed.
- *Rule 2:* As long as the size of object remained unchanged, its lifetime was extended to its last request as appeared in a trace.
- *Rule 3:* Object apparent only once throughout a trace expired suddenly after its use. (This object tends to be a kind of dynamic one, which is by default not cached by Squid. Thus, this rule is defined so.)

In practice, the expiration value can be retrieved from 'Expires' or 'max-age' field inside HTTP protocol header [31]. All references representing dynamic object requests were not excluded from the traces during the preprocessings to reflect actual performances against all types of requests. (That is why our performance results were not so high, unlike some of the related works.)

Once the one-month BO and NY traces had been preprocessed, the references belonging to the first 15 days of each trace were duplicated into a new trace resulting in a 15-day BO trace and a 15-day NY trace. This allows objects in both 15-day traces to have maximum one-month lifespans (as a result of applying the heuristic rules to the one-month traces) rather than merely 15 days. This makes the heuristic rule 3 more practical for static objects. The preprocessing results are characterized in Table 1. Notice that the last feature must be the maximum bytes (of total unique objects) since the sizes of several unique objects were found to change over a reference stream in each trace.

Table 1: *Characteristics of simulated traces.*

Feature	BO		NY	
	15 days	31 days	15 days	31 days
Total requests	352,224	639,187	639,199	1,311,880
Total requested bytes	2,294,688,191	4,149,211,314	6,499,655,874	17,067,821,671
Total unique objects	181,624	323,979	290,851	593,365
Max. bytes of total unique objects	1,386,970,321	2,262,144,480	4,791,008,825	10,801,010,237

4.2 Performance Metrics

The three standard metrics [8, 28] in the world of traditional web caching were used for cloud cache performance measurements. They are defined as follows.

For an object i,

$$byte-hit\ ratio = \frac{\sum_{i=1}^{n} s_i h_i}{\sum_{i=1}^{n} s_i r_i} \quad (2)$$

$$delay-saving\ ratio = \frac{\sum_{i=1}^{n} l_i h_i}{\sum_{i=1}^{n} l_i r_i} \quad (3)$$

$$hit-rate = \frac{\sum_{i=1}^{n} h_i}{\sum_{i=1}^{n} r_i} \quad (4)$$

where s_i is the size of i, h_i is how many times a valid copy of i is found in cache, r_i is the total number of requests to i, and l_i is the loading latency of i from cloud.

In addition to the standard metrics, a new metric cost-saving ratio was used to capture the economical performances of all the simulated strategies. In particular, cost-saving ratio measures how much loading monetary cost can be saved by serving the valid copies of requested objects from local cache. For an object i,

$$cost-saving\ ratio = \frac{\sum_{i=1}^{n} c_i s_i h_i}{\sum_{i=1}^{n} c_i s_i r_i} \quad (5)$$

where c_i is the data-out charge rate for loading i from cloud. This metric is particularly useful for hybrid cloud where organization employs multiple cloud providers, which might charge data-outs based on different pricings. Notice that in a uniform cost model (described in next sub-section), the value of c_i in Eq. (5) is identical for all objects, thus can be eliminated and resulting in Eq. (2). This means that cost-saving ratio is absolutely equal to byte-hit ratio in uniform cost model. In fact, economy is accepted as the core benefit of cloud computing, therefore this metric is helpful for future cache replacement strategies aiming for cloud computing for demonstrating their economical performances, which are highly of cloud consumer interests.

4.3 Cost Models

To demonstrate economical performances based on Eq. (5), there can be two cost models to be used in simulations. One is a uniform cost model in which all the 50 domains are assumed to be hosted on the same public cloud provider, thus the same data-out charge rate was applied uniformly across all requested objects. The other is a nonuniform cost model in which the 50 domains were assumed to be hosted on two or more public cloud providers probably for risk management, which is critical to some types of businesses (e.g., hospitals, stock trading and air transportation control) so that different charge rates were associated with the domains. Objects fetched from the same domain were always charged at the same rate. Nonuniform cost-based performance results are not presented here but can be found in [5] and [6].

4.4 Window Size Parameter

An algorithmic parameter significantly influencing all performance aspects of Cloud is the window size. Both optimal and worst window sizes were used in simulations and determined by extensively trial-and-error experiments. In particular, optimal window size was tuned for a maximum byte-hit ratio (or cost-saving ratio) for each certain trace and cache size using the uniform cost model. Since different window sizes were found to yield identical maximum byte-hit, the smallest window size was selected as the representative value of optimal window size due to requiring less CPU time. Optimal window sizes yield performance maxima, whereas worst window sizes produce performance minima and were tuned towards minimum byte-hit ratios based on the uniform cost model.

Table 2 shows the optimal window sizes (wso) and the worst window sizes (wsw) used for the simulations. The cache sizes are presented in percent of the maximum bytes of total unique objects of the 15-day traces shown in Table 1. This means that the simulated cache sizes were fixed to absolute ones and used to serve request streams beyond the first 15 days (i.e., the one-month trace durations). This emulated realistic condition in which cache size in practice cannot grow without limitation in proportion to the maximum bytes of total unique objects, requested throughout a cloud cache's entire running period, which can be several years. Infinite cache size, which enables cachings without cache replacement at all, was also simulated against each trace to show performance upper bounds. The infinite cache size for each trace must be set to at least the maximum bytes of total unique objects of each trace shown in Table 1.

By looking at Table 2, one finding can be drawn early: both optimal and worst window sizes were not proportional to cache sizes.

Table 2: *Window sizes used for Cloud policy simulations.*

Cache size (of first 15 days)	BO				NY			
	15 days		31 days		15 days		31 days	
	wso	wso	wso	wso	wso	wso	wso	wso
10%	500	25,000	8,000	35,000	2,000	100	1,000	60,000
20%	1,300	40,000	8,000	60,000	5,000	100	8,000	100
30%	100	60,000	8,000	5,000	500	80,000	3,500	100

4.5 Results and Discussions

First, let us consider the simulation results in four performance metrics using one-month BO and NY traces shown in Fig. 2 and Fig. 3, respectively, of Cloud in comparison with LRU, GDSF and LFUDA strategies. (To accommodate the differentiation of the presented performance values among the simulated policies, the Y-axis minimum values of all

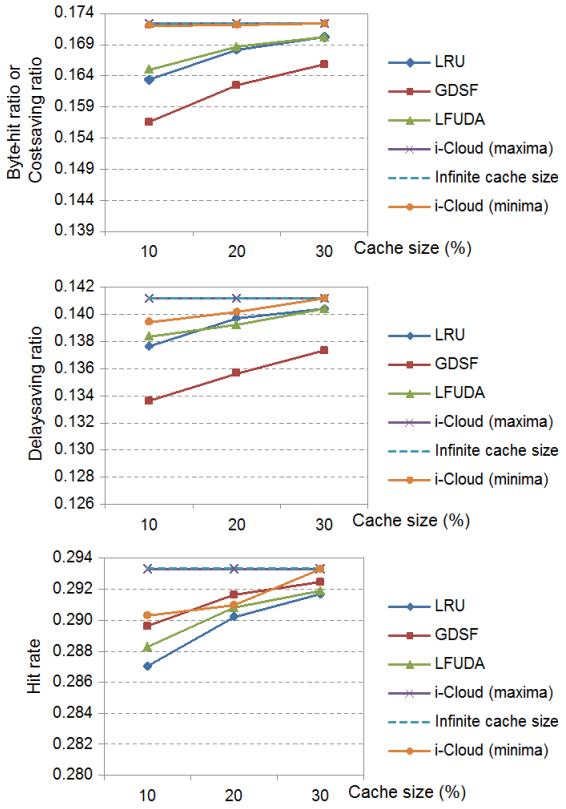

Fig.2: *Performance results using 31-day BO trace.*

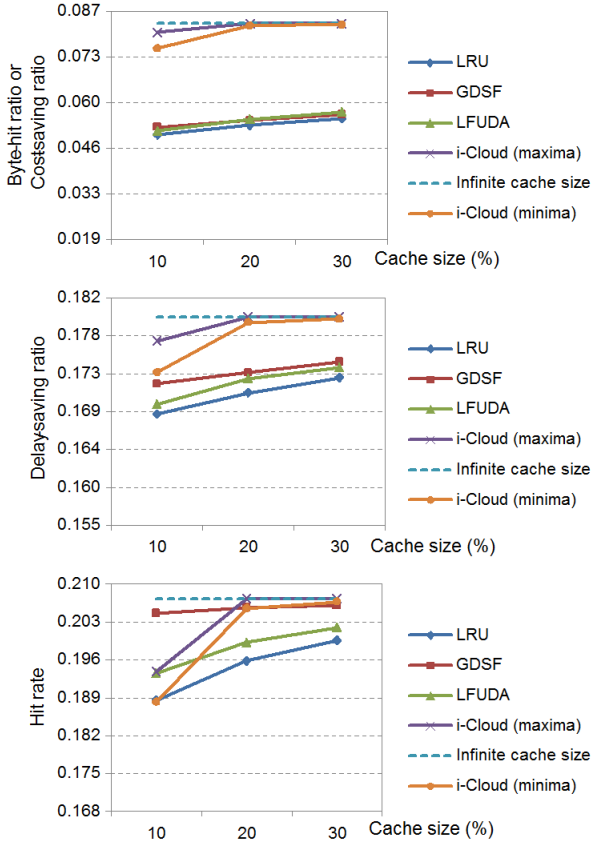

Fig.3: *Performance results using 31-day NY trace.*

graphs in Fig. 2 to 5 are not bounded to zero.)

- They demonstrate that Cloud produced against the other three strategies the optimal byte-hit, cost-saving and delay-saving performances for all the cache sizes. Optimal hit rates were also delivered by Cloud when using 20% and 30% cache sizes for both traces and 10% cache size for BO trace. It should be noted that across 20% and 30% cache sizes Cloud delivered perfectly stable performance maxima in all metrics very close to those of infinite cache size. This substantiates a very promising client-side cloud caching as a service. The results, however, came out at the price of trial-and-error tuning of the optimal window sizes, shown in Table 2.
- At 20% and 30% cache sizes, Cloud also performed best not only in byte-hit metric but also hit-rate one. Achieving the superior hit and byte-hit ratios simultaneously leads to a groundbreaking finding: it is not always that a replacement policy trades off byte-hit ratio against hit rate. Because Cloud tends to remove smaller objects, this finding contradicts the conventional rule of thumb in traditional web caching: "Strategies that tend to remove bigger objects improve the hit-rate but decrease the byte-hit-rate" [8]. Another finding that places an objection to the rule of thumb is that a policy removing smaller objects could actually improve hit rate. These are probably because Cloud strategy not only relies on object sizes but also several other input parameters. The other finding supported by the rule of thumb is that a policy preserving bigger objects enhanced byte-hit ratio.
- Even the Cloud's performance minima in terms of byte-hit, cost-saving and delay-saving ratios also outperformed the other policies throughout all cache sizes. The hit-rate minima were rather unpredictable in the rank using 10% and 20% cache sizes but turned out to overcome all the other policies using 30% cache size

Now, let us compare the one-month results shown in Fig. 2 and 3 with the shorter 15-day results shown in Fig. 4 and 5.

- It can be seen across all cache sizes that the ideal, maximum and minimum performances of Cloud based on the BO and NY traces in terms of byte-hit, cost-saving and delay-saving ratios improved in longer runs whereas the hit rates decreased. In particular, deploying Cloud in BO community for one month instead of 15 days increased byte-hit and cost-saving performance maxima by 0.01617 to 0.01620 depending on the cache sizes and increased delay-saving performance maxima by 0.0026 across all cache sizes, whereas hit performance minima decreased by 0.00648 to 0.00655 depending on the cache sizes. On the other hand, deploying Cloud against NY community behavior for one month in-

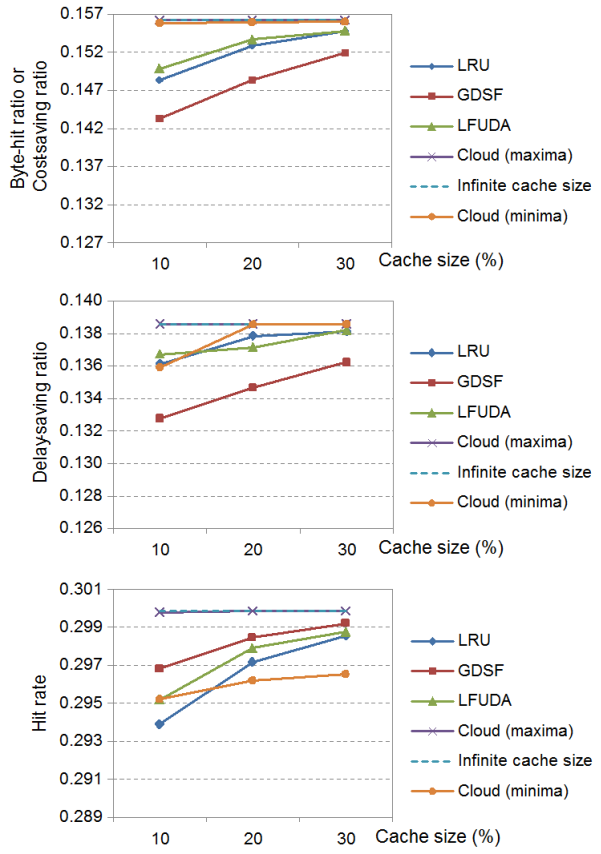

Fig.4: *Performance results using 15-day BO trace.*

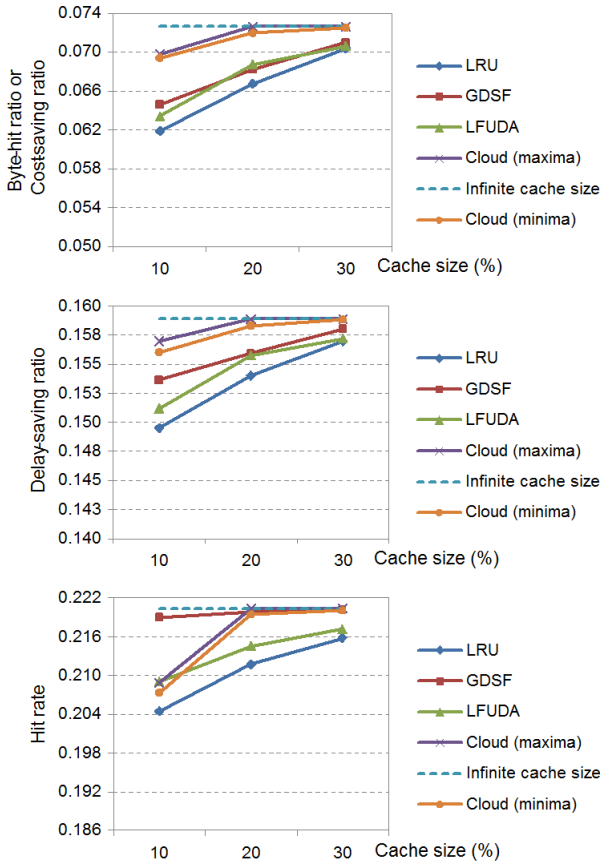

Fig.5: *Performance results using 15-day NY trace.*

stead of 15 days increased byte-hit and cost-saving performance maxima by 0.01064 to 0.01084 depending on the cache sizes and increased delay-saving performance maxima by 0.01987 to 0.02079 depending on the cache sizes, whereas hit performance minima decreased by 0.01309 to 0.01587 depending on the cache sizes. Notice that when trace durations were doubled, Cloud's byte-hit and cost-saving ratios using BO traces were more fluctuated than NY ones, whereas the delay-saving and hit rates using BO traces were less fluctuated than NY ones.

- Based on the longer BO trace for all cache sizes, the performance maxima and minima in all metrics except hit rate of Cloud improved. Based on the longer NY trace, the performance maxima and minima in all metrics except hit rate of Cloud also improved. This can be inferred that although the absolute cache sizes used against both one-month traces were as small as those of 15-day traces, Cloud's performance maxima and minima could improve in terms of byte-hit, cost-saving and delay-saving ratios.
- By contrasting the overall performance results, Cloud seemed fitter for BO community than NY one except for the delay saving performances.

It is worth to conduct the further analysis of how much benefit of Cloud can be gained from window-size tuning. In other words, how maximum the performance of Cloud is improved when window size was switched from the worst values to the optimal ones. This can be examined by measuring the widest gaps between the maxima and minima performances among both one-month traces and all simulated cache sizes: the widest gaps between performance maxima and minima can only be found in Fig. 3 and were as much as 0.47% byte-hit ratio and cost-saving ratio, 0.37% delay-saving ratio, and 0.56% hit rate. The significances of these maximum performance improvements could be translated to more meaningful information as follows.

- The 0.47% gap of byte-hit ratio can be translated roughly to 21.48 TB of saved data transfer per annum based on a representative scenario where cloud data is transferred through 10 Gbps Metro Ethernet with 50% bandwidth utilization for 8 work hours a day, and 260 workdays per year.
- The 0.47% improvement of cost-saving ratio can be translated to about 2,639.52 USD saved per annum based on the representative scenario and Google App Engine's data-out charge rate (which is 0.12 USD/GB as of June 2013 [2]).
- The 0.37% improvement of delay-saving ratio can be interpreted as approximately 7.7 hours per year of saved data-transfer time based on the represen-

tative scenario.

Nevertheless, actual performance improvements might be greater or less depending on the inherent characteristics of user-generated workloads (e.g., contemporaneous proximity, object sizes, TTLs and loading latencies), data-out charge rates and data transfer volumes in real environments where Cloud policy is deployed.

5. CONCLUSION

This paper presents Cloud cache replacement policy that aims to improve cloud scalability, economy and responsiveness. Cloud has access recency as a priority factor for object replacement decision. Cloud parameterizes a window size to generalize the recencies of objects within a formulated object cluster. The lowest profitable clustered objects are purged from cloud cache.

Using proper window sizes, Cloud's performances boosted over LRU, GDSF and LFUDA whereas poor window sizes degraded Cloud's performances. A breakthrough finding is that Cloud could attain both high byte-hit and hit rates at the same time based on our test cases. Another significant benefit of Cloud against the other well-known strategies is its stable optimal performances in byte-hit, cost-saving and delay-saving ratio metrics based on most simulated cache sizes.

Finally, we are developing the automated discovery mechanism of optimal window sizes for Cloud policy.

6. ACKNOWLEDGEMENT

This research is financially supported by Thailand's Office of the Higher Education Commission, Thailand Research Fund, and Sripatum university (grant MRG5580114). The authors also thank Duane Wessels, National Science Foundation (grants NCR-9616602 and NCR-9521745) and the National Laboratory for Applied Network Research for the trace data.

References

[1] Amazon.com Inc. (2013). *Amazon Simple Storage Service* [Online]. Available: http://aws.amazon.com/s3/#pricing

[2] Google Inc. (2013). *Google App Engine Pricing* [Online]. Available: https://cloud.google.com/pricing/index

[3] Microsoft. (2013). *Windows Azure* [Online]. Available: http://www.windowsazure.com/en-us/pricing/details/

[4] Amazon.com Inc. (2013). *Amazon CloudFront* [Online]. Available: http://aws.amazon.com/cloudfront/

[5] T. Banditwattanawong, "From Web Cache to Cloud Cache," *Proceedings of 7th International Conference in Grid and Pervasive Computing, Hong Kong*, pp. 1-15, 2012.

[6] T. Banditwattanawong and P. Uthayopas, "Cloud Cache Replacement Policy: New Performances and Findings," *Proceedings of 1st Annual PSU Phuket International Conference 2012*, 2013.

[7] T. Banditwattanawong and P. Uthayopas, "Improving Cloud Scalability, Economy and Responsiveness with Client-Side Cloud Cache," *Proceedings of 10th IEEE International Conference ECTI-CON 2013*, 2013.

[8] S. Podlipnig, and L. B?sz?rmenyi, "A Survey of Web Cache Replacement Strategies," *ACM Computing Surveys*, Vol. 35, pp. 374-398, 2003.

[9] D. Wessels, Squid: The Definitive Guide. O'Reilly Media Inc, 2004.

[10] N. Megiddo and D. S. Modha, "Outperforming LRU with an Adaptive Replacement Cache Algorithm," *Computer*, Vol. 37, pp. 58-65, 2004.

[11] J. Jeong and M. Dubois, "Cache Replaccement Algorithms with Nonuniform Miss Costs," *IEEE Transactions on Computers*, Vol. 55, pp. 58-65, 2004.

[12] H. Chen, Y. Xiao, and X. S. Shen, "Update-Based Cache Access and Replacement in Wireless Data Access," *IEEE Transactions on Mobile Computing*, Vol. 5, pp. 1734-1748, 2006.

[13] K. Geetha, N. A. Gounden, and S. Monikandan, "Semalru: An Implementation of Modified Web Cache Replacement Algorithm." *Proceedings of NaBIC. IEEE*, pp. 1406-1410, 2009.

[14] S.-W. Shin, K.-Y. Kim, and J.-S. Jang, "LRU Based Small Latency First Replacement (SLFR) Algorithm for the Proxy Cache," *Proceedings of the 2003 IEEE/WIC International Conference on Web Intelligence*, 2003.

[15] J. Shim, P. Scheuermann, and R. Vingralek, "Proxy Cache Algorithms: Design, Implementation, and Performance," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, pp. 549-562, 1999.

[16] A. R. Sarma and R. Govindarajan, "An Efficient Web Cache Replacement Policy," *Proceedings of HiPC*, 2003.

[17] N. Chand, R. Joshi, and M. Misra, "Data Profit Based Cache Replacement in Mobile Environment," *Proceedings of Wireless and Optical Communications Networks*, 2006.

[18] J. Xu, Q. Hu, W.C. Lee, and D. L. Lee, "Performance Evaluation of An Optimal Cache Replacement Policy for Wireless Data Dissemination," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, pp. 125-139, 2004.

[19] L. Yin, G. Cao, and Y. Cai, "A Generalized Target-Driven Cache Replacement Policy for Mobile Environments," *Proceedings of*

IEEE/IPSJ International Symposium on Applications and the Internet, 2003.

[20] K. Kim and D. Park, "Least Popularity-Per-Byte Replacement Algorithm for A Proxy Cache," *Proceedings of Parallel and Distributed Systems*, 2001.

[21] K. Li, T. Nanya, and W. Qu, "A minimal access cost-based multimedia object replacement algorithm," *Proceedings of Parallel and Distributed Processing Symposium*, 2007.

[22] P. Triantafillou and I. Aekaterinidis, "Web Proxy Cache Replacement: Do's, Don'ts, and Expectations," *Proceedings of International Symposium on Network Computing and Applications*, 2003.

[23] Y. Chen, Z.-Z. Li, and Z.-W. Wang, "A GA-based Cache Replacement Policy," *Proceedings of Machine Learning and Cybernetics*, 2004.

[24] K. Li; H. Shen, "An Improved GreedyDual Cache Document Replacement Algorithm," *Proceedings of Web Intelligence 2004 IEEE/WIC/ACM International Conference*, 2004.

[25] H. Wang; J. Peng; Y. Wu; H. Feng, "SzLFU(k) Web cache replacement algorithm," *Proceedings of IEEE Region 10th Conference on Computers, Communications, Control and Power Engineering*, 2002.

[26] H. ElAarag and S. Romano, "Improvement of The Neural Network Proxy Cache Replacement Strategy," *Proceedings of the Spring Simulation Multiconference*, 2009.

[27] J.-C. Bolot and P. Hoschka, "Performance Engineering of The World Wide Web: Application to Dimensioning and Cache Design," *Computer Networks and ISDN Systems*, Vol. 28, pp. 1397-1405, 1996.

[28] A. Balamash and M. Krunz, "An Overview of Web Caching Replacement Algorithms," *IEEE Communications Surveys and Tutorials*, Vol. 6, No. 2, pp.44–56, 2004.

[29] T. Banditwattanawong, S. Hidaka, H. Washizaki, and K. Maruyama, "Optimization of Program Loading by Object Class Clustering," *IEEJ Transactions on Electrical and Electronic Engineering*, Vol. 1, No. 4, pp. 397–407, 2006.

[30] National Laboratory for Applied Network Research. (2012). *Weekly Squid HTTP Access Logs* [Online]. Available: http://www.ircache.net/

[31] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. (1999).*Hypertext Transfer Protocol - HTTP/1.1* [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

Thepparit Banditwattanawong received B.E. from King Mongkut's Institute of Technology Ladkrabang, Thailand and M.E. from Asian Institute of Technology, Thailand. He obtained Ph.D. in Informatics from The Graduate University for Advanced Studies, Japan. He was a founder and leader of a national project Thailands IPv6 testbed at National Electronics and Computer Technology Center. He is currently an assistant professor and an assistant director of Ph.D.IT. program at Sripatum university. His main areas of research interests include cloud computing and distributed computing.

Putchong Uthayopas received his bachelor and master degree in electrical engineering from Chulalongkorn University in 1984 and 1988. He received master and PhD in computer engineering from University of Louisiana in 1994 and 1996 accordingly. His research interest is in cluster computing, grid and cloud computing system and tools. He published more than 130 refereed publication in conferences and Journals. Putchong Uthayopas is a co-founder of the Thai National Grid project. In 2012, he received a distinguish computer engineer award in system integration from the Engineering Institute of Thailand.