V-Cache: Towards Flexible Resource Provisioning for Multi-tier Applications in IaaS Clouds

Yanfei Guo, Palden Lama, Jia Rao, Xiaobo Zhou
Department of Computer Science
University of Colorado, Colorado Springs, USA
Email addresses: {yguo, plama, jrao, xzhou}@uccs.edu

***Abstract*—Although the resource elasticity offered by Infrastructure-as-a-Service (IaaS) clouds opens up opportunities for elastic application performance, it also poses challenges to application management. Cluster applications, such as multi-tier websites, further complicates the management requiring not only accurate capacity planning but also proper partitioning of the resources into a number of virtual machines. Instead of burdening cloud users with complex management, we move the task of determining the optimal resource configuration for cluster applications to cloud providers. We find that a structural reorganization of multi-tier websites, by adding a caching tier which runs on resources debited from the original resource budget, significantly boosts application performance and reduces resource usage. We propose V-Cache, a machine learning based approach to flexible provisioning of resources for multi-tier applications in clouds. V-Cache transparently places a caching proxy in front of the application. It uses a genetic algorithm to identify the incoming requests that benefit most from caching and dynamically resizes the cache space to accommodate these requests. We develop a reinforcement learning algorithm to optimally allocate the remaining capacity to other tiers. We have implemented V-Cache on a VMware-based cloud testbed. Experiment results with the RUBiS and WikiBench benchmarks show that V-Cache outperforms a representative capacity management scheme and a cloud-cache based resource provisioning approach by at least 15% in performance, and achieves at least 11% and 21% savings on CPU and memory resources, respectively.**

I. Introduction

As an important cloud offering, the Infrastructure-as-a-Service (IaaS) cloud provides users with bundles of hardware resources in the form of virtual machines (VMs). Although users have the flexibility in deciding how much resource to rent, it is often difficult for them to translate their performance and cost goals into the corresponding resource requirements [18]. Cluster applications, such as multi-tier websites and data analytics, place an additional burden on users requiring not only accurate capacity planning but also proper partitioning of the resources into a number of VMs [17].

While the choice in resource partitioning (e.g., the cluster size and the type of resources) under a certain resource budget is critical to the performance of user applications, it is also important to cloud providers. Mis-configured user clusters may incur excessive access (usually from a subset of the VMs) to the cloud hardware, such as processors and I/O devices, creating significant interferences to the co-running applications. Besides the impact on the overall Quality-of-Service the cloud infrastructure provides, improperly configured virtual clusters also increase the complexity of cloud management and reduce the utilization of data center hardware. For example, cloud providers may need to employ more complex algorithms consolidating such clusters onto physical machines and are likely to be conservative when over-commiting cloud resources [32].

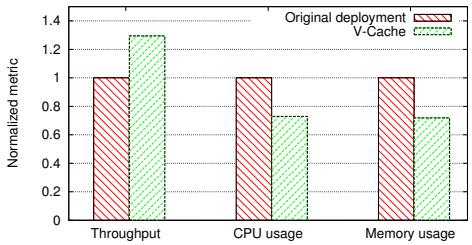


Figure 1. Resource re-partitioning by V-Cache improves application-level performance and reduces resource usage.

We argue that even when users explicitly request a certain amount of cloud resources, cloud providers should still be able to flexibly provision these resources as long as the alternative provisioning also meets users' performance goals. With flexible provisioning, providers can arrange the resource allocation in a way that is most suitable for the cloud platform and co-running applications. As a result, performance interferences might be mitigated avoiding expensive cross-machine migrations and the consolidation ratio could be increased. There is existing work focusing on dynamic provisioning of cloud resources according to varying application demands [13], [21], [29], [30]. However, such dynamic provisioning requires new cloud mechanisms for fine-grained resource metering and charging [3]. A viable approach under the current cloud pricing scheme (i.e., fixed rate charged on a hourly basis) is to re-organize the resources a client requests for possible optimizations, while keeping the total amount of resources allocated to this client unchanged.

We show that a proper partitioning of multi-tier application resources benefits both cloud users and providers. Figure 1 draws the performance and resource consumptions of the RUBiS benchmark when a total capacity of 4 GHz CPU and 4 GB memory resources were partitioned into tiers differently. The *original deployment* refers to the typical three-tier deployment of RUBiS (i.e., includes a web server, application server, and database server). We carefully adjusted the resources allocated

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.12

on each tier to maximize the effective system throughput [14]. Alternatively, in *V-Cache*, we created a cache VM using part of the total resources and placed it in front of the web tier for storing the recently accessed web content. The rest of the capacity were used to host other three tiers. Figure 1 shows that the re-partitioning of resources, especially the use of the caching tier, significantly boosted application performance by 29.5%, and reduced CPU and memory usage by 26.1% and 28.2%, respectively.

The motivational example suggests that a structural reorganization of multi-tier applications with a cache tier may considerably optimize performance and resource usage, even compared with the finely-tuned provisioning scheme. Caching has been widely used in multi-tier websites for reducing network bandwidth and access latency [9], [24]. If properly configured, such cache servers can be made transparent to clients and require no modifications to the multi-tier website. Therefore, cloud providers can take advantage of such flexibility to transparently integrate a caching tier into the multi-tier application and to add, remove, and resize the tier for better performance and resource efficiency.

However, the determination of the optimal resource allocation for each tier, in particular the caching tier, is not trivial. First, the workload of multi-tier websites does not always benefit from the caching tier. While requests for static content get most speedup, the ones for dynamic content can hardly get help from caching. This requires a mechanism that analyses request processing cost and redirects the requests with potential speedups to the cache. Second, placing a caching tier in front of a multi-tier website significantly changes the intensity and pattern of the traffic at each tier, making the modeling of tiers difficult. Finally, the time-varying Internet traffic requires that the provisioning of multi-tier websites be dynamically adjusted to match the actual application demands.

These challenges motivated us to develop an automated and adaptive approach to partitioning and allocating resources for multi-tier applications in clouds. In this paper, we present V-Cache, a machine learning based approach that flexibly manages the resources of multi-tier websites to improve application performance and reduce resource usage. More specifically, we make the following contributions:

1) **Transparent request interception and clustering.** We transparently intercept the requests coming to the multi-tier website and use an adaptive fuzzification-based approach to cluster the requests into groups. We characterize each request cluster by its request type, processing cost, and cache hit rate, and then use the information to determine which groups of requests should be serviced by the caching tier.
2) **Cost-aware selective caching and cache resizing.** Based on the request clusters and their associated costs for in and out of cache request processing, we identify the requests that benefit most from caching. We develop a heuristic-based genetic algorithm to accelerate the request selection process and propose a method to predict the required cache size according to current traffic flow.
3) **Self-adaptive tier capacity management** We develop a reinforcement learning approach to optimally allocate the remaining capacity to all the tiers taking into consideration of the multi-tier application performance.
4) **Design and implementation of V-Cache.** We design and implement a prototype of V-Cache in our university cloud testbed. Experimental results on two representative multi-tier benchmarks show that V-Cache achieved significant performance improvement and resource savings compared with a typical 3-tier deployment without caching. It also outperformed a recently proposed virtual cache management scheme in both static and dynamic workloads [10]. We further show that additional resource savings could be achieved by sharing a unified caching tier among multiple websites.

The rest of this paper is organized as follows. Section II discusses the request processing of multi-tier applications and gives motivational examples on the use of caching. Section III and Section IV elaborate the key designs and implementation of V-Cache, respectively. Section V gives experimental results. Related work is presented in Section VI. We conclude this paper and discuss future works in Section VII.

II. Background and Motivation

We first discuss the architecture of multi-tier applications and how a caching tier boosts performance and reduces resource usage. Then, we show that the characteristics of the workload affect the effectiveness of caching and give the motivation for the cost-aware request caching.

In general, multi-tier applications consist of three tiers, including the web tier, the application tier, and the data tier. More specifically, in a multi-tier website, the web server (e.g., Apache httpd) presents page contents in a web-based interface to client browsers; the application server (e.g., Tomcat) implements the business logic and functionalities; the database server (e.g., MySQL) maintains information pertaining to the web service. The content delivered by multi-tier websites includes both static and dynamic contents. The static content is the information that can be viewed and shared by all users, such as cascading style sheets (CSS), javascripts, and images. The dynamic content is the user-specific information generated at the time users request the page including user profiles, member-only pages and database-driven HTML content.

Caching is a widely used approach to accelerating content delivery. There are multiple places that cache servers can be deployed in the three-tier architecture. A HTTP reverse proxy (e.g., Varnish) can sit in front of the web tier to cache complete page contents. Alternatively, an in-memory key-value store (e.g., memcached) can be deployed between the application and database tiers to cache only database query results. If properly configured, both approaches offload client requests from the website and provide lower response times. Since the integration of an in-memory data store into multi-tier websites requires necessary modifications to the application logic, it can hardly be made transparent to the owner of the website. In contrast, a reverse proxy is able to transparently serve clients

Table I
REQUEST PROCESSING COST WITH DIFFERENT RESOURCE ALLOCATION.

Request URI	Access frequency	Request type	Processing cost on cache	Processing cost on application		Miss rate
				Balanced	Optimized	
/index.html	0.35	Static	$43\,ms$	$221\,ms$	$46\,ms$	4%
/logo.jpg	0.30	Static	$21\,ms$	$140\,ms$	$22\,ms$	5%
/BrowseRegions.php	0.10	Dynamic	$40\,ms$	$583\,ms$	$543\,ms$	14%
/SearchItemByRegions.php	0.12	Dynamic	$74\,ms$	$2676\,ms$	$2458\,ms$	51%
/ViewItems.php	0.13	Dynamic	$58\,ms$	$620\,ms$	$586\,ms$	19%

on behalf of the multi-tier website through traffic interception. Therefore, adding a reverse proxy is a viable approach to restructuring multi-tier applications in clouds.

However, the determination of the optimal cache server configuration is not trivial and it depends on the characteristics of workloads. To study the effectiveness of caching in different scenarios, we created a controlled testing environment. We deployed the RUBiS benchmark in four VMware VMs. The VMs hosted the Varnish proxy, Apache web server, Tomcat application server, and MySQL database server, respectively. The total resource budget for running the benchmark was 4 GHz CPU and 4 GB memory shared by all VMs. We exercised the benchmark with 2000 concurrent clients.

Table I lists the individual request types of the RUBiS benchmark and their associated processing costs (latency) when served from a cache server or directly from the multi-tier application. When measuring the individual cost, we issued the corresponding request at a speed matching its access frequency in RUBiS. As such, we measured the cost for each request type assuming an infinite cache. For the measurement on the multi-tier application, we show the process costs of two schemes. *Balanced* refers to the configuration suitable for typical workloads. *Optimized* refers to the resource allocation tuned for a specific request. For instance, we allocated sufficient resources to the web tier for request `index.html` and put more resources on the application and database tiers for request `BrowseRegions.php`.

We make three important observations from Table I. First, the use of a cache server can significantly accelerate request processing. Second, caching is mostly effective for static content with an average miss rate less than 5%. In contrast, cached dynamic contents incur high miss rate due to the expiration of their time-to-live (TTL) values, after which cached requests are invalidated. Finally, although caching is effective for static content, it is possible to achieve similar performance by carefully configuring the multi-tier website. For dynamic content, the improvement due to caching could be an order of magnitude over the best possible multi-tier settings.

Next, we study the impact of cache size and caching policy on application performance and the combined resource usage of all tiers. We measured the effective throughput (see Section III-F for definition), CPU, and memory consumption of the multi-tier website under RUBiS browsing workload. Note that the overall resource usage includes resources consumed by the caching tier and all other three tiers. Since the browsing workload contains read-mostly requests and largely static content, we set to cache static content whenever possible and cache dynamic content only if there is available space in the cache. Figure 2 shows the performance and resource usage of RUBiS with different cache sizes. In Figure 2, we can see that caching significantly boosts the application effective throughput compared with the no-cache setting. As the size of the cache increases, application throughput increases accordingly until the improvement slows down at the cache size of 16 MB.

Figure 2(b) and Figure 2(c) show that caching effectively offloads requests from the multi-tier website resulting in reduced overall resource consumption. However, when application throughput stagnates, further increments in the cache size incur steep increments in both CPU and memory usages. An examination of the cache log reveals that when the cache size increased, the cache server began to cache dynamic content. Since such dynamic contents have short TTLs, the resources required to handle these requests were mostly wasted due to high cache miss rates.

In the last test, we fixed the cache size at 16 MB and compared four caching policies. *Static first* and *Dynamic first* prioritize static content and dynamic content at the time of caching, respectively. Least frequently used *(LFU)* and Least frequently and costly used *(LFCU_K)* [5] uniformly cache both static and dynamic contents and evict the least frequently used request. *LFCU_K* also considers the miss penalty (i.e., the processing cost on the application) and selects the least costly content for eviction. Figure 3 shows that *Static first* achieves the best performance and the least resource usage among the four policies because the workload contains a large amount of static content. *LFU* does not take the processing cost into consideration. Both *Dynamic first* and *LFCU_K* favor requests with high miss penalties but fail to consider the high miss rate of dynamic content. Another issue with caching policies based on replacement algorithms is that such policies are only effective when the cache is full and do not handle request expirations.

In summary, the determination of the optimal cache configuration for multi-tier websites has unique requirements. (1) Requests should be selectively served in the cache tier considering the processing cost of individual requests and the coordination with the other tiers. (2) The cache tier size

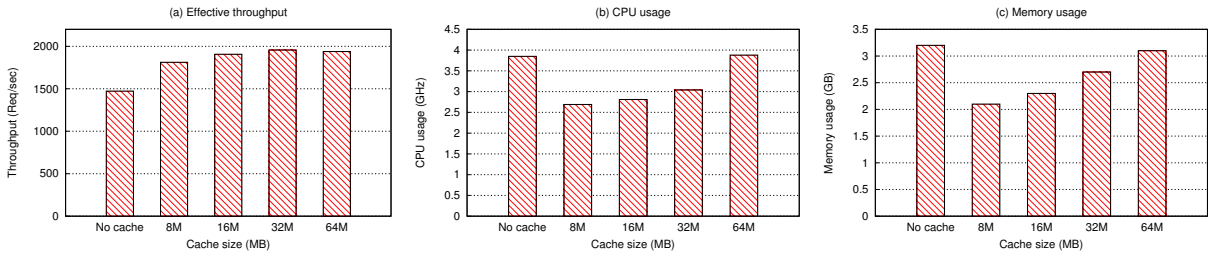


Figure 2. The impact of cache size on (a) application effective throughput, (b) CPU usage, and (c) memory usage.

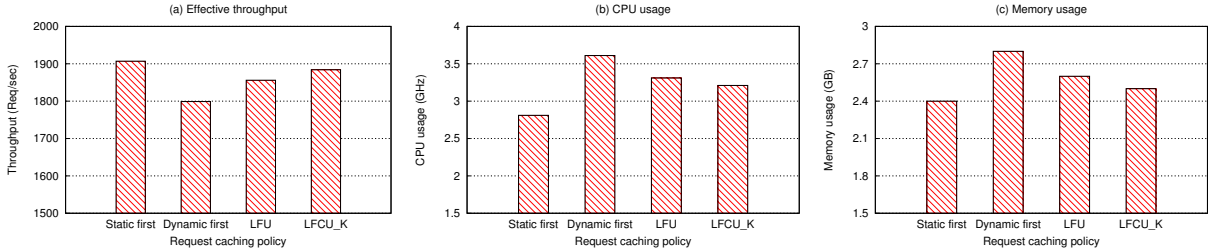


Figure 3. The impact of caching policy on (a) application effective throughput, (b) CPU usage, and (c) memory usage.

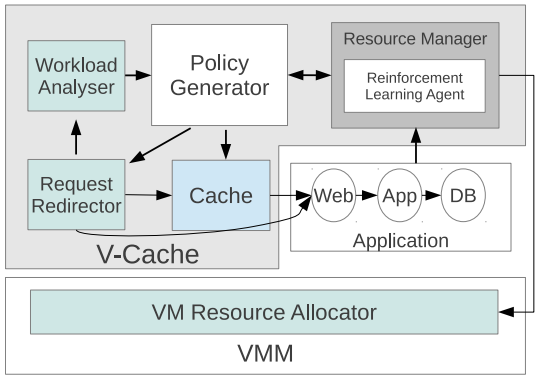


Figure 4. The architecture of V-Cache.

should be properly configured. Under-provisioned or over-provisioned cache spaces result in either performance penalties or excessive resource consumptions. (3) To achieve the minimum resource usage and the maximum cache performance, selective caching is more desirable than selective eviction as the latter inevitably incurs performance penalties and resource wastes for short-lived requests. These challenges motivated us to develop V-Cache, an automated approach to cost-aware selective caching, dynamic cache resizing, and coordinated tier capacity management.

III. The Design of V-Cache

A. Overview

We design V-Cache as a set of standalone daemons residing in a management VM. Figure 4 shows the architecture of V-Cache.

The *workload analyser* transparently intercepts the incoming traffic and performs clustering of the requests based on their request types, content sizes, and the processing costs. Besides clustering the requests, the workload analyser also maintains the statistics of completed requests, such as the response time and the cache hit rate. Such statistics are used by the policy generator and the request redirector to determine and apply selective caching.

The core of V-Cache is the design of the *policy generator*, which identifies the set of requests that benefit most from caching. The policy generator takes the request clusters as input and outputs a request redirection map to the request redirector. Based on the selected requests for caching, the policy generator also determines the minimum size of the cache to accommodate these requests.

The *request redirector* is essentially a web proxy server intercepting all incoming traffic. Based on the redirection map provided by the policy generator, the request redirector examines the `URI` and `Host` fields of a request's HTTP header. If the request falls in the cluster that is mapped to the cache server, the request redirector forwards it to the caching tier. Otherwise, the request is sent to the web tier of the multi-tier website bypassing the caching tier.

Once the request redirection map and the cache size are determined, the resources used by the caching tier are debited from the total resource budget. The *resource manager* allocates the remaining resources to all the tiers considering the overall performance of the multi-tier website. Note that the memory size of the caching tier is determined by the policy generator, but the CPU allocation is managed by the resource manager.

In the following, we elaborate the design of each part of V-Cache.

B. Request Clustering

Request clustering identifies requests that are essentially similar in their processing costs in and out of cache. Since requests for static content and dynamic content have distinct characteristics, we perform clustering for such requests separately. The request analyser classifies the requests into two groups, static and dynamic, according to their URIs. This ensures that static content and dynamic content are treated differently when deciding the caching policy. We perform the clustering of requests based on the size of the requested content.

We use an adaptive fuzzification based clustering approach, which is derived from the structure learning in neural fuzzy control [20]. Unlike clustering algorithms like k-means, it does not need a specified total number of clusters. This avoids the misleading information of the total number of clusters. In the clustering, we use Gaussian function to define different clusters. The Gaussian function can recognize a range of requested content size, which is determined by the mean m_i and the standard deviation σ_i. When a new request comes in, the algorithm checks its requested content size s with all existing clusters. If the content size of a request is recognized by a cluster, the request will be added to that cluster. If the content size of the request is not recognized by any cluster, a new cluster will be added. We assign the mean of new cluster m_i^{new} to request content size s_i and the standard deviation σ_i^{new} to a predefined value. The Gaussian function is defined as

$$u_i = \exp\left(-\frac{(s-m_i)^2}{\sigma_i^2}\right). \tag{1}$$

The newly generated cluster could be similar to an existing one. To eliminate it, we perform a similarity check before adopting the cluster as a part of neural network. The similarity measurement method was originally proposed in [25]. Suppose $u_A(x)$ and $u_B(x)$ are two Gaussian functions and their means and standard deviations are m_A, m_B, σ_A, σ_B, respectively. The similarity of two Gaussian functions is measured as:

$$E(A,B) = \frac{|A \cap B|}{\sigma_A\sqrt{\pi} + \sigma_B\sqrt{\pi} - |A \cap B|}.$$

Assuming $m_A \geq m_B$,

$$\begin{aligned}|A \cap B| = &\frac{1}{2}\frac{h^2(m_B - m_A + \sqrt{\pi}(\sigma_A + \sigma_B))}{\sqrt{\pi}(\sigma_A + \sigma_B)} \\ &+ \frac{1}{2}\frac{h^2(m_B - m_A + \sqrt{\pi}(\sigma_A - \sigma_B))}{\sqrt{\pi}(\sigma_B - \sigma_A)} \\ &+ \frac{1}{2}\frac{h^2(m_B - m_A - \sqrt{\pi}(\sigma_A - \sigma_B))}{\sqrt{\pi}(\sigma_A - \sigma_B)}\end{aligned}$$

where $h(x) = max(0, x)$. In the case of scenario $\sigma_A = \sigma_B$,

$$(|A \bigcap B|) = \frac{1}{2}\frac{h^2(m_B - m_A + \sqrt{\pi}(\sigma_A + \sigma_B))}{\sqrt{\pi}(\sigma_A + \sigma_B)}. \tag{2}$$

Only when the measured similarity is smaller than a predefined threshold, the new cluster is added to the clusters.

Then the request analyser samples a small portion of requests from each cluster. From these samples, the analyser obtains the request processing costs of each cluster in and out of cache.

C. The Optimal Caching Policy

The optimal caching policy is the mapping of requests to the cache or to the multi-tier application that incurs the least total processing cost for all requests. For n request clusters, each of which can be either processed in or out of the cache, there are 2^n possible request mappings. For example, suppose the workload has n request clusters. A request mapping M redirects request cluster 1 to p to the cache and sends request cluster $p+1$ to n directly to the application. We define the cost of the request mapping M as

$$y = \sum_{i=1}^{p}[c_i n_i h_i + a_i n_i (1-h_i)] + \sum_{j=p+1}^{n} a_j n_j \tag{3}$$

where c_i and a_i are the processing costs of request cluster i on cache and application, respectively. n_i is the number of requests in cluster i. h_i is the hit rate of cluster i. By enumerating all 2^n possible mappings, one is able to find the optimal request mapping that has the least overall processing cost.

D. Policy Generation: A Genetic Algorithm-based Approach

As the number of request clusters increases, the time required to compute the optimal caching policy grows exponentially. It is prohibitively expensive to enumerate all the request mappings, especially when the policy needs to be computed online. Genetic algorithms are a set of methods that are used to accelerate the process of optimization and searching. Such algorithms rely on a search heuristic to find near-optimal solutions. We formulate the search of the optimal caching policy as a genetic algorithm.

More specifically, we represent one request mapping strategy set as a chromosome. The structure of the chromosome for n request clusters is represented by a chromosome as $\mathbf{x}$

$$\mathbf{x} : \{l_1, l_2, \cdots, l_n\}$$

where each segment l_i represents whether this request cluster will be forwarded to the cache or to the application.

The V-Cache randomly generates N chromosomes to form the initial population P_1. The population evolves by forming a child population P_{t+1} from the parent population P_t. This emulates the natural evolution process and assumes that some of the new populations could be better than the old ones. Chromosomes are selected to form new solutions by their fitness - those chromosomes with higher fitness values have higher chance to be selected. The evolution process is done by iteratively creating child populations based on the current parent population P_t. The child population Q_t is created through two steps: crossover and mutation.

The genetic algorithm first picks two chromosomes $\mathbf{x}$ and $\mathbf{y}$ from P_t based on their fitness values. The crossover procedure is to generate offsprings by swapping the codes of $\mathbf{x}$ and $\mathbf{y}$ at

random locus k (point in chromosome) as shown in Eq. (4). The child chromosomes $\mathbf{x}'$ and $\mathbf{y}'$ are added to the child population Q_t.

$$\left.\begin{array}{l}\mathbf{x}:\{l_1,l_2,\cdots,l_n\}\\ \mathbf{y}:\{m_1,m_2,\cdots,m_n\}\end{array}\right\} \xrightarrow{crossover} \left\{\begin{array}{l}\mathbf{x}':\{l_1,l_2,\cdots,m_k,m_{k+1},\cdots,m_n\}\\ \mathbf{y}':\{m_1,m_2,\cdots,l_k,l_{k+1},\cdots,l_n\}\end{array}\right. \tag{4}$$

For each chromosome $\mathbf{x}' \in Q_t$, the algorithm randomly changes the code at one locus (highlighted in Eq. (5) and creates a mutated chromosome $\mathbf{x}''$. The $\mathbf{x}'$ is randomly picked from Q_t with mutation rate ϕ.

$$\begin{array}{l}\mathbf{x}':\{l_1,l_2,\cdots,m_k,\mathbf{m}_{k+1},\cdots,m_n\}\\ \xrightarrow{mutate} \mathbf{x}'':\{l_1,l_2,\cdots,m_k,\mathbf{m'}_{k+1},\cdots,m_n\}\end{array} \tag{5}$$

After the new chromosomes are generated, the algorithm calculates the fitness of each chromosome. The fitness of a chromosome is the indicator of how good the given solution (represented as the chromosome) is to the optimization problem. The objective of the policy generation is to minimize the overall processing cost. The definition of the cost function is given in Eq. (3).

The algorithm will then select chromosomes from Q_t to build the parent population P_{t+1} for the next iteration. However, due to the randomness of crossover and the mutation, the child chromosomes in child population Q_t are not guaranteed to be better (*i.e.* have lower cost) than their parents. We use the tournament selection algorithm [27] for the building of P_{t+1}. The tournament algorithm creates a comparing set C by randomly picking M chromosomes from the parent population P_t. Each child chromosome in Q_t is compared with the chromosomes in the comparing set C. Only those child chromosomes that have lower cost than all the competitors in C will be put in P_{t+1}. This drives the genetic algorithm to improve the fitness during each evolution.

By iterating this evolution process, we can find a near-optimal solution for the policy generation problem.

E. Cache Size Determination

In V-Cache, each cached request occupies one cache block and the block size is set to the size of the largest content. Therefore, the cache size is determined by the content sizes and the number of requests stored in cache. Based on the request mapping, we determine the number of requests going to the cache server. We obtain the requested content sizes of different requests from the workload analyser. Then the cache size is calculated as

$$s_{cache} = max(S_i)\sum_{i=i}^{p} d_i$$

where D_i is the number of requested contents and S_i is the size of requested content in request cluster i, respectively. The size of the cache is determined by the maximum size of the contents multiplies the number of contents.

F. Self-adaptive Tier Capacity Management

The self-adaptive tier capacity management controls the allocation of the remaining resources to all tier VMs. As discussed in [31], it is hard to obtain an accurate system model for resource allocation due to the dynamics workload and the complexity introduced by caching. The reinforcement learning is a process of learning through interactions with an external environment. It does not assume any knowledge of the system it works in and thus does not require any model of the underlying system. The tier capacity management problem of application's VMs can be formulated as a finite Markov decision process. It consists of a set of states and several actions for each state. During the transition of each state, the learning agent perceives the reward defined by a reward function $r(s,a)$. The goal of the reinforcement learning agent is to develop the policy $\pi : S \rightarrow A$, which can maximize the cumulative rewards through iterative trial-and-error interactions. We use Q-Learning agent in the reinforcement learning approach.

In the resource allocation of the application's VMs, we define the state space S as the set of possible resource allocations for each VM. For an application that has n VMs, the state space (S) is represented as a collection of state vectors (s):

$$s = [r_{11}, r_{12}, \cdots, r_{ni}].$$

The elements in the state vector are resource allocations, in which i is the number of the resource types. In V-Cache, we only control the CPU and the memory resources.

The action for each state element is represented as a vector. We define three actions for each state element: keep, increase, and decrease. Hence, an action vector can be $(1,0,0)$, $(0,1,0)$, or $(0,0,1)$. For example, $(1,0,0)$ means to keep the current value of one state element. The action set (A) is represented as a collection of action vectors (a):

$$a = [a_{r_{11}}, a_{r_{12}}, \cdots, a_{r_{ni}}].$$

We use one Q-Learning agent to control the effective system throughput. The Q-Learning agent uses a Q-Table to determine the action choice on each state. The Q-Table stores the Q-Value for each state-action pair. The learning process will continuously update the Q-Values based on the reward it receives. The key to the design of a RL algorithm is to define the reward signal that reflects the high-level objective. We use the effective throughput to measure the goodness of a start-action pair. The effective throughput is defined as the number of requests that meet the Service-Level Objectives (SLOs) of the application response time [14]. The reward function of an action a on state s in k_{th} time slot is defined using the effective throughput (ET) and the normalized effective throughput (NET):

$$r(s_k, a_k) = \beta |ET_{s_k,a_k} - ET_{s_{k-1},a_{k-1}}| \tag{6}$$

where $\beta = NET_{s_k,a_k} - NET_{s_{k-1},a_{k-1}}$. It uses the change in normalized effective throughput β as the correction factor. For

example, if the effective throughput has increased significantly and the normalized effective throughput has almost no change, it implies that the change in the effective throughput could be just due to the variance of the number of incoming requests. This phenomenon indicates that the resource allocation of the application's VMs is favorable for the present workload and no significant update should be applied to the reinforcement learning. With the reward function, the Q-Value of an action a on state vector s in k_{th} time slot is updated as $Q(s_k, a_k)$. It is refined by

$$Q(s_k, a_k) = Q(s_k, a_k) + \varepsilon[r(s_k, a_k) - Q(s_k, a_k) + \gamma Q(s_{k+1}, a_{k+1})] \quad (7)$$

where ε is the learning rate and γ is the discount rate to guarantee that the accumulated reward converges in continuing tasks. We apply a variable learning rate for fast convergence. The learning rate for different state-action pair is defined as

$$\varepsilon = \frac{\mu}{N(s, a)} \quad (8)$$

where μ is a given constant and $N(s, a)$ is the number of times that state-action pair (s,a) has been visited. The future action a_{k+1} is determined using *greedy* policy.

IV. Implementation

We built a testbed in a university prototype data center, which consists of five Dell PowerEdge R610 servers and two Dell PowerEdge R810 servers. Totally, they have 10 Intel 6-core Xeon X5650 CPUs, 8 Intel 6-core E7540 CPUs, and 704 GB memory. The servers are connected with 10 Gbps Ethernet. VMware vSphere 5.0 is used for server virtualization.

As many others [7], [14], [23], [31], [34], [36], we use RUBiS [6] as the benchmark application in conducting the experiments. RUBiS is an open source multi-tier Internet benchmark application. It emulates three different categories of workload at different concurrency levels. It provides flexibility allowing us to evaluate V-Cache under different workloads and conduct sensitivity analysis. We also use WikiBench [2], [33], a multi-tier benchmark application based on the real data and workload trace of Wikipedia.

For a multi-tier application, we allocate three VMs to host the Apache web server, PHP application server, and MySQL database server, respectively. The maximum capacity of each VM is up to 2 GHz CPU and 2 GB memory. V-Cache uses a dedicated VM. It debits resources from the application's VMs by coordinated cache and application resizing. All VMs use Ubuntu server 10.04 with Linux kernel 2.6.35.

V-Cache deploys Varnish Cache 3.0.1, an open-source web cache application [1]. Its request redirector intercepts all incoming requests and recognizes each request based on its `URI` field and `Host` field in the HTTP header. The request is forwarded to the cache or the application based on the request mapping generated by V-Cache's policy generator.

V. Performance Evaluation

Using the RUBiS benchmark application, we first present the application performance improvement and resource utilization efficiency due to V-Cache. We then demonstrate the impact of employing a shared V-Cache for multiple multi-tier applications. We further study the performance impact of the cost-aware request redirection. In addition, we evaluate the performance of V-Cache with the WikiBench application that uses real data and workload trace of Wikipedia.

A. Performance of V-Cache

1) Impact on the Effective Throughput: For performance comparison, we implemented a representative cache-based resource provisioning approach. Elastic cloud cache (ECC) [10] is a distributed cache system designed for cloud environments. It provides an elastic cache by scaling up and down the number of cache nodes to deal with workload variations. We implemented it for cache VM resizing. As ECC does not consider resizing application VMs, we tailored the approach with a reinforcement learning based application VM resizing.

We also implemented a finely-tuned resource provisioning approach, VCONF [31], which is a reinforcement learning based approach for automated configuration of VMs. VCONF does not consider using a cache tier.

The application performance metric is the effective system throughput [14]. The soft time bound and the hard time bound are set to $1000\,ms$ and $1500\,ms$, respectively. We apply one stationary workload that emulates 6000 concurrent users. We also apply a bursty workload with 6000 concurrent users, using the approach proposed by Mi *et al.* [26] that changes the think time of each user.

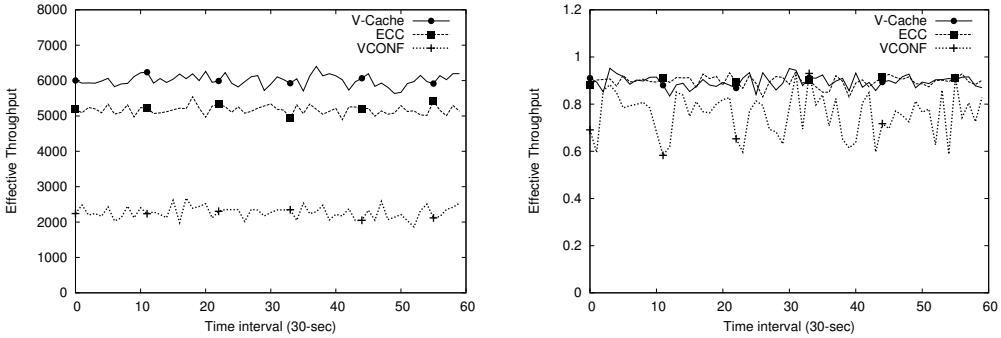
Figures 5(a) and 5(b) show the effective throughput due to the three different approaches under the stationary workload and the bursty workload, respectively. Note that in the experiment, three approaches use the same total amount of CPU and memory resources.

Figure 5(a) shows that V-Cache and ECC outperform VCONF by 3 times and 2.6 times in the effective throughput under the stationary workload. This is due to the use of an elastic cache VM. Due to the request clustering and cost-aware request redirection, V-Cache also achieves 15.4% higher effective throughput than ECC does.

Under the bursty workload, we normalize the effective throughput by the number of incoming requests. Figure 5(b) shows that V-Cache and ECC achieve higher normalized effective throughput than VCONF, but also more stable application performance. These results demonstrate that V-Cache is able to significantly improve the processing capability of a multi-tier application with the same overall resource capacity.

2) Impact on the Resource Utilization Efficiency: We compare the resource utilization efficiency of V-Cache, ECC and VCONF under a stationary workload scenario. The RUBiS workload was set to emulate 2000 concurrent users, each with a mean think time of 7 seconds. This is a relatively light workload according to the overall resource availability. V-Cache, ECC and VCONF all obtain very close effective throughput. However, they consume different amount of resources.

Figure 6 shows the average resource allocation due to the three different approaches. We observe that V-Cache uses 27%



(a) Stationary workload. (b) Bursty workload.

Figure 5. Performance of a multi-tier application using V-Cache, ECC, and VCONF.

less CPU and 38% less memory resources than VCONF does. Compared to ECC that is also based on a dedicated cache VM, V-Cache uses 11% less CPU and 21% less memory resources. The results demonstrate that V-Cache is able to significantly improve the resource utilization efficiency.

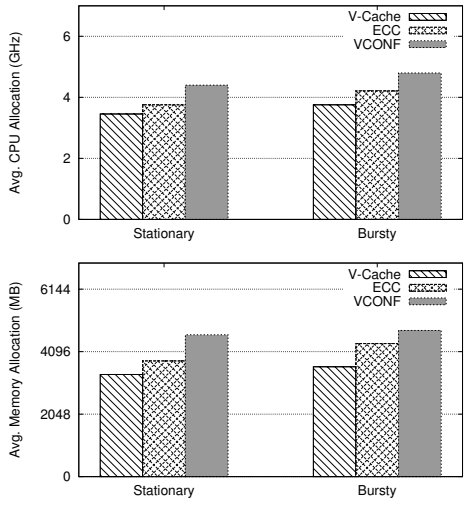


Figure 6. Resource allocation under a light workload.

We further study the resource utilization efficiency of the three approaches under a highly dynamic workload. We instrumented the workload generator of RUBiS to change the number of concurrent users at the runtime. Figure 7 plots the normalized effective throughput in a 60-minute period with a 30-second sampling interval. The dynamic workload starts with 2000 concurrent users. The number of concurrent users is changed at the 10_{th}, 30_{th}, 50_{th}, 70_{th}, 90_{th}, and 110_{th} intervals to 3000, 4000, 5000, 3500, 1000, 2000, respectively. We observe that V-Cache achieves the highest effective throughput. This is due to the advantages of using a flexible cache VM with request clustering. Further, the cost-aware request redirection technique enables V-Cache to utilize resources in a most efficient way. VCONF is of significant performance

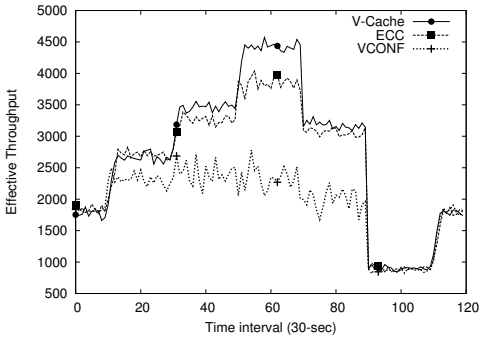


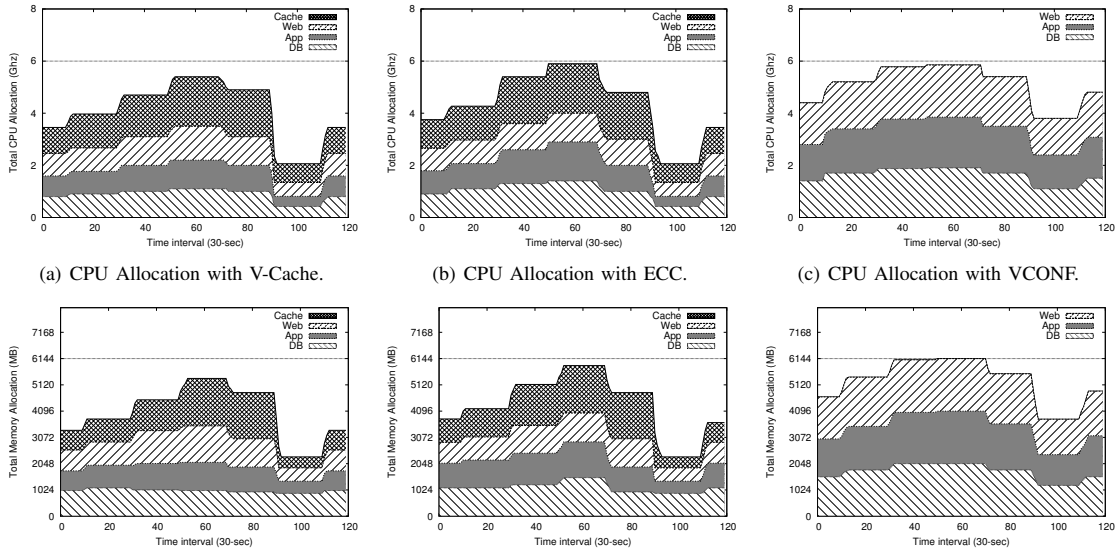
Figure 7. Performance comparison of V-Cache, ECC, and VCONF under a highly dynamic workload.

degradation when the workload increases. Without an elastic cache VM, the application's VMs are easily overwhelmed by the significantly increased workload.

To better understand the resource allocation of the three approaches under the dynamic workload, we illustrate the CPU and memory allocation of VMs at individual tiers due to each approach in Figure 8. Results show that all three approaches are able to dynamically change the VM resource allocations to meet the workload variations.

Compared with ECC, V-Cache not only achieves better performance but also reduces overall resource consumption. Figure 8(a) and Figure 8(b) show that ECC allocates more CPU resource to the cache VM. It implies that the cache VM in ECC processes more requests than the cache VM in V-Cache. Instead, V-Cache allocates more CPU resource to the web tier and less CPU resource to the application tier and the database tier. This is due to the cost-aware request direction of V-Cache. It is able to allow the application VMs to process low-cost requests and save the cache VM's capacity for high-cost requests. It significantly reduces the resource allocation to the cache VM, application VM, the database VM, and the overall resource consumption in the system.

From Figure 8(d) and Figure 8(e), we observe a similar trend in the memory allocation as to the CPU allocation. Note



(a) CPU Allocation with V-Cache. (b) CPU Allocation with ECC. (c) CPU Allocation with VCONF.

(d) Memory Allocation with V-Cache. (e) Memory Allocation with ECC. (f) Memory Allocation with VCONF.

Figure 8. Traces of CPU and memory allocations using V-Cache, ECC, and VCONF.

that there is a significant increase in the memory allocation to the application VM and the database VM in ECC. This is due to the increased number of requests for dynamic content, which are merely speed up by caching. Due to the short expiration time of dynamic contents, increasing cache size does not necessarily lead to performance improvement.

Figure 8(c) shows that between the 30_{th} and 70_{th} intervals, VCONF almost allocates all CPU resource to the web, application and database VMs. Each VM is reaching its upper bound of CPU allocation. Figure 8(f) shows that between the 30_{th} and 70_{th} intervals, VCONF allocates every byte of memory to the web, application and database VMs. Referring to the results in Figure 7, we observe that the performance degradation of VCONF is due to the resource limitation. Even consuming all available resources is not enough for the significantly increased workload. In contrast, V-Cache and ECC are able to adaptively resize the cache VM and the application's VMs for performance improvement.

Due to the request clustering and cost-aware request redirection, V-Cache makes the most efficient use of the available resources and yields the best resource utilization efficiency. ECC tends to increase the cache VM capacity to store more content when the workload increases. As the cache size increases, more dynamic content will be cached. However, due to the short expiration time, the requests for these dynamic content can merely be speed up through caching. V-Cache considers the processing cost in and out of the cache VM for each request and only caches those requests that have greater potentials to be speed up by caching. This cost-aware request redirection makes the most effective capacity trade-off between the cache VM and the application VMs.

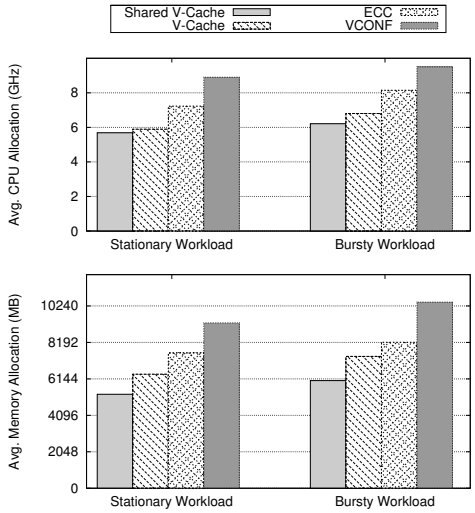


Figure 9. Resource allocation of approaches with two RUBiS applications.

B. V-Cache shared by Multiple Applications

We consider two scenarios, both with the same total amount of CPU and memory resources. In one scenario, one multi-tier application has a dedicated V-Cache system. In the other, two multi-tier applications share one V-Cache system. For each application, we emulate a light workload of 2000 users. V-Cache, ECC and VCONF all obtain very close effective throughput. But they consume different amount of resources. We also compare the resource utilization efficiency by the shared V-Cache and by the dedicated V-Cache.

Figure 9 shows the resource allocation of four different approaches. The resource allocation of the shared V-Cache

is the lowest among all approaches. It consumes 22.2% and 36.1% less CPU resource than ECC and VCONF, respectively. It also consumes 30.5% and 43.2% less memory resource than ECC and VCONF, respectively.

Compared with the dedicated V-Cache, the shared V-Cache consumes 16.3% and 17.6% less CPU and memory resources, respectively. This is because the overhead of using V-Cache can be amortized if it is shared by multiple applications. Sharing one V-Cache with multiple RUBiS applications also allows data sharing between correlated applications. This further reduces the memory consumption of V-Cache. This experiment demonstrates that such a V-Cache system is very beneficial to the IaaS cloud.

C. Impact of Cost-aware Request Redirection

V-Cache uses the cache VM only for requests that have the potential of being speed up by caching. This is due to the cost-aware request redirection. We compare it with the least frequently used (LFU) cache replacement algorithm and the least frequently and costly used (LFCU_K) cache replacement algorithm [5]. LFU favors high-frequency accesses. It evicts less frequently used contents. Caching high-frequency contents can increase the cache hit rate and improve the cache performance. LFCU_K considers both the access frequency and the miss penalty (*i.e.* the processing cost out of the cache). It evicts least frequently used and least costly contents from the cache.

We use three different RUBiS workload mixes: browsing, bidding, and selling. The differences in the dynamic content ratio, content size distribution, and business logic can significantly affect the cache performance. The workload is set to emulate 4000 concurrent users that will not overload the system. We evaluate the performance of different approaches using a fixed-size cache of 512 MB.

Figure 10 shows that V-Cache outperforms LFU and LFCU_K under all three workload mixes. Under the browsing workload mix, V-Cache outperforms LRU and LFCU_K by 2.2% and 3%, respectively. The performance improvement is not very significant due to the low dynamic content ratio of the browsing workload, in which most of the requests are retrieving static content. This is an ideal scenario for a cache system because the static contents have longer expiration time and higher potential of being speed up by caching. Thus, all three approaches are able to achieve decent performance.

As the dynamic content ratio increases, the performance differences among three approaches are becoming more significant. V-Cache outperforms LFU by 11.1% and 9.6% under the bidding and selling workload mixes, respectively. V-Cache outperforms LFCU_K by 8.4% and 12.3% under the bidding and selling workload, respectively. Although LFCU_K considers both the access frequency and the miss penalty, the cache performance is still largely affected by the dynamic contents. The access frequency and the miss penalty do not necessarily represent the potential speedup by caching for a given request. The results demonstrate that with fixed-size cache, using V-Cache's cost-aware request redirection can make the most efficient use of the cache space.

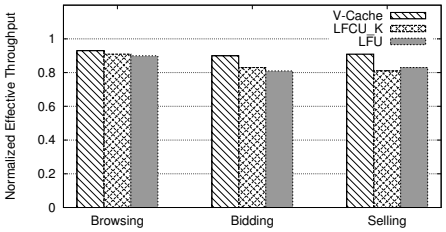


Figure 10. Performance of different approaches with fixed cache size.

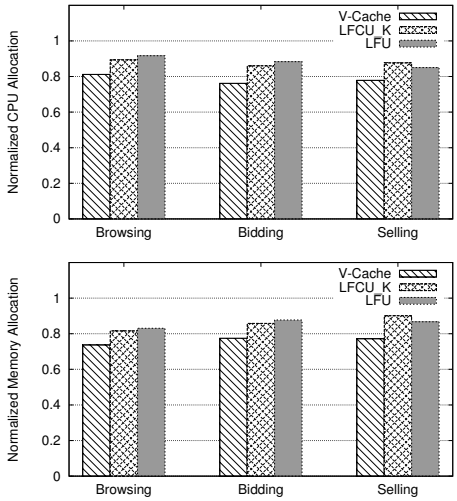


Figure 11. Resource usage of different approaches with fixed cache size.

Figure 11 shows the CPU and memory resource allocations due to V-Cache, LFCU_K and LFU. With the cost-aware request redirection, V-Cache uses less CPU resource and less memory resource than LFCU_K or LFU does. Caching dynamic content is not very helpful in reducing the workload to the application's VMs due to the short expiration time of the cached dynamic content. Thus, In LFU and LFCU_K, more resources of CPU and memory are needed by the application's VMs, causing higher overall resource consumption.

D. V-Cache with WikiBench Application

We evaluate V-Cache's performance with WikiBench [2], [33]. The workload has low write ratio. Most of the requests are read-only that are cachable. However, the workload still has a high dynamic content ratio, which in turn provides an opportunity for taking the advantage of V-Cache's cost-aware request redirection. We generate a dynamic workload. It starts with 2000 users, changes to 3000 users at the 20_{th} interval, and changes back to 2000 users at the 40_{th} interval.

Figure 12 shows the effective throughput due to V-Cache and VCONF under the dynamic workload. The results show that during the first 20 intervals and the last 20 intervals when the workload is at 2000 users, two approaches achieve similar performance in effective throughput. But when the workload hikes between the 20_{th} and 40_{th} intervals, V-Cache achieves

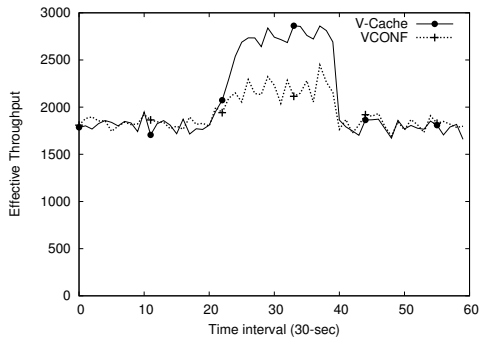


Figure 12. Throughput of WikiBench using V-Cache and VCONF.

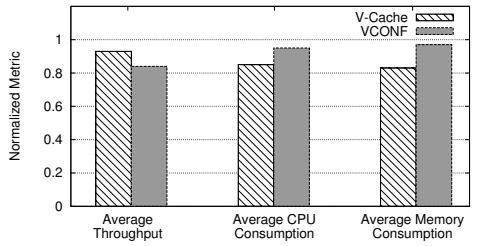


Figure 13. Normalized metrics of WikiBench using V-Cache and VCONF.

19.2% higher effective throughput than VCONF does. This is due to the fact that V-Cache is able to adaptively resize the cache VM and the application's VMs to meet the workload dynamics. On the other hand, VCONF only adaptively resize the application's VMs.

Figure 13 shows the normalized throughput, CPU consumption and memory consumption due to V-Cache and VCONF under the dynamic workload. The results are the average values during the 60-interval experimental period. We observe that V-Cache significantly improves both the performance and the resource utilization efficiency.

VI. Related Work

Autonomic resource allocation of multi-tier applications in virtualized environments is an important research topic [15], [19], [28], [35]. There were studies on the performance modeling and analysis of multi-tier servers with queueing foundations [22], dynamic server provisioning on multi-tier server clusters for end-to-end delay guarantee [34], and percentile-based delay guarantee in multi-tier service architecture [20].

There are recent studies that focus on improving user-perceived performance by automated VM resizing [16], [30], [31]. VCONF is a reinforcement learning based approach for VM auto-configuration [31]. It identifies the performance interference between different VMs and the sequence dependent of VM performance as the major challenges. It improves performance of TPC-W, TPC-C, and SPECjbb benchmark applications. Han *et al.* proposed a lightweight resource scaling approach for multi-tier applications hosted in cloud environments [16]. It is able to improve the resource utilization of the underlying hardwares. However, the improvement on performance and resource utilization efficiency is limited because of the upper bounded VM resources. It can hardly improve the performance when the system is meeting a bursty workload or highly dynamic workload.

Web cache has been widely used in improving the performance of web applications, especially when the workload increase dramatically. The web cache stores page content and speeds up web request processing. Previous studies focused on cache architectures, cache placement and replacement algorithms, cache prefetching mechanisms for different applications [4], [5], [8], [11], [12]. For many e-Commerce sites, web pages are created dynamically based on the business processes and databases. These dynamic pages usually are marked as non-cachable. *CachePortal* introduces intelligent dynamic page invalidation approach to enabling web caching for dynamic pages [8].

There are two recent studies on web cache for search engines, which incorporate the notion of the processing costs into the caching policies. Gan *et al.* proposed interesting cache policies that take the processing costs of search queries into account [12]. The cost function essentially represents the disk access cost of queries. It is estimated by computing the sum of the lengths of the posting lists for the query terms. Ozcan *et al.* proposed different approaches that calculate the cost using CPU processing time obtained when a query has been executed [5]. They used the cost and frequencies of queries to decide the cache eviction. However, the studies focused on the fixed-size cache and back-end servers. In the IaaS clouds, the size of the cache and back-end servers can be elastic and thus the costs of queries are variable. Our work proposes a cost-aware request redirection technique that is integrated with automated resource provisioning for IaaS clouds.

One recent study designed an elastic cloud cache (ECC) for scientific applications in cloud environments [10]. It proposed a distributed cache system for improving the performance of applications under bursty workloads. It used an automated scaling algorithm to adapt the number of cache nodes according to the workload variations. The approach focuses on improving the response time of the applications. It does not address the resource utilization efficiency issue. It also does not incorporate the cache scaling with the application resizing. Applying a cache to an application will significantly change the magnitude and characteristics of workload. Lacking of the incorporation can result in resource utilization inefficiency. In this paper, we integrate the cache resizing and application resizing in an automated manner. We design new approaches for request clustering and cost-aware selective caching that improve performance and reduce resource usage.

VII. Conclusions

The resource elasticity offered by IaaS clouds opens up opportunities for elastic application performance, but also poses challenges to application management. In this paper, we propose and design V-Cache, a machine learning based

approach to flexible provisioning of resources for multi-tier applications in clouds. V-Cache transparently places a caching proxy in front of the application. To achieve the optimal performance and resource efficiency, V-Cache uses a genetic algorithm to identify the incoming requests that benefit most from caching and dynamically resizes the cache size to accommodate these requests. We also develop a reinforcement learning algorithm to optimally allocate the remaining capacity to other tiers. We have implemented V-Cache on a VMware-based cloud testbed. Experiment results on the RUBiS and WikiBench benchmarks show that V-Cache outperforms a representative capacity management scheme and a cloud-cache based resource provisioning approach by at least 15% in effective system throughput, and achieves at least 11% and 21% savings on CPU and memory resources, respectively. V-Cache demonstrates that by flexibly provisioning the resources owned by users, providers reduce the resource requirement per application and increase the consolidation ratio while still meeting users' goals.

Our future work will be on extending V-Cache for heterogeneous applications and integrating admission control for overload control and performance guarantee.

Acknowledgement

This research was supported in part by U.S. NSF CAREER Award CNS-0844983 and research grant CNS-1217979.

REFERENCES

[1] Varnish Cache. https://www.varnish-cache.org/.

[2] WikiBench. http://www.wikibench.eu/.

[3] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The resource-as-a-service (RaaS) cloud. In *Proc. of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)*, 2012.

[4] S. Albers. New results on web caching with request reordering. In *Proc. of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2004.

[5] I. S. Altingovde, R. Ozcan, and O. Ulusoy. A cost-aware strategy for query result caching in web search engines. In *Proc. of European Conference on IR Research on Advances in Information Retrieval (ECIR)*, 2009.

[6] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *Proc. IEEE Int'l Workshop on Workload Characterization (WWC)*, pages 3 – 13, 2002.

[7] X. Bu, J. Rao, and C.-Z. Xu. A reinforcement learning approach to online web system auto-configuration. In *Proc. IEEE Int'l Conference on Distributed Computing Systems (ICDCS)*, 2009.

[8] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proc. ACM SIGMOD*, 2001.

[9] J. R. Challenger, P. Dantzig, A. Iyengar, M. S. Squillante, and L. Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Trans. on Netwworking (TON)*, 12(2):233–246, 2004.

[10] D. Chiu, A. Shetty, and G. Agrawal. Elastic cloud caches for accelerating service-oriented computations. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[11] T. Feder, R. Motwani, R. Panigrahy, and A. Zhu. Web caching with request reordering. In *Proc. of 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[12] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proc. of Int'l Conference on WWW*, 2009.

[13] A. Gordon, M. Hines, D. D. Silva, M. Ben-Yehuda, M. Silva, and G. Lizarraga. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *Proc. ACM ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE)*, 2011.

[14] Y. Guo, P. Lama, and X. Zhou. Automated and agile server parameter tuning with learning and control. In *Proc. of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, pages 656–667, 2012.

[15] Y. Guo and X. Zhou. Coordinated vm resizing and server tuning: Throughput, power efficiency and scalability. In *Proc. IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 289–297, 2012.

[16] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *Proc. IEEE/ACM Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, 2012.

[17] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proc. of ACM Symposium on Cloud Computing (SOCC)*, 2011.

[18] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the Tenant-Provider Gap in Cloud Services. In *Proc. of ACM Symposium on Cloud Computing (SOCC)*, 2012.

[19] M. Korupolu, A. Singh, and B. Bamba. Coupled placement in modern data centers. In *Proc. of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[20] P. Lama and X. Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *Proc. IEEE/ACM Int'l Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 151–160, 2010.

[21] P. Lama and X. Zhou. AROMA: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. ACM Int'l Conference on Autonomic Computing (ICAC)*, pages 63–72, 2012.

[22] P. Lama and X. Zhou. Efficient server provisioning with control for end-to-end delay guarantee on multi-tier clusters. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):78–86, 2012.

[23] P. Lama and X. Zhou. NINEPIN: Non-invasive and energy efficient performance isolation in virtualized servers. In *Proc. IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN)*, 2012.

[24] W.-S. Li, O. Po, W.-P. Hsiung, K. S. Candan, and D. Agrawal. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proc. of Int'l Conference on WWW*, 2003.

[25] C. Lin and C. S. G. Lee. Real-time supervised structure/parameter learning for fuzzy neural network. In *Proc. IEEE Int'l Conference on Fuzzy Systems*, pages 1283–1291, 1992.

[26] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In *Proc. IEEE Int' Conference on Autonomic Computing (ICAC)*, 2009.

[27] B. L. Miller, B. L. Miller, D. E. Goldberg, and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[28] J. Moses, R. Iyer, R. Illikkal, S. Srinivasan, and K. Aisopos. Shared resource monitoring and throughput optimization in cloud-computing datacenters. In *Proc. of IEEE Int'l Symposium on Parallel and Distributed Processing (IPDPS)*, 2011.

[29] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proc. of the 5th ACM European conference on Computer systems (EuroSys)*, 2010.

[30] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proc. of the EuroSys Conference (EuroSys)*, pages 13–26, 2009.

[31] J. Rao, X. Bu, C. Xu, L. Wang, and G. Yin. Vconf: A reinforcement learning approach to virtual machines auto-conguration. In *Proc. IEEE Int'l Conference on Autonomic Computing Systems (ICAC)*, 2009.

[32] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. Optimizing virtual machine scheduling in numa multicore systems. In *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[33] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[34] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Trans. on Autonomous and Adaptive Systems*, 3(1):1–39, 2008.

[35] Q. Wang, S. Malkowski, D. Jayasinghe, P. Xiong, C. Pu, Y. Kanemasa, M. Kawaba, and L. Harada. The impact of software resource allocation on n-tier application scalability. In *Proc. of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[36] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang. Probabilistic performance modeling of virtualized resource allocation. In *Proc. IEEE Int'l Conference on Autonomic computing (ICAC)*, 2010.