Developing an optimized application hosting framework in Clouds

Xuanhua Shi\textsuperscript{a,\ast}, Hongbo Jiang\textsuperscript{b}, Ligang He\textsuperscript{c}, Hai Jin\textsuperscript{a,\ast}, Chonggang Wang\textsuperscript{d}, Bo Yu\textsuperscript{a}, Xueguang Chen\textsuperscript{e}

\textsuperscript{a} Service Computing Technology and Systems Lab/Cluster and Grid Computing Lab, School of Computer, Huazhong University of Science and Technology, China
\textsuperscript{b} Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China
\textsuperscript{c} Department of Computer Science, University of Warwick, United Kingdom
\textsuperscript{d} InterDigital Communications Corp., USA
\textsuperscript{e} Department of Automation Control and Engineering, Huazhong University of Science and Technology, China

\section*{1. Introduction}

Clouds (or data centers) become increasingly popular nowadays in hosting Web applications \cite{11}. A Cloud may host a large number of applications. Many empirical studies show that the Web request rate is bursty and could fluctuate dramatically in a short period of time \cite{20}. Accordingly, it is not cost-effective to over-provision the resources in Clouds to satisfy the potential peak demands. Therefore, it is a crucial performance issue to judiciously place the applications across the physical machines in a Cloud.
In order to effectively utilize system resources, modern Web applications typically run on top of a middleware and rely on the middleware to dynamically allocate resources and meet the performance goals. This is the so-called dynamic hosting technology [1,29]. Previous studies [27,9,17] in this area focus on developing the algorithms to place the application instances on a given set of machines. Most existing algorithms were designed for a particular type of applications and assume that the application instances are independent with each other. In Cloud environments, however, there may exist a large number of different types of applications in the same machine, and these different applications may compete for underlying hardware resources.

Another issue that is often encountered in Cloud environments is that the hosted applications are often data-intensive, and that the requested data may scale up and down dynamically [3,30]. Under this circumstance, the CPU cycles may be wasted in busy-waiting for the completion of I/O operations. It is very difficult to accurately model the problem of busy-waiting combined with resource competition among applications. This is because CPU consumptions depend heavily on resource competition, which is in turn affected by the number of concurrent applications and their execution patterns at runtime. In the existing studies, it is assumed that the CPU consumption increases linearly as the level of the workload submitted to the system increases. The benchmarking experiments conducted in this paper show that this may not be always true when the system is serving concurrent application requests.

To address the aforementioned issues, we design and implement an Enhanced Application PlACement Framework (EAPAC) in this paper. There are two main components in EAPAC: an application-level load balancer and an application server manager. The former component dispatches application requests to specific application servers, while the latter manages the resource allocation in individual application servers.

The rest of this paper is organized as follows. Section 2 illustrates the motivation of this work through the benchmarking experiments. The design and implementation of EAPAC are presented in Section 3. Section 4 evaluates the performance of EAPAC. The related work is presented in Section 5. Finally, Section 6 concludes the paper.

2. Motivation

Existing application placement methods in literature [27] usually treat all types of request equally and estimate a server’s resource consumption by simply adding up the resource consumption caused by each individual request when it is running in the server alone. However, our studies show that the resource consumptions estimated by this approach may not be very accurate when multiple requests are running in the system concurrently. The inaccuracy may reduce the efficiency of the request/application scheduling strategies proposed in literature.

We conducted the benchmarking experiments to demonstrate this situation. The experiment platform consists of two IBM HS21 nodes taken from High Performance Computing Center at the Huazhong University of Science and Technology. Each HS21 node is equipped with a 2-way, 4-core Intel Xeon CPU E5345 running at 2.33GHz, and with 8GB memory and a 73GB SCSI hard disk. The two nodes are interconnected via a Gigabit Ethernet network and a 10Gbps Infiniband. Both nodes run the RedHat Enterprise Linux 5 (Linux 2.6.18-8.el5). An emulated data-intensive application on JBoss is implemented, which creates 100MB data when receiving a request. One node is used to generate and send the requests using Apache Benchmark (AB), while the other node serves the requests, each of which invokes an instance of the emulated application.

In total, 1000 http requests are generated, and they are sent in the following four patterns, which represent different concurrency levels: (1) one request every time; (2) two concurrent requests every time; (3) four concurrent requests every time; (4) eight concurrent requests every time.

Fig. 1 shows the results of the average processing time of one request in the aforementioned four scenarios. The x-axis represents the concurrency level, and y-axis is the average processing time of a request. Different legends in the figure correspond to the time spent in different types of operations in the benchmark application. As can be seen from this figure, the “usr”, “iowait” and “sys” time dominate the processing time. The “iowait” time refers to the time when the CPU waits for the completion of I/O operations. The “usr” and “sys” time are the time when the application is running in the user and kernel space of the operation system, respectively. Note that the processing time in the figure is the time when the CPU is physically occupied for processing the request (not the duration between the time when the request starts execution and the time when it finishes). It can be observed from this figure that the physical processing time for a request increases as the concurrency level increases. For example, the processing time for the concurrency level of 8 is about 4 seconds, while the physical processing time for the concurrency level of 1 is about 1.2 seconds. This means that when there are 8 requests running in the server concurrently, their total CPU consumption is not 1.2 × 8 = 9.6 seconds, but 4 × 8 = 36 seconds.

Fig. 2 shows the processing time of each request as the above experiments progressed. It can be verified from this figure that when the concurrency level increases, the processing time of the requests increases. Further observations in this figure show that when the concurrency level is 1, 2 and 4, the processing times of the requests are mostly distributed around certain values (which are about the average processing time shown in Fig. 1). However, when the concurrency level is 8, the processing times become volatile, which suggests that the system is now in an unstable state. The results observed in this figure verify those obtained in Fig. 1. Moreover, they indicate that the system may become unstable when the concurrency level is too big (this phenomenon will be discussed in more detail later in this paper).

The results obtained in the above benchmarking experiments deviate from the assumption widely made in the existing studies of application scheduling. The existing studies assume that the system employs a perfect time-sharing fashion to run multiple application instances concurrently and therefore, the processing time of an instance increases linearly with the number of instances running in the system. From another perspective, the benchmarking results obtained in this paper suggest that the CPU consumption is not necessarily proportional to the level of workload submitted to the system. The underlying reason for the non-linearity may be due to the workload management strategies employed in the operating system. There are two possible solutions for this problem. One is to revise the operation system kernel. The other lies in the application level by judiciously allocating to application servers both an appropriate amount and combination of different types of request. This paper focuses on the latter approach and develops a new application placement framework, EAPAC.

3. The EAPAC framework

This section presents the design and implementation of EAPAC. Fig. 3 depicts the architecture of EAPAC, which consists of two main components: the Load Balancer (LB) and the Application Placement Manager (APM) (i.e., the “Manager” in the figure). As shown in the figure, the clients send the requests to the LB which then selects an application server to
handle each request. While the LB schedules the incoming requests to application servers, the APM is responsible for mapping the application servers to the physical servers with the aim to achieve the optimized performance. The APM make the application placement decisions based on the physical resource capacity, such as CPU power and I/O bandwidth, as well as the information of the requests, such as the number of the requests, resource demands of the requests, and so on.

3.1. The algorithm for the load balancer

In a Web hosting system, a physical server can host a diverse collection of application and the same application can be deployed on multiple physical machine. Take Fig. 3 as an example. Physical Server 1 hosts app 1 and app 2, while app 2 is deployed on both Server 1 and Server 2. The LB is situated at the application level, which recognizes the type of the requests and then dispatches the requests to the corresponding applications for service. The load-balancing strategy is performed among the same type of applications. In this paper, the LB adopts an existing load balancing algorithm, called Weighted Least-Connections (WLC) [26]. In WLC, performance weights are assigned to each application on different physical servers and the percentage of live connections is proportional to its weight value. For example, in Fig. 3, if app 2 on Server 1 is assigned with the weight of 0.3 and app 2 on Server 2 with 0.7, then 70% of the requests for app 2 will be sent to Server 2. The weights are assigned by the APM in EAPAC.

3.2. The algorithm for the application placement manager

A new algorithm is developed for the APM in this paper. The algorithm is employed by the APM to make application placement decisions. Two factors are considered in order to make the decisions.

One is the concurrency level within an application. As illustrated in Section 2, concurrent requests may lead to the distortion of resource consumptions. Therefore, in the first step, the characteristic of the requests are examined. The output of this step is the maximum number of the requests that can be accepted by an application on a physical server. We use the following way to determine the concurrency level of an application. If the application takes $m$ seconds to complete a request, the concurrency level of the application is regarded as $\frac{1}{m}$, which suggests the application is able to cope with no more than $\frac{1}{m}$ requests in one second.

The other factor is the concurrency level among different applications sitting in the same physical server. The concurrent requests from different applications can also cause the distortion of resource consumptions. Consider a variation of the scenario presented in Section 2. In the new scenario, two applications, app i and app j, are deployed in the physical server. A request to one of the applications will create a file of size 100MB. The experiments show that when the requests are served concurrently by these two applications, similar distortion of resource consumptions occurs. We further examine the relationship between the requests for two applications, and an exclusive tuple is designed to represent the concurrency relation between two applications. The tuple for app i and app j takes the form of $(i, j, \frac{1}{m+n})$, if app i and app j take m and n seconds to complete a request, respectively. The exclusive tuples will be used by EAPAC to avoid allocating too many concurrent requests to different applications on the same physical server.

In order to make application placement decisions, EAPAC needs to predict the arrival rate of the incoming requests. In EAPAC, the following method is used for prediction. EAPAC records the number of requests in the current time windows, and the recorded number is used to predict the number of requests in the next time windows.
(i) maximize \( \sum_{m \in M} \sum_{n \in N} L_{mn} \)

(ii) minimize \( \sum_{m \in M} \sum_{n \in N} |I_{mn} - I_{mn}^*| \)

(iii) minimize \( \sum_{n \in N} \left| \frac{\sum_{m \in M} L_{mn}}{\Omega_n} - \rho \right| \)

with

(iv) \( \forall m \in M, \forall n \in N, \quad I_{mn} = 0 \text{ or } 1 \)

(v) \( \forall m \in M, \forall n \in N, \quad 0 < MA_{mn} \leq 1 \)

(vi) \( \forall m \in M, \forall n \in N, \quad MA_{mn} = 0 \Rightarrow I_{mn} = 0 \)
\( \forall m \in M, \forall n \in N, \quad MA_{mn} > 0 \Rightarrow I_{mn} = 1 \)

(vii) \( \forall m \in M, \forall n \in N, \quad I_{mn} = 0 \Rightarrow L_{mn} = 0 \)

(viii) \( \forall m \in M, \forall n \in N, \quad O_{mn} = CI_n L_{mn} \)

(ix) \( \forall n \in N, \sum_{m \in M} \Upsilon_m MA_{mn} \leq \Gamma_n \)

(x) \( \forall n \in N, \sum_{m \in M} L_{mn} \leq \Omega_n \)

(xi) \( \forall n \in N, \sum_{m \in M} O_{mn} \leq O_n \)

(xii) \( \forall m \in M, \sum_{n \in N} L_{mn} \leq \omega_m \)

(xiii) \( \forall m \in M, \sum_{n \in N} O_{mn} \leq P_m \)

The application placement problem can be modeled as Eq. (1). The objectives of the application placement manager developed in this paper are as follows:

1. Maximizing CPU utilization: this is expressed in Eq. (1)(i), where \( L_{mn} \) denotes the amount of CPU cycles allocated to application \( m \) on node \( n \).
2. Minimizing application switching: this is expressed in Eq. (1)(ii), where \( I_{mn} \) denotes the new application deployment matrix, while \( I_{mn}^* \) denotes the old deployment matrix.
3. Balancing workload among the physical nodes: this is expressed in Eq. (1)(iii), where \( \rho \) denotes the average load of all physical nodes.

The application placement model considers the following constraints:

1. An element of the deployment matrix, \( I \), is either 0 or 1, shown in Eq. (1)(iv).
2. An element of the application placement matrix, \( MA \), is between 0 and 1, as shown in Eq. (1)(v).
3. Eq. (1)(vi) presents the relationship between the matrix \( I \) and the matrix \( MA \).
4. If an element of the deployment matrix is 0, then the resource utilization is 0, shown in Eq. (1)(vii).
5. Without considering the concurrency impact, the I/O utilization and the CPU utilization has the linear relationship for an application. This is shown in Eq. (1)(viii).
6. The total memory demand of all applications on node \( n \) is no more than the total memory of node \( n \), shown in Eq. (1)(ix). \( \Upsilon_m \) denotes the memory demand of application \( m \), and \( \Gamma_n \) denotes the memory capacity of machine \( n \).
7. The total CPU demand of all applications on node \( n \) is no more than the CPU capacity of node \( n \), as shown in Eq. (1)(x). \( \Omega_n \) denotes the CPU capacity of machine \( n \).
8. The total I/O demand of all applications on node \( n \) is no more than the I/O capacity of node \( n \), as shown in Eq. (1)(xi). \( O_n \) denotes the I/O capacity of node \( n \), and \( O_{mn} \) denotes the I/O demand of application \( m \) on node \( n \).
9. The CPU capacity allocated to application \( m \) on all nodes is no more than the total CPU demand of application \( m \), as shown in Eq. (1)(xii).
10. The total I/O capacity allocated to application \( m \) on all nodes is no more than the total I/O demand of application \( m \), as shown in Eq. (1)(xiii).
Fig. 4. Pseudocode for application placement algorithm.

The application placement model is a variant of the Class Constrained Multiple-Knapsack problem [19]. We solve the problem using the Greedy Algorithm [4]. Its pseudo code is shown in Fig. 4 and Fig. 5, which are explained as follows.

The algorithm aims to compute a better application placement matrix, $MA$. The matrix has two dimensions. A row corresponds to a physical node, and the column is the application ID. An element of the matrix, $ma_{ij}$, represents out of all requests submitted to application $i$, the percentage of those that are dispatched to node $j$. If $ma_{ij}$ is 0, then application $i$ is not deployed on node $j$.

In order to calculate $MA$, we first study the hardware resource capacity, such as CPU, memory and IO, as shown in Lines 3, 4 in Fig. 4. The expected request rate and the resource consumption also need to be known before computing the placement matrix, as shown from Line 5 to Line 8. After obtaining the total resource consumption, we can compute the average load of all physical nodes. Suppose we have 4000 requests in next 10 seconds, and each request consumes 10,000 CPU cycles. Also assume the total CPU cycles that can be provided by all servers in next 10 seconds are 50,000,000. Then the average load of all the servers is $\frac{4000 \times 10,000}{50,000,000} = 80\%$. With these parameters, the 'calc_better_place_matrix' function, which is shown in Fig. 5, will loop for 'MaxLoop' times to calculate a better matrix according to the objectives in Eq. (1).

In the 'calc_better_place_matrix' function (as shown in Fig. 5), we compare the load between two nodes one by one (based on the current placement matrix), as shown in Lines 10, 11. If the load on node $J$ is higher than the load on node $J'$, we will try to reduce the load difference generated by application $I$ and $I'$ with the following steps: (1) Computing the ratio between the CPU consumption and the IO consumption for application $I$ and $I'$ (Lines 12–16); (2) Comparing the load on node $J$ generated by application $I$ and the load on node $J'$ by application $I'$. If the load on $J$ is higher than the load on $J'$, subject to the constraint that the CPU-to-IO ratio is smaller, then the algorithm reduces the load on $J$ with the updated load difference from application $I$ and application $I'$ (the decreased load by application $I$ is larger than the increased load by application $I'$), as shown from Line 19 to Line 28; (3) After changing the workload of application $I$ and $I'$ on node $J$ and $J'$, the algorithm updates the placement matrix $ma_{ij}$, $ma_{ij}'$, $ma_{i}j$ and $ma_{i}j'$ accordingly.

If the IO consumption for application $I$ or application $I'$ is zero, the CPU-to-IO ratio cannot be calculated. As shown from Line 31 to Line 45, we move $K$ proportion of load generated by application $I$ from node $J$ to node $J'$. In order to guarantee the load that is moved to node $J'$ will not exceed the IO capacity of node $J'$, the value of $K$ is tuned in Lines 40 and 41. After adjusting the load in this way, the load among nodes will be more balanced.

After executing the 'calc_better_place_matrix' function, the control flow returns to the place() function in Fig. 4. The place() function will compare the new matrix and the existing matrix, shown in Lines 17 and 18. If the system can fulfill the objectives in Eq. (1), the new matrix is regarded as a better solution and the loop continues. If the algorithm cannot further improve the solution after executing 10 consecutive loop iterations, the algorithm will break the loop, as shown in Lines 20, 21.

3.3. Implementation

Fig. 6 illustrates the architecture of EAPAC. As mentioned in the previous section, there are two parts in EAPAC: Load Balancer (LB) and Application Placement Manager (APM). The LB forwards the requests to the suitable applications, and the APM handles the application placement on the back-end physical nodes. Besides, there is a testing node in EAPAC, which
estimates the resource consumptions of the submitted requests, including CPU, memory and I/O. In Fig. 6, the rectangular boxes in solid lines represent the key functional modules of EAPAC, while the boxes in dashed lines represent the state information maintained by EAPAC. The arrows in solid lines represent the communication flow, while the arrows in dashed lines represent the movement of the requests.

Each functional module in EAPAC is discussed as follows.

Matcher: The EAPAC maintains a list of the applications. For an incoming request, the Matcher determines which application is able to serve it.

Counter: The Counter counts the requests submitted to each application. The number of the requests will be sent to the APM for making application placement decisions.

Req Dispatcher: The Req Dispatcher works as a router for requests, it forwards a request to a specific application on a specific node, and establishes the connection between the application and the clients. The routing decisions are made based on the matrix produced by the Req Analyzer, which is described below. The load balancing algorithm discussed in Section 3.1 is performed by this module.

Req Analyzer: The Req Analyzer has two functions: (1) analyzing the requests, and (2) making application placement decisions. Based on the Config DB, which is described next, the Req Analyzer analyzes the requests and estimates their resource consumptions. Also, the Req Analyzer decides how to dispatch the requests. As demonstrated in Section 2, the requests’ CPU consumptions may increase disproportionately when the concurrency level exceeds a certain threshold. Therefore, the dis-

```
1: function calc_better_place_matrix(MA[], R_A[], R[])
2: {
3:   // adapting application placement matrix
4:   // Compare load on each nodes
5:   // M is the number of applications, N is the number of physical nodes
6:   for(l = 1; l <= M; l++) {
7:     for(j = 1; J <= N; J++) {
8:       for(i = 1; I <= M; I++) {
9:         // compare the total load on node J and node J'
10:        if(Sum J > Sum J') {
11:           */ get the ratio between the CPU consumption and
12:           IO consumption of application I and application I' */
13:          Cl = CPU_I / IO_I; // app I
14:          Cl' = CPU_I' / IO_I'; // app I'
15:          */ compare the load of application I on node J
16:          and the load of application I' on node J' */
17:          if (Load_I > Load I') & (Cl < Cl') {
18:            K = Min(Load I', Load_U - Load I', (SumJ-SumJ') / 2);
19:            Kio = K/Cl;
20:            K' = Kio*Cl;
21:            */ move workload of app I on node J with K quarter to node J'
22:            move workload of app I' with K' quarter to node J' */
23:            Load_I -= K;
24:            Load_U -= K;
25:            Load_I' -= K;
26:            Load_I' -= K;
27:            Load_I -= K;
28:            Load_I -= K;
29:            //Change MAU, MAU, MAJ, and MAJ accordingly;
30:           }
31:       } else (Load_I > Load I') {
32:         K = Min(Load I', Load_U - Load I', (SumJ-SumJ') / 2);
33:         */ move some workload of app I on node J to node J' */
34:         SWAP_I = 0;
35:         if(IO <> 0) {
36:           Cl = CPU_I / IO_I; SWAP_I = K/Cl;
37:           // Sum J' refers to IO load on node J'
38:           // M_IO_J refers to IO capacity of node J'
39:           if(SWAP_I = 0) {
40:             SWAP_I = (M_IO_J - Sum_J) / IO;
41:             SWAP_I = SWAP_I;
42:           }
43:           Load_I -= K;
44:           Load_I' -= K;
45:           //Change MAU, MAU, MAJ, and MAJ accordingly;
46:           }
47:         }
48:         return MA[];
49:       }
50:   }
51: }
```

Fig. 5. Pseudocode for computing application placement matrix.
the patching principle is to maintain the concurrency level in an application below the desired threshold. The Req Analyzer will generate the application placement matrix, $MA$, as shown in Fig. 4.

**Config DB**: The **Config DB** is a database which stores the following information: (1) the resource demands of the applications e.g., demand for CPU cycles and memory; (2) arrival pattern of requests, i.e., how many requests arrive in a specific interval; (3) resource status of the physical servers in the data center, e.g., free CPU cycles, free memory; (4) rules defined by the administrators, e.g., giving more resources to the app 1; (5) rules generated by the Req Analyzer, e.g., app 1 and app 2 should not be placed in the same physical server.

**App Placer**: The **App Placer** is the controller for applications. The **App Placer** starts, resumes, or turns off applications on physical nodes based on the matrix $MA$.

**Forecastor**: The **Forecastor** has two functions: (1) collecting the resource status and (2) forecasting the arrival pattern of the requests. With the data in **Config DB**, the **Forecastor** predicts the number of requests in the next interval, and sends the results to **Config DB**. As for collecting status, the **Forecastor** collects both the status of the physical nodes and the resource consumptions of applications.

The **Req Analyzer** is the key component in EAPAC, which instructs how to host the applications in the data center. Similar to the traditional way illustrated in [27], it also analyzes whether there exist resource conflicts among the requests. In EAPAC, we implement two versions of **Req Analyzer**: the first version performs the application placement by assuming that the applications’ CPU consumptions increase linearly as the number of requests increases. The second version implements the **Req Analyzer** in the following way: EAPAC first studies the resource consumption of the requests in a testing node, and tries to identify the resource conflicts, which will lead to the extra CPU overhead. The established resource conflict rules are stored in the **Config DB**. When a new request arrives, the **Req Analyzer** will check with the **Config DB** to avoid the resource conflicts caused by concurrent requests. The two versions are implemented using PHP. We will evaluate their performance in next section.

### 4. Performance evaluation

We conducted a set of experiments. All experiments were running on the dedicated IBM Blade cluster in HPCC as mentioned in Section 2. All HS21 nodes were connected with a Gigabit Ethernet network. We constructed two experimental settings. In the first settings, we used nine HS21 nodes in HPCC: One HS21 node was used to run EAPAC, and the remaining eight to deploy applications. The HS21 node is equipped with a 2-way, 4-core 2.33GHz Intel Xeon CPU E5345 and 8GB memory. To illustrate the performance of EAPAC more clearly, we only used one core of the Xeon CPU to run the LB and the APM of EAPAC.

We used three types of application in our experiments: (a) the transactional Web e-Commerce benchmark tpwc with 1000 items, which is used to represent a realistic application; (b) a synthetic application that imposes a tunable CPU load (called a CPU-bound application); (c) a synthetic application that imposes a tunable I/O load (called an IO-bound application). We did not choose any memory-intensive applications in our experiments, because we studied many real Web
applications and found that their memory consumptions did not change dynamically with different number of requests. We also found that CPU is always the bottleneck for the Web systems. To illustrate the resource conflicts of concurrent requests, we used data-intensive applications. We ran seven applications in total. Unless otherwise stated, the applications were run within the JBoss Application Servers. We tested the CPU and I/O consumptions for these applications and the results are shown in Table 1. The CPU consumption is measured with million CPU cycles, and the I/O consumption measured with the time it takes to complete the I/O operation. The application requests were generated by Httperf\[12\].

In the second experimental settings, we used different number of HS21 nodes (varies from 1 to 16 nodes) to host the applications. Similar to the first experimental settings, we ran the same set of applications. Only one core of the Xeon CPU is used to host EAPAC. The application requests are evenly generated for all applications by Httperf.

In all graphs presented in this section, each measurement is the average of 5 independent experiment runs. Unless otherwise stated, the experiments are conducted in the first experimental settings.

### 4.1. Throughput

The first metric we used to evaluate the performance of EAPAC is reply rate, which can reflect the system throughput. The reply rate is defined as the number of connections between the clients and the Web servers in one second. We compared the reply rate between EAPAC and the method presented in literature\[27\]. The method in\[27\] (called Tang’s method in this paper) also considers the application placement, but it does not address the challenges imposed by concurrent requests.

In order to understand the performance of EAPAC, the request rate generated by Httperf is the same for each of these seven applications. For example, 20 requests/second means that each application gets 20 requests a second.

Figs. 7(a) and 7(b) show the function of reply rate over the request rate in EAPAC and Tang’s method, respectively. It can be observed from these two figures that when the request rate is less than a threshold, the reply rates for both methods increase linearly as the request rate increases, and that two methods have the same increasing rate. When the request rate is higher than the threshold, the reply rates start to decrease or fluctuate. This suggests that this threshold is the maximum request rate that the application is able to handle. By comparing Figs. 7(a) and 7(b), it can be seen that the maximum request rate in EAPAC is higher than that in Tang’s method, and consequently the reply rate achieved by EAPAC is higher (by about 35%) than that by Tang’s method. This is because our method takes the concurrency level into account and the applications are placed in the way to reduce the concurrency level in the servers.

Another observation from Fig. 7 is that in Tang’s method, the increasing rate of application appf diminishes when the request rate is greater than 30, while in our method appf has the similar increasing rate as other applications until the request rate reaches 100. Since appf has the biggest resource demand among all applications, this result suggests that compared with Tang’s method, our method performs better when the applications have more intensive resource demand.

### 4.2. Error connections

In Fig. 7(b), the peak reply rate of appf is much larger than that of other applications, and the reply rate of appf changes sharply. To analyze this situation, we recorded the error connections in our experiments, the recorded data is shown in Fig. 8. From the application set-up, each experiment can be completed in 15 seconds, and therefore we ran each experiment for 15 seconds. In Fig. 8, the x-axis shows the request rate, and the y-axis shows the number of total errors in one experiment for 15 seconds. Fig. 8(a) shows the error rate of EAPAC, while Fig. 8(b) shows the error rate of Tang’s method.

The following observations can be made in Fig. 8:

1. There are no error connections when the request rate is less than a threshold reply rate: in Fig. 8(a), there are no error connections when the request rate is less than 100; in Fig. 8(b), there are no errors when the request rate is less than 40. When the request rate is more than 40, the number of the errors increases for appf. This shows that EAPAC has less error connections compared with Tang’s method.

2. When the request rate is greater than 110, there are error connections for the applications in EAPAC. This suggests that EAPAC does not have stable behaviors over this threshold. This is due to the fact that Httperf does not control the requests. Httperf only generates the http requests in the set-up stage before experiments.

3. The total number of errors in EAPAC is much less than that in Tang’s method, as shown in Fig. 8, when the request rate is 140, the total errors in Fig. 8(a) is about 300, while the total errors in Fig. 8(b) is about 2400.

4. In Fig. 8(b), the request rate is 140, the total errors in Fig. 8(a) is about 300, while the total errors in Fig. 8(b) is about 2400.
the number of errors for all applications changes dynamically, while in Fig. 8(a), the number of errors only increases slowly. This indicates that even with some errors, the EAPAC system is still practical, while the applications deployed using Tang’s method are out-of-service when the request rate is greater than the threshold reply rate. This result suggests that EAPAC has better availability than Tang’s method.

4.3. Response time

The performance of EAPAC is also evaluated in terms of response time. In our experiment, the processing cycle of a request is as follows. Httperf generates a request and send it to the EAPAC, which further forwards the request to an application server. The application server then processes the request and returns the response to Httperf. The time duration of a request’s processing cycle is defined as the request’s response time.

Figs. 9(a) and 9(b) show the requests’ average response time in EAPAC and Tang’s method, respectively, as the request rate increases. As can be seen from Fig. 9(a), the response times of all applications except tpcw stay almost constant when the request rate is less than 110, and then increase gradually. In Fig. 9(b), the response times start to increase dramatically when the reply rate reaches 80 (except for appf, which starts to increase when the reply rate is 40). This indicates that compared with Tang’s method, our method is able to deal with heavier workload.
It can also been observed from Fig. 9 that the response times of all applications change at a similar pace in our method, while the response times become volatile after a certain point in Tang’s method. This suggests that our application placement strategy can achieve better load balancing among servers than Tang’s method.

In order to get insight into the details of processing a request, we deployed the applications in 16 nodes and generated the requests with the rate of 150. And then the response times of all requests were recorded as the experiment progressed. The recorded response times are shown in Fig. 10. The x-axis shows the timeline of the experiment, and the y-axis shows the response time. A data point in Fig. 10 corresponds to the response time of a request.

The following observations can be made in Fig. 10:

1. At the beginning of the experiment, the response times achieved by EAPAC and by Tang’s method are quite close. In both cases, the response time is less than 500 ms. When the experiment timeline reaches 2000 ms, the average response time achieved by Tang’s method is about 500 ms, while the response time by EAPAC is about 100 ms. This suggests that the EAPAC users can get their request responses 4 times faster than the users of Tang’s method.

2. When the timeline gets to 14,000 ms, the shortest response time for Tang’s method is bigger than 1500 ms and the average response time is about 2000 ms, while for EAPAC, the average response time is still about 100 ms. This result once again shows that EAPAC performs better than Tang’s method.
Fig. 9. Response time.

(3) The response time achieved by EAPAC is stable as the experiment progresses, which is around 100 ms. While the response time by Tang’s method varies a lot. This suggests that compared with Tang’s method, EAPAC can deliver more stable Quality of Service.

4.4. Overhead

We also compared our methods with Tang’s method in terms of overhead. Overhead is measured as the time spent in reaching the application placement solution. In EAPAC, the Forecaster only takes the resource status of the back-end servers. In order to measure the overhead of EAPAC, we wrote a script to gather the CPU utilization of the node which hosts EAPAC. The script monitors the CPU status from /proc/stat in real time.

In the first experiment, we used one core of the HS21 node to run EAPAC. The CPU utilization of the core is plotted in Fig. 11.

It can be seen from Fig. 11 that EAPAC and Tang’s method have the similar CPU utilization. This shows that although EAPAC outperforms Tang’s method, it is not at the expense of a higher overhead.

Another interesting observation is that CPU utilization decreases when the request rate exceeds a threshold. This is because when the request saturates the system, there will be many error responses. The system does not process the error responses and therefore the CPU utilization decreases. By comparing Fig. 11 and Fig. 7, it can be seen that the threshold in
Fig. 11 is very close to the request rate at which the reply rate starts fluctuating. This result verifies the analysis about the reply rate.

We also investigated the overhead of EAPAC in the second experimental settings, because an important factor that affects the overhead is the number of nodes that EAPAC is managing.

In the second experimental settings, we recorded the CPU consumption of EAPAC under the maximum stable reply rate. Fig. 12 shows the CPU consumptions under different number of nodes.

The following observations can be made in Fig. 12:

1. When the number of nodes increases from 1 to 16, the CPU consumption increases from 10% to 50%. This shows that the overhead increases at a much slower rate than the increasing rate of the number of nodes.

2. EAPAC and Tang’s method have similar overhead. When the number of nodes increases, the overhead of EAPAC is slightly higher than that of Tang’s. However, considering EAPAC can achieve much higher reply rate as the number of nodes increases, EAPAC still has a much higher performance-to-cost ratio.

4.5. Scalability

We investigate the scalability of EAPAC using the second experimental settings.

As we can see from the discussions in Section 4.1, the maximum stable reply rate can reflect the throughput of the application servers. So Fig. 13 only plots the maximum stable reply rate under different number of servers. For the sake of clarity, we only plot the data for tpcw and appf. It can be seen that the throughput of Tang’s method has less prominent increase than that of EAPAC, as the number of nodes increases. This shows that EAPAC has better scalability in terms of reply rate.

Another observation is that in EAPAC, the reply rate of appf increases slightly faster than that of tpcw. The phenomenon is due to the impact of concurrency. When there are more nodes, EAPAC can deploy the application servers on more nodes, which reduces the concurrency level. This indicates that EAPAC can moderate the performance penalty caused by the concurrent requests.

5. Related work

There is a growing interest in the efficient management of data centers and hosting platforms. Some of them target the algorithms of hosting applications. For example, Kimbrel et al. [9] and Tang et al. [27] proposed the dynamic application placement algorithms in response to the changes in application demands. Their work is close to EAPAC. However, as illustrated in Section 1, EAPAC analyzes the requests and make application placement decisions by taking into account both concurrency of the requests and resource conflicts of the applications. The algorithm proposed by Urgaonkar et al. [29] allows the applications to share machines. However, the algorithm only considers a single bottleneck resource, does not dynamically change the number of instances of an application. Moreover, the algorithm does not aim to minimize the placement changes. The placement problems have also been modeled using various approaches, including bin packing, multiple knapsack, and multi-dimensional knapsack, etc. [19]. These modeling techniques are the theoretical foundation of the application placement strategy used in EAPAC.
Some research works target different resource-sharing environments [6,15,22,23]. Qin et al. [15] propose two I/O-aware load-balancing schemes for two types of clusters, which address the scheduling algorithm for IO-intensive applications. In EAPAC, the application placement considers the IO impacts on CPU consuming. Jiang et al. [6] proposed a service-on-demand framework for application service with virtual machines. Chase et al. proposed a dynamic virtual cluster which shares resources at the granularity of the entire machines [2]. In EAPAC, however, the application servers are running in a native, non-virtualized environment.

Placing application replicas is one of the key aspects in a hosting platform. It received much attention in the past. Previous work in this area mostly focused on the algorithms of determining the number and location of the replicas, either in the wide area network [7,17,18,8,10] or within a given data center. In EAPAC, the applications can be replicated, but the requests are scheduled by the load balancer. The replication algorithm is outside the scope of this paper.

Al-Qudah et al. addressed the problem of efficient enactment of application placement when the number and location of application instances have been determined [1]. Some key technologies was adopted in [1], such as prefetching [13]. In this paper, we do not address these problems. But these methods can be easily integrated into EAPAC. The agility of hosting platforms has been recently addressed by Qian et al. [16]. Unlike our work, the authors targeted the environments in which application instances are organized in the application-level clusters and hosted on virtual machines.
6. Conclusions and future work

Application placement is an importance issue in Clouds (which are also called data centers). In this paper, we design, implement, and evaluate an application placement framework, EAPAC, for Cloud environments. EAPAC analyzes the Web requests before making application placement decisions, aiming to avoid resource competition among different applications. We carried out extensive experiments for performance evaluation. Compared with the existing application placement method in literature, EAPAC improves throughput by 30% for all applications we tested. Moreover, the workload is better balanced by EAPAC, and consequently it achieves more stable and shorter response time.

Our future work includes the following directions: (1) We will integrate the virtual machine technology into EAPAC, aiming to enable EAPAC to support resource provision \cite{28,24}; (2) Currently, EAPAC handles the resource competition from different applications. We will address the resource co-allocation for applications \cite{5,25,14}; (3) In data centers, the fine-granularity monitoring can improve the back-end resource utilization. We will plan to strike a balance between the monitoring granularity and the monitoring overhead; (4) Currently, we only use a simple WLC load balancing algorithm. In the future, we will explore more advanced load balancing algorithms to study the relationship between the load balancing algorithm and the application placement algorithm.
Acknowledgments

This paper is partly supported by the NSFC under grant Nos. 60973037, 61133008, 61073024 and 61142010, Wuhan Chenguang Program under grant No. 201050231075, National Science & Technology Pillar Program of China under grant No. 2012BAH14F02, and the Research Project Grant of the Leverhulme Trust (Grant No. RPG-101) in the UK.

References