

EAPAC: An Enhanced Application Placement Framework for Data Centers

Xuanhua Shi¹ Hongbo Jiang² Ligang He³ Hai Jin¹ Chonggang Wang⁴ Bo Yu¹ Fei Wang¹

¹Service Computing Technology and System Lab / Cluster and Grid Computing Lab,
School of Computer, Huazhong University of Science and Technology, China

²Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China

³Department of Computer Science, University of Warwick, United Kingdom

⁴InterDigital Communications Corp., USA

¹{xhshi, hjin, yubo}@hust.edu.cn, whwangfei@gmail.com, ²hongbojiang2004@gmail.com, ³liganghe@dcs.warwick.ac.uk, ⁴cgwang@ieee.org

Abstract—Emerging data centers may host a large number of applications that consume CPU power, memory, and I/O resources. Previous studies focus on the allocation of resources in order to perfectly satisfy the demands seen in the current cycle, and the existing application placement algorithms are all based on applications. The existing application placement algorithms in literature assume that the consumption of system resources is proportional to the level of workloads submitted to the system. In this paper, we revealed that it may not be the case in some circumstances. Based on this observation, we design and implement an application placement framework, called EAPAC, for data centers. The developed framework is able to judiciously allocate to application servers a proper mixture of different types of application requests as well as an appropriate number of requests in each type. Further, we investigate the issue of resource conflicts among different applications when there exist concurrent requests in the system. We have conducted extensive experiments to evaluate the performance of the developed framework. The experiment results show that compared with the existing studies, EAPAC can improve the performance by 30% in terms of the reply rate. Especially, when there are concurrent requests in the system, the performance can be improved by 100%.

I. INTRODUCTION

Recently, data centers are of growing importance in hosting web applications [10]. Many empirical studies show that the web request rate is bursty and could fluctuate dramatically in a short period of time [17]. Accordingly, it is not cost-effective to over-provision the resources in data centers to satisfy the potential peak demands of all applications. Therefore, it is a crucial performance issue to judiciously manage the resources in data centers where a large number of applications are hosted.

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 their performance goals. This is so called dynamic hosting technology [1, 23]. Previous studies [9, 14, 21] 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 application and assume that the application instances are independent with each other. In the data center environments, however, there may exist a

large number of different types of applications, and moreover, different applications may compete for resources.

Another issue introduced in data servers is that the applications hosted in data servers are often data-intensive and the requested data may scale up and down rapidly [3, 24]. Under this circumstance, the CPU cycles may be wasted in waiting for I/O operations, although the CPU consumption is high. It is very difficult to accurately model the problem of busy waiting combined with resource competition among applications, because CPU consumption depends heavily on resource competitions, which in turn depends on the number of concurrent applications and their execution patterns at run time. In the existing studies, it is assumed that the CPU consumption increases linearly as the level of workload submitted to the system increases. The benchmarking experiments conducted in this paper show that this may not be case where the system is serving the concurrent applications requests.

To address the aforementioned problems, we design and implement an Enhanced Application Placement Framework (EAPAC) in this paper. There are two main components in EAPAC: 1) an application-level load balancer, which dispatches application requests to specific application servers and 2) an application server manager, which manages the resource allocation in individual application servers.

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

II. MOTIVATION

Existing application placement methods in literature [21] usually treat all types of requests equally and estimate a server's resource consumption by simply adding up the resource consumption caused by each individual request when it is run in the server alone. However, our studies show that this approach generates poor estimations when multiple requests are running in the system concurrently. This inaccuracy potentially leads to much reduced efficiency of request/application

scheduling.

We conduct the following benchmarking experiments to demonstrate this situation. The experiment platform consists of two IBM HS21 nodes taken from High Performance Computing Center¹ at 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). We implemented an emulated data intensive application on JBoss, which creates 100 MB data. One node is used to generate and send the requests using Apache Benchmark (AB)², 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 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. In all four patterns, only when the current batch of requests has been completed by the server, will Apache Benchmark send the next batch.

Figure 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 seen from this figure, the “usr” time, “iowait” time and the “sys” time dominate the processing time. The “iowait” time refers to the time when the CPU waits for the I/O operations to be completed. 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 time 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 $1.2 \times 8 = 9.6$ seconds, but $4 \times 8 = 36$ seconds.

The benchmarking result deviates from the assumption widely made in the existing studies of application scheduling, i.e., 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, this benchmarking result suggests that the CPU consumption is not necessarily proportional to the level of

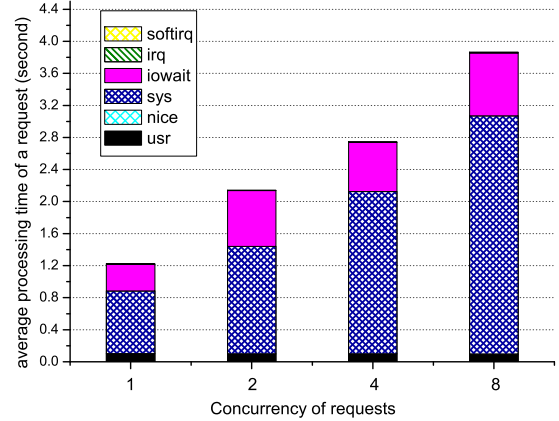


Fig. 1. CPU Consumption

workload submitted to the system. The underlying reason for the non-linearity may be due to the scheduling strategies employed in operating system. There are two possible solutions for this problem. One is to adjust the scheduling strategy in the kernel of the operation system. The other lies in the application level by judiciously allocating to application servers an appropriate number and combination of different types of requests. This paper focuses on the latter approach and develops a new application placement framework to address this issue.

III. DESIGN AND IMPLEMENTATION

This section presents the design and implementation of EAPAC. Figure 2 depicts the overall system, which consists of two main components: the Load Balancer (LB) and the Application Placement Manager (APM). 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 maps 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 optimal performance. The application placement decision is made by the APM based on the physical resource capacity, such as CPU power, I/O bandwidth, and the information of the requests, such as the number of the requests, resource consumption of the requests.

A. Algorithm for Load Balancer

In a web hosting system, a physical server can host a diverse collection of application servers and an application server can be deployed on multiple physical servers. Take Figure 2 as an example. Physical *Server 1* hosts *app1* and *app2*, while *app2* is deployed on *Server 1* and *Server 2*. The Load Balancer is at the application level, which recognizes the type of the requests and then dispatches the requests to corresponding applications for service. The load-balancing strategy is performed among the same type of application

¹HPCC, <http://grid.hust.edu.cn/hpcc>

²AB, <http://httpd.apache.org/docs/2.0/programs/ab.html>

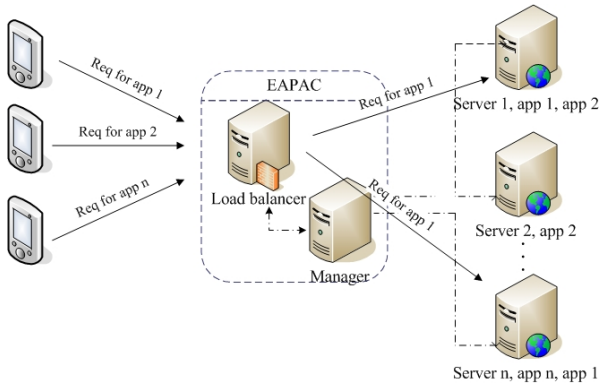


Fig. 2. System Model

servers. In this paper, an existing load balancing algorithm, referred to as Weighted Least-Connections (WLC) [20], is adopted in the load balancer. In WLC, performance weights are assigned to each application server on different physical nodes and the percentage of live connections is proportional to the weight value assigned to the server. For example, in Figure 2, if *app2* on *Server 1* is assigned with the weight of 0.3 and *app2* on *Server 2* assigned with 0.7, then 70% of the requests for *app2* will be sent to *Server 2*. The weight assignment is performed by the Manager in EAPAC.

B. Algorithm for Application Placement

A new algorithm is developed in this paper. The algorithm is employed by APM to make application placement decisions. Before application placement, two factors should be considered.

One is the concurrency level within an application server. 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. In this paper, we only address concurrency problem from the data-intensive application, and we use a simple way to study this: if the I/O operation for one request takes m seconds, the concurrency level of the application is $1/m$, and in turn EAPAC suggests less than $1/m$ requests in one second.

The other factor is the concurrency level among different application servers. Similar to the concurrent requests of the same type (i.e. being served in the same application), it can also cause the distortion of resource consumptions when different types of requests are being concurrently served by different applications in the same physical server. We take a variation of the scenario presented in section 2. There are two applications in one data center, *app i* and *app j*, with each request to *app i* and *app j* will create one 100MB file. In this situation, the concurrent requests to each application will lead to resource consumption distortion. The second step examines the relationship among the requests for different applications. The output of the study will be the exclusive tuple of the applications. The exclusive tuple of the two data-intensive

applications *app i* and *app j* is like this, $(i, j, 1/(m+n))$, if the I/O operation for one request for *app i* takes m seconds and the I/O operation for one request for *app j* takes n seconds. With the exclusive tuple, the idea behind EAPAC is to try to avoid parallel requests to these applications on one physical server.

The application placement is giving the weights for the real application server on a specific physical node. Among many resources, we choose CPU, memory, and disk I/O as the representative ones to be considered by EAPAC. To decide how to place application across back-end servers, we should predict how many requests will be sent by the client. In EAPAC, we predict the requests number with the number of the current time interval. In the application placement, we count all the requests and predict how many requests will be sent to the application servers in a specific interval. These predicted values are then used by EAPAC for the placement decision.

$$(i) \text{ maximize } \sum_{m \in M} \sum_{n \in N} L_{mn}$$

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

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

with

$$(iv) \forall m \in M, \forall n \in N, I_{mn} \in [0, 1]$$

$$(v) \forall m \in M, \forall n \in N, MA_{mn} \in (0, 1]$$

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

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

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

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

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

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

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

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

The objectives of application placement are:

1) Maximum CPU utilization, as shown in Eq. 1(i). The

L_{mn} refers to the CPU cycles allocated on node n for application m ;

- 2) Minimum application switching, as shown in Eq. 1(ii). The I_{mn} refers to the new application deployment matrix, and I_{mn}^* refers to the old deployment matrix;
- 3) Most balanced load among the physical nodes, as shown in Eq. 1(iii). The ρ refers to the average load of all physical nodes.

The application placement considers the following constraints:

- 1) The elements of the deployment matrix is either 0 or 1, shown in Eq. 1(iv);
- 2) The elements of the placement matrix is between 0 and 1, as shown in Eq. 1(v);
- 3) Eq. 1(vi) presents the relationship between the deployment matrix and the placement matrix;
- 4) If the element of the deployment matrix is 0, the resource utilization is 0, shown in Eq. 1(vii);
- 5) Without concurrency problem, the I/O utilization and the CPU utilization have linear relationship for one application, as shown in Eq. 1(viii);
- 6) The total memory demand of all applications on node n is less than the total memory of node n , shown in Eq. 1(ix). Υ_m refers to the memory demand of application m , and Γ_n refers to the memory capacity of machine n
- 7) The total CPU demand of all applications on node n is less than the CPU capacity of node n , as shown in Eq. 1(x). Ω_n refers to The CPU capacity of machine n ;
- 8) The total IO demand of all applications on node n is less than the IO capacity of node n , as shown in Eq. 1(xi). O_n refers to the IO capacity of node n , and $O_m n$ refers to the IO demand of application m on node n . With this constraints, the parallel requests can be avoided;
- 9) The total CPU demand for application m on all nodes is less than the total CPU demand of the application m , as shown in Eq. 1(xii);
- 10) The total IO demand for application m on all nodes is less than the total IO demand of the application m , as shown in Eq. 1(xiii);

The application placement is a variant of the Class Constrained Multiple-Knapsack problem [16]. We solve the problem by means of Greedy Algorithm [4]. The high level pseudo code is shown in Figure 3.

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 ID of an application server. The element of the matrix, ma_{ij} , represents the percentage of the requests submitted to application i that are forwarded to node j . If ma_{ij} is 0, then application i is not deployed on node j .

To get the placement matrix, we first studied the hardware resource capacity, such as CPU, memory and IO, as shown in Line 3,4 in Figure 3. 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 4,000 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 80%. With these parameters, the ‘calc better place matrix’ function will loop for ‘MaxLoop’ times to calculate a better matrix according to the objectives in Eq. 1.

In the ‘calc_better_place_matrix’ function, we compare the load between two nodes one by one (based on the existing placement matrix), as shown in Line 34, 35. 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' (Line 36 - 40); 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 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 43 to 52; 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 55 to Line 69, 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 Line 64 and 65. With these load movement, the load among nodes can be more balanced.

After executing the ‘calc_better_place_matrix’ function, the algorithm will compare the new matrix and the existing matrix, shown in Line 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 based on the objectives after executing 10 consecutive loop iterations, the algorithm will break the loop, as shown in Line 20, 21.

C. Implementation

Figure 4 illustrates EAPAC architecture. As mentioned in the previous section, there are two parts in EAPAC: *Load Balancer* and *Manager*. The *Load Balancer* forward the requests to the proper application server, and the *Manager* handles the application server placement on the back-end physical nodes. Besides, there is a testing node in EAPAC, which estimates the resource consumptions of the requests submitted to the application servers, including CPU, memory and I/O. In Figure 4, 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.

```

1: function place()
2: {
3:     // To get the hardware resource capacity, CPU, IO, Mem
4:     hd_resource = get_hardware_resource();
5:     //Forecast request rate and resource consumption in the future
6:     apps_request_resource_array = forecast_request_resource(
7:         get_apps_single_request_resources(),
8:         forecast_apps_request_rate() );
9:     /*compute load average on all physical servers */
10:    avg_load = Cmp_avg_Id(apps_request_resource_array, hd_resource);
11:    // Loop MaxLoop to get better matrix, MaxLoop by default 1000
12:    for(i=0; i<MaxLoop; i++) {
13:        /* adjusting place_matrix,
14:        minimize load standard deviation ; minimize switching application server */
15:        better_place_matrix = calc_better_place_matrix(
16:            place_matrix, apps_request_resource_array, hd_resource );
17:        if(better_place_matrix is better than place_matrix) {
18:            set place_matrix = better_place_matrix;
19:        }
20:        if(K times no_improvement) // K by default 10
21:            break_out_of_loop;
22:    }
23:    return place_matrix;
24: }
25: function calc_better_place_matrix(MA[ ][ ], R_A[ ], R[ ])
26: {
27:     // adapting application placement matrix
28:     // Compare load on each nodes
29:     // M is the number of applications, N is the number of physical nodes
30:     for(l = 1; l <= M; l++) {
31:         for(J = 1; J <= N; J++) {
32:             for(l' = 1; l' <= M; l'++) {
33:                 for(J' = 1; J' <= N; J'++){
34:                     // compare the total load on node J and node J'
35:                     if Sum J > Sum J' {
36:                         if(IO_l <>0) && (IO_l' <>0) {
37:                             /* get the ratio between the CPU consumption and
38:                             IO consumption of application l and application l' */
39:                             CI = CPU_l / IO_l ; // app l
40:                             CI' = CPU_l' / IO_l' ; // app l'
41:                             /* compare the load of application l on node J
42:                             and the load of application l' on node J' */
43:                             if (Load_lJ > Load_l'J') && (CI < CI') {
44:                                 K = Min(Load l'J', Load_lJ - Load l'J', (SumJ-SumJ') /2);
45:                                 Kio = K/CI';
46:                                 K' = Kio*CI;
47:                                 /* move workload of app l on node J with K quarter to node J'
48:                                 move workload of app l' with K' quarter to node J */
49:                                 Load_lJ -= K;
50:                                 Load_l'J += K';
51:                                 Load_lJ' += K;
52:                                 Load_l'J' -= K;
53:                                 //Change MA_lJ , MA_l'J , MA_l'J and MA_l'J accordingly;
54:                             }
55:                         }
56:                     }
57:                     else if (Load_lJ > Load l'J') {
58:                         K = Min(Load l'J', Load_lJ - Load l'J', (SumJ-SumJ') /2);
59:                         /* move some workload of app l on node J to node J' */
60:                         SWAP_l_IO = 0;
61:                         if(IO <> 0) {
62:                             CI = CPU_l / IO_l ; SWAP_l_IO = K/CI;
63:                             // SumJ'_IO refers to IO load on node J'
64:                             // M_IO_J' refers to IO capacity of nodeJ'
65:                             If( SWAP_l_IO > ( M_IO_J' - Sum_J'_IO)) {
66:                                 SWAP_l_IO=M_IO_J'-Sum_J'_IO;
67:                                 K = CI * SWAP_l_IO;
68:                             }
69:                         }
70:                         Load_lJ -= K;
71:                         Load_lJ' += K;
72:                         //Change MA_lJ , MA_l'J , MA_l'J and MA_l'J accordingly;
73:                     }
74:                 }
75:             }
76:         }
77:     }
78:     return MA[ ][ ];
79: }

```

Fig. 3. Pseudocode for Application Placement Algorithm

The arrows in solid lines represent the communication flow, while the arrows in dashed lines represent the movement of the requests

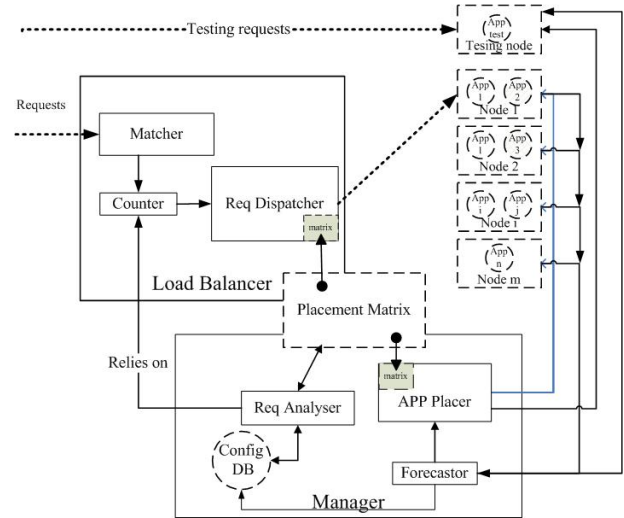


Fig. 4. EAPAC Architecture

Matcher: EAPAC maintains a list of the applications. For an in-coming request, the *Matcher* will select which type of application servers 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 *Manager* for making application placement decisions.

Req Dispatcher: The *Req Dispatcher* works as a router for requests, it forwards the request to a specific application server on a specific node, and establishes the connection between the application server and the clients. How and where to forward these requests is based on the matrix produced by the *Req Analyser*, which is described below. The load balancing algorithm is performed by this component.

Req Analyser: The *Req Analyser* has two functions: 1) analyzing the requests and 2) making application placement decisions. Based on the *Config DB*, which is described next, the *Req Analyser* analyzes the requests to estimate resource consumptions by the request. Also, the *Req Analyser* decides how to dispatch the requests. As demonstrated in section II, the consumption of physical resources (CPU cycles) may increase unproportionately when the concurrency level exceeds a certain threshold. Therefore, the dispatching principle is to maintain the concurrency level in an application server below the desired threshold. The *Req Analyser* will output a placement matrix, as shown in Figure 3. The Placement Matrix, *MA*, is the two-dimensional matrix as illustrated in Figure 3.

Config DB: The *Config DB* is a database which stores the following information: (1) the resource demands of application servers, e.g., how many cpu cycles and memory the applications consume; (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 administrators, e.g., giving more resources to the *app1*. (5) Rules generated by Req Analyser, e.g., *app1* and *app2* have resource confliction so that they should not be placed in the same physical server.

App Placer: The *App Placer* is the controller for application servers. The *App Placer* starts, resumes, or closes application servers on physical nodes based on the *Placement Matrix*.

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

D. Request Analysis and Application Placement

The *Req Analyzer* is the key component in EAPAC, which instructs how to host application servers in the data center. Similar to the traditional way illustrated in [21], it also analyzes whether there exist resource conflictions among the requests.

In EAPAC, we implement two versions of *Req Analyzer*. The first version performs application placement by assuming that all the applications consuming the CPU increases linearly as the number of requests increase. The second version implements the *Req Analyzer* in the following way: EAPAC first studies the resource consumption of the requests with a testing node, and tries to find the resource confliction, such as the I/O resource consumption, which will lead to extra CPU overhead. The studied resource confliction rules are stored in the *Config DB*. When a new request arrives, the *Req Analyzer* will check the *Config DB* to avoid the resource confliction of parallel requests. The two implementations are implemented with PHP. We will evaluate the performance in next section.

IV. PERFORMANCE EVALUATIONS

We conduct a set of experiments. All experiments are running on the dedicated IBM Blade cluster in HPCC as mentioned in section 2. All HS21 nodes are connected with a Gigabit Ethernet network. Each measurement is repeated 5 times, and the average is presented in the paper.

A. Throughput Evaluation

We use the following setup to evaluate the performance of EAPAC. We use nine HS21 nodes in HPCC: one HS21 node is used to run EAPAC, and the remaining eight to deploy applications. The HS21 node is equipped with a 2-way, 4-core 2.33 GHz Intel Xeon CPU E5345 and 8 GB memory. To illustrate the performance of EAPAC more clearly, we only take one core of Xeon CPU to run the *Load Balancer* and the *Manager* of EAPAC. The applications are running on the HS21 nodes with 8 cores, that is, the applications made fully use of the resources in physical nodes.

We use three types of applications in our experiments: (a) the transactional web e-Commerce benchmark TPC-W with

1000 items, which is intended to represent a realistic application; (b) a synthetic application that imposes a tunable CPU load (referred to as a CPU-bound application); (c) a synthetic application that imposes a tunable I/O load (referred to as IO-bound application). We do not choose any memory-intensive applications in our experiments, because we studied many real web applications and find that their memory consumption does 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 conflictions of parallel requests, we take data-intensive applications.

We run seven applications in total. Unless otherwise stated, the applications are run within the JBoss Application Servers. We test the CPU and I/O consumptions for these applications and the results are shown in Table I. The CPU consumption is measured with million CPU cycles, and the I/O consumption is measured with the time it takes to complete the I/O operation. We choose only one data-intensive application in our experiments, because EAPAC uses a hybrid application placement method, the parallel requests for one application can demonstrate exclusive requests for different applications. The application requests are generated by *httperf* [11].

TABLE I
APPLICATION SET UP

id	title	CPU(Million Cycle)	I/O (millisecond)
1	tpcw	10.45	0
2	appa	3.65	0
3	appb	24.27	0
4	appc	82.45	0
5	appd	192.98	0
6	appe	379.11	0
7	appf	259.36	98

The first experiment study the throughput of EAPAC. We compare the throughput of two application placement methods, the method in EAPAC and the method in literature [21]. The method in [21] (which we call Tang's method) also considers the application placement, but it does not address the challenges introduced by running concurrent requests

To be easy 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.

The first metric we use 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.

Figure 5(a) and 5(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 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 the threshold is the maximum request rate that the application can tolerate. By comparing Figure 5(a)

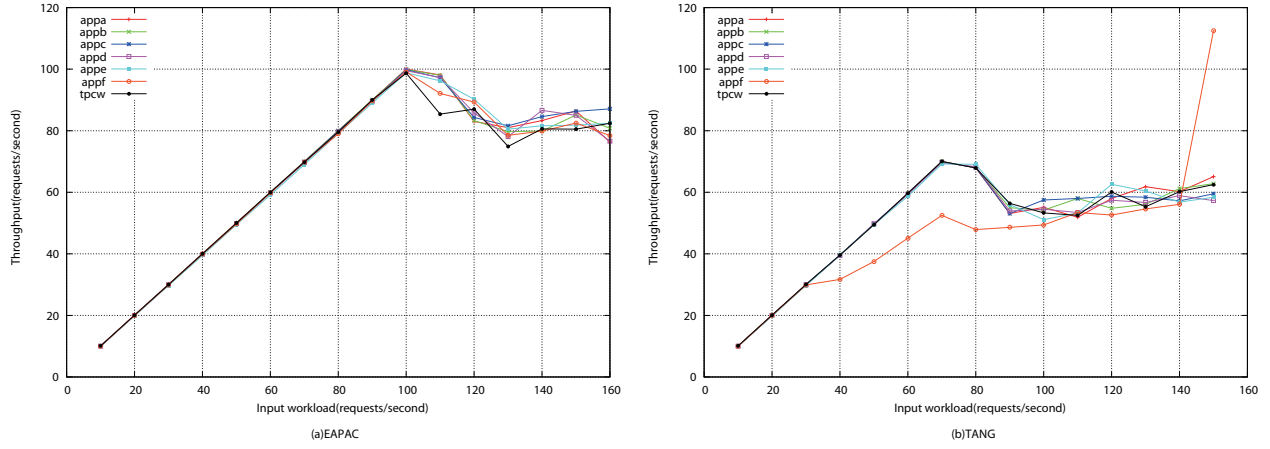


Fig. 5. Reply Rate

and 5(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 Figure 5 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 of the reply 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.

In Figure 5(b), the peak reply rate for *appf* is much larger than other applications, and the reply rate for *appf* change sharply. To analyze this situation, we record the error connections in our experiments, the recorded data is shown in Figure 6. From the application setup, we can finish each experiment in 15 seconds, so we run each experiment for 15 seconds. In Figure 6, the x-axis shows the request rate, and the y-axis shows the number of total errors in one experiment for 15 seconds. The Figure 6(a) shows the error rate of EAPAC, and Figure 6(b) shows the error rate of Tang’s method.

We can make the following observation from Figure 6:

(1) There are no error connections when the request rate is less than stable reply rate: in Figure 6(a), there are no error connections when request rate is less than 100; in Figure 6(b), there are no errors when request rate is less than 40. When the request rate is more than 40, the number of errors increase for *appf*. This shows that EAPAC has less error connections compared with Tang’s method. (2) When request rate is greater than 110, there are error connections for applications in EAPAC. That means that both methods do not perform stable. This is due to httpperf does not control the request, it only generates http requests as set up before experiments, which

leads to the error data mass. (3) The total number of errors in EAPAC is much less than with Tang’s method, as shown in Figure 6, when the request rate is 140, the total errors in Figure 6(a) is about 300, while the total errors in Figure 6(b) is about 2400. (4) In Figure 6(b), the number of errors for all applications changes dynamically, while in Figure 6(a), the number of errors for all applications grows slowly. This shows that even with some errors, the EAPAC system is still practical, while the applications placed by Tang’s method are out-of-service when request rate is larger than the stable reply rate. So we can conclude that EAPAC has better availability than Tang’s method.

The performance of EAPAC is also evaluated in terms of response time. In our experiment, the processing cycle of a request is as follows. Httpperf 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 httpperf. The time duration of a request’s processing cycle is defined as the request’s response time.

Figure 7(a) and 7(b) show the requests’ average response time in EAPAC and Tang’s method, respectively, as the request rate increases. As seen from Figure 7(a), the response time stays almost constant when the request rate is less than 110, then increases gradually except TPC-W. In Figure 7(b), the response time starts to increase dramatically when the reply rate reaches 80 (except *appf*, it 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 be observed from Figure 7 that the response time of all applications changes in a similar pace in our method, while the response time becomes volatile after the 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.

We also compare our methods with Tang’s method in terms of overhead. Overhead is measured as the time spent to reach the application placement solution. In EAPAC, the *Forecaster*

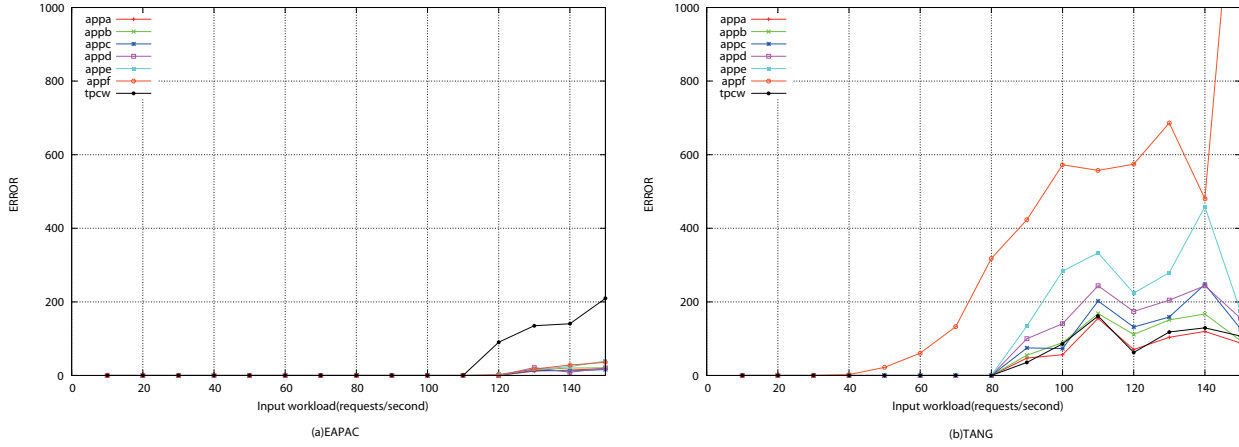


Fig. 6. Request Errors

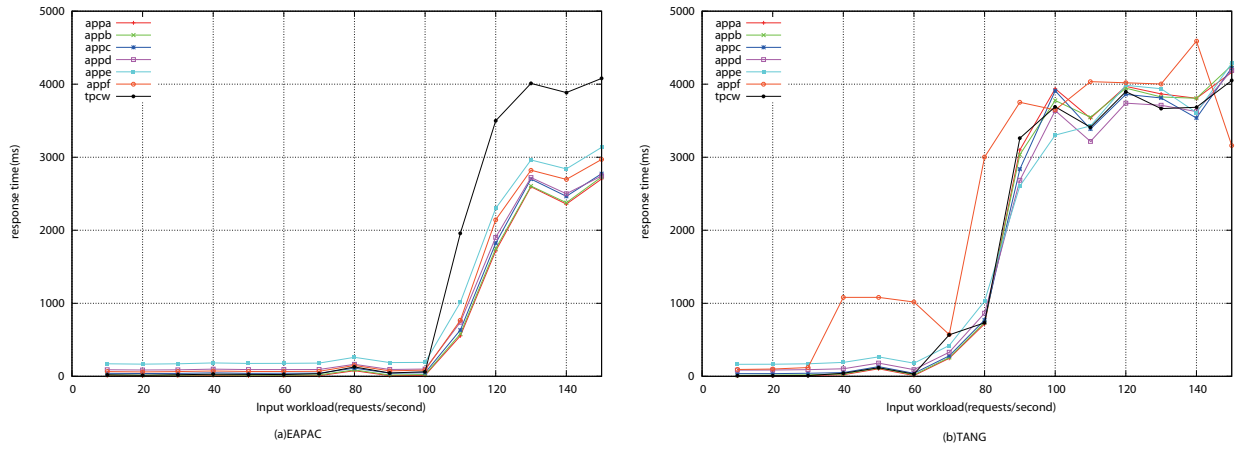


Fig. 7. Response Time

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 where EAPAC is executed. The script monitors the real-time CPU status from `/proc/stat`.

In the first experiment, we use one core of the HS21 node to run EAPAC. The core’s CPU utilization is plotted in Figure 8.

It can be seen from Figure 8 that EAPAC and Tang’s method have similar CPU utilization. This shows that although EAPAC outperforms Tang’s method, it is not at the expense of higher overhead.

Another interesting observation is that CPU utilization decreases when the request rate exceeds a threshold. This is because when the request rate 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 Figure 8 and Figure 5, it can be seen that the threshold in Figure 8 is very close to the request rate that makes the reply rate start fluctuating, which confirms the above analysis.

B. Scalability Evaluation

We also conducted a set of experiment to investigate the scalability of EAPAC. We run the same set of applications as in the previous experiment on the IBM HS21 Blade cluster. Similar to the previous experiment, we take one core of Xeon CPU to host the *Load Balancer* and the *Manager*. In order to study the scalability, we use different number of HS21 nodes to host the application. Again, the application requests are evenly generated for all applications by `httperf`.

As we can see in the previous experiment, the stable reply rate can reflect the real throughput of the application servers. so Figure 9 only plots the maximum stable reply rate with different number of servers. For the sake of clarity, the reply rate of both TPC-W and *appf* grows linearly with the number of nodes. This shows that EAPAC is scalable in terms of reply rate. It can also be seen from the figure that when the number of nodes is less than 4, the reply rate of EAPAC and Tang’s method is very close, while when the number of nodes is greater than 4, the reply rate of EAPAC is greater than that of Tang’s method. This suggests that EAPAC is more scalable.

Another observation is that in EAPAC, the reply rate of *appf*

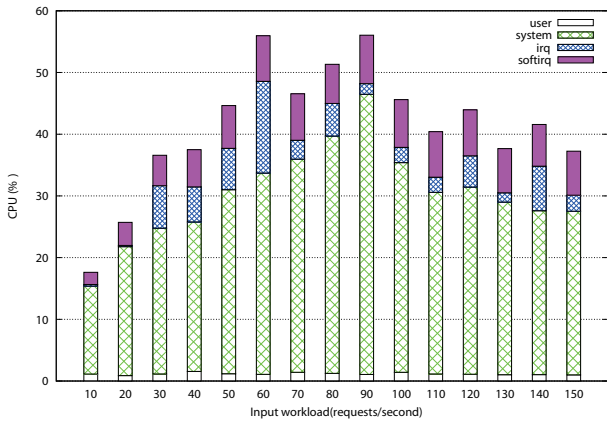
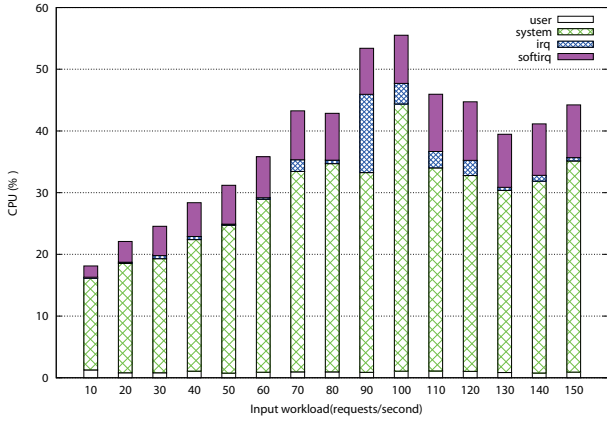


Fig. 8. CPU Load

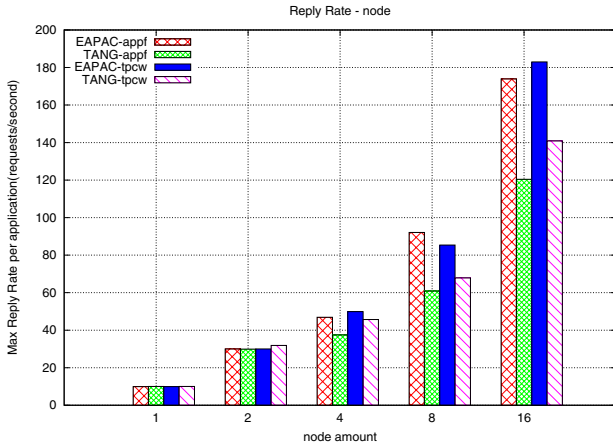


Fig. 9. Scalability with Throughput

increases a little faster than that of TPC-W. The phenomenon is due to the concurrency problem. 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 problem caused by the concurrent requests.

As we discussed in the previous experiment, the response

time for requests has great impact to the QoS of web servers. We have studied the scalability with reply rate. According to the reply rate for 16 nodes, the reply rate for EAPAC is about 30% higher than Tang’s method. To see the details of the processing requests, we generate the application with the request rate at 150, and we plot the raw data of the response time for all request in 15 seconds, which is shown in Figure 10. The x-axis shows the time line of the experiment, and the y-axis shows the response time. A point in Figure 10 corresponds to the response time of a request.

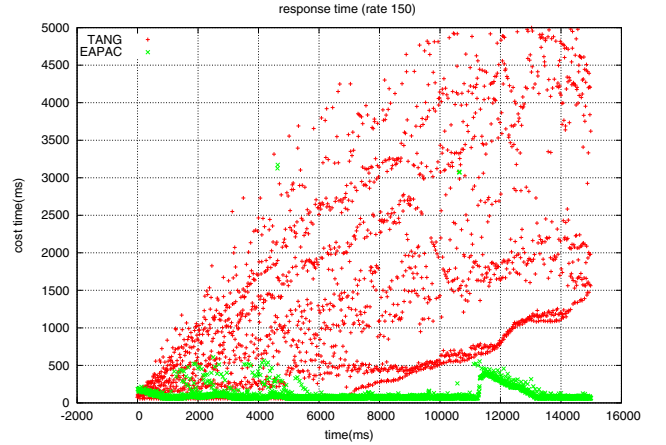


Fig. 10. Scalability with Response Time

The following observations can be made in Figure 10:

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

(2) When the timeline goes to 14000ms, the shortest response time for Tang’s method is larger than 1500ms and the average response time is about 2000ms, while for EAPAC, the average response time is still about 100ms. This result once again shows that EAPAC performs better than Tang’s method.

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

In summary, the experimental results in this section demonstrate that EAPAC is superior to traditional methods, especially when there exist concurrent requests in the system.

V. RELATED WORK

There is a growing interest in the efficient management of data centers and hosting platforms. Some of them target the algorithms of application hosting. For example, Kimbrel et al. [9] and Tang et. al. [21] proposed the algorithms of

dynamic application placement in response to the changes in application demands. Their work is quite close to EAPAC. As illustrated in section 1, EAPAC analyzes the requests and makes application placement decisions based on both application servers and the requests' impact among them; The algorithm proposed by Urgaonkar et al. [23] 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, and not try to minimize placement changes. The placement problems have also been modelled using various approaches, including bin packing, multiple knapsack, and multi-dimensional knapsack [16]. These modelling technique is the theoretical foundation of the application placement strategy in EAPAC.

Some research works target different resource-sharing environments. 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.

The placement of application replicas is one of the key aspects in a hosting platform. It got much attention in the past. Previous work in this area mostly focused on the algorithms for determining the number and location of the replicas, either in the wide area network [7, 8, 14, 15] 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 is determined [1]. Some key technologies was adopted in [1], such as prefetching [12]. 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. [13]. Unlike our work, the authors targeted the environments in which application instances are organized in the application-level clusters and hosted on virtual machines.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we design, implement, and evaluate an application placement framework, EAPAC. EAPAC analyzes the web requests before making application placement decisions, aiming to avoid resource competition among different applications. We carry out extensive experiments for performance evaluation. Compared with Tang's method in literature, EAPAC improves throughput by 30% for all applications we tested, and achieves more than twice improvement in throughput for the applications serving concurrent requests. Moreover, the workload is better balanced by EAPAC and in turn 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 to enable EAPAC to support resource provision [18, 22]; (2) Currently, EAPAC handles the resource competition from different

applications. We will address the resource co-allocation for applications [5, 19].

ACKNOWLEDGMENTS

This paper is partly supported by the National 973 Basic Research Program of China under grant No. 2007CB310900, NSFC under grant No. 60973037, and 61073024, NCET Program Grant NCET-07-0334, and Wuhan Chenguang Program under grant No. 201050231075.

REFERENCES

- [1] Z. Al-Qudah, H. Alzoubi, M. Allman, M. Rabinovich, and V. Liberatore. Efficient application placement in a dynamic hosting platform. In *WWW*, 2009.
- [2] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *HPDC*, 2003.
- [3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
- [4] G. Gutin, A. Yeo, and A. Zverovich. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the tsp. *Discrete Applied Mathematics*, 117:81–86, 2002.
- [5] G. C. Hunt and M. L. Scott. The coin automatic distributed partitioning system. In *OSDI*, 1999.
- [6] X. Jiang and D. Xu. Soda: A service-on-demand architecture for application service hosting utility platforms. In *HPDC*, 2003.
- [7] J. Kangasharju, J. W. Roberts, and K. W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4), 2002.
- [8] M. Karlsson and C. T. Karamanolis. Choosing replica placement heuristics for wide-area systems. In *ICDCS*, 2004.
- [9] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Svirdenko, and A. Tantawi. Dynamic placement for clustered web applications. In *WWW*, 2006.
- [10] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for oltp and transactional web applications. In *ICDE*, 2004.
- [11] D. Mosberger and T. Jin. httpperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.
- [12] A. E. Papathanasiou and M. L. Scott. Aggressive prefetching: an idea whose time has come. In *HotOS*, 2005.
- [13] H. Qian, E. Miller, W. Zhang, M. Rabinovich, and C. E. Wills. Agility in virtualized utility computing. In *VTDC*, 2007.
- [14] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *INFOCOM*, 2001.
- [15] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *ICDCS*, 1999.
- [16] H. Shachnai and T. Tamir. On two class-constrained versions of the multiple knapsack problem. *Algorithmica*, 29(3):442–467, 2001.
- [17] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *OSDI*, 2002.
- [18] B. Sotomayor, K. Keahey, and I. Foster. Combining batch execution and leasing using virtual machines. In *HPDC*, 2008.
- [19] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI*, 2005.
- [20] S. G. T. Schroeder and B. Ramamurthy. Scalable web server clustering technologies. *IEEE Network*, 14(3):38–45, 2000.
- [21] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *WWW*, 2007.
- [22] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1), March 2008.
- [23] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *OSDI*, 2002.
- [24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.