Improving Application Placement Load-Balancing for Cluster-based Web Applications

Chen Tian, Hongbo Jiang, *Member, IEEE,* Arun Iyengar, *Senior Member, IEEE,* Xue Liu, *Member, IEEE,* Zuodong Wu, Jinhua Chen, Wenyu Liu, *Member, IEEE,* and Chonggang Wang, *Senior Member, IEEE*

***Abstract*—Dynamic application placement for clustered web applications heavily influences system performance and quality of user experience. Existing approaches claim that they strive to maximize the throughput, keep resource utilization balanced across servers, and minimize the start/stop cost of application instances. However, they fail to minimize the worst case of server utilization; the load balancing performance is not optimal. Another problem is that, some applications need to communicate with each other; we call such applications *dependent applications*. There is significant system cost if the instances of *dependent applications* are placed in different servers, and our work takes this cost into consideration.**

This paper has two major contributions. First we investigate how to minimize the resource utilization of servers in the worst case, aiming at improving load balancing among clustered servers. We formulate this new optimization objective as a min-max problem; a novel framework based on binary search is proposed to detect an optimal load balancing solution. Second, we consider communication cost among applications in making placement decisions. Since applications can communicate, it is desirable to put applications which communicate with each other on the same node. We define system cost as the weighted combination of both placement change and inter-application communication cost. Our application placement framework attempts to maximize the number of instances of *dependent applications* that reside in the same set of servers. Extensive experiments have been conducted and effectively demonstrate that: 1) the proposed framework achieves a good allocation for clustered web applications. In other words, requests are evenly allocated among servers, and throughput is still maximized; 2) both placement change costs and inter-application communication costs are low; 3) our algorithm approximate an optimal solution within polynomial time and is promising for practical implementation in real deployments.

***Index Terms*—Load balancing, application placement, algorithm design, cluster-based service, Class Constrained Multiple-Knapsack Problem.**

I. Introduction

WEB applications make it possible to deliver critical services provided by organizations directly to clients [1], [2], [3]. Modern web applications typically run on top of a middleware system, which is responsible for processing client requests and for allocating resources at a high rate [4]. Clustering technology enables middleware systems to achieve high degrees of scalability and availability. On the other hand, it also poses great challenges in scalable and high performance computing. For instance, it is often cost-inefficient when designing data centers to simultaneously handle the potential peak demands of all the applications [5], due to the dynamical fluctuation of request rate. As a result, the middleware systems are supposed to allow dynamic resource allocation to meet different performance requirements from diverse applications. The problem becomes *dynamic application placement*: given a set of machines[1] with constrained resources[2] and a set of Web applications with dynamically changing demands, how many instances of each application should be run, and where should they be placed?

In past work, this problem has been generally formulated as a variant of the Class Constrained Multiple-Knapsack Problem [7], [8], with multiple objectives such as maximizing the throughput of the whole system, and minimizing the disturbance due to application instance placements start/stop, just to name a few. The scheme in [6] is advantageous over other placement algorithms in terms of computational scalability, application satisfied demand and placement change. There are still two problems. First, its load balancing is not optimized: some servers could be heavily loaded after executing the application placement algorithm. We illustrate this problem by the server load distribution results in Figure 1. The detailed context of this experiment is given later in Section IV-B, the key points are that the total system load is 50% averaged from 500 independent experiments. As we can see from the results, around 22% of the servers have utilization close to the whole system load; however, there are a number of servers that have load higher than 80%, and some of them even have 100% utilization. As a result, the response times on the servers with high utilization could be significantly increased [9], and the clients served by these servers may be exposed to unnecessary long response latency, which is unfavorable for real-time web-based applications such as stock trading and multimedia streaming. To alleviate this problem, the *worst case* of individual server utilization[3] should be minimized and load balancing in the whole system should be improved.

Another problem with previous works is their cost model:

The corresponding author is Hongbo Jiang: hongbojiang2004@gmail.com.

C. Tian, H. Jiang, Z. Wu, J. Chen and W. Liu are with Huazhong University of Science and Technology, Wuhan, China, 430074.

A. Iyengar is with IBM T.J. Watson Research Center, Hawthorne, NY 100086.

X. Liu is with University of Nebraska-Lincoln, Lincoln, NE, 68588-0150.

C. Wang is with NEC Laboratories America, Princeton, NJ 08540.

[1]In this paper, we alternatively use the terms of *machine*, *nodes*, or *server* unless explicitly explained. All these terms are referred to as the web servers that handle the client requests.

[2]Like [6], this paper only considers CPU and memory resources. However, our framework can be easily extended to deal with other types of resources.

[3]In this paper, we refer to the *worst case* of individual server utilization as the highest CPU utilization among all machines.

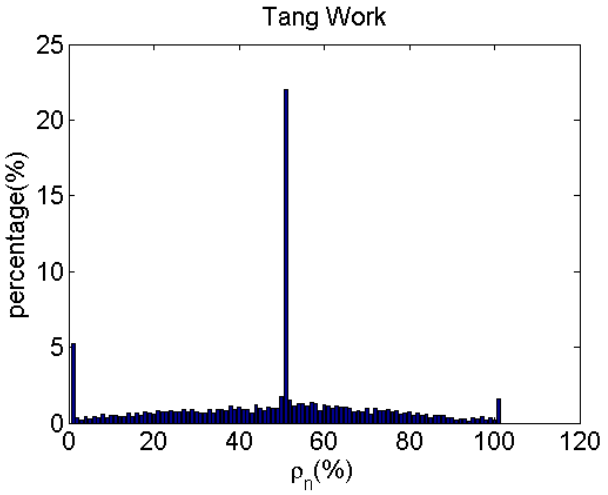


Fig. 1. Server Load Distribution of Tang's work with configuration (L_{mem}= 0.6,L_{cpu}=0.5,reset-all-apps).

only start/stop cost of application instances are considered. Some web applications have extensive communications among themselves, which we called *dependent applications*. In this paper, we assume that these dependencies are paired: application *a*'s instances need data from application *b*'s instances, and vice versa. It is desirable to put applications which communicate with each other on the same set of servers (i.e.,localize the inter-application communication) as much as possible; the communication can then be efficiently accomplished by inter-process mechanisms such as shared memory or local sockets. By contrast, if instances of *dependent applications* are placed in different servers, significant system cost can be incurred: first, this utilizes network bandwidth; second, the communication dependency among application instances may slow down the applications and degrade their performance. To sum up, the system cost should also model the inter-application communication cost; by minimizing this cost in the algorithm, we can achieve localize the inter-application communication as much as possible.

In this paper, we propose an enhanced application placement framework, which complements previous works and has contributions in the following aspects.

- **Improving load balancing**. Previous studies [5], [6] presented approximation algorithms to deal with multiple optimization objectives in terms of throughput, placement changes, and load balancing. While acknowledging the effectiveness of the problem statement with those objectives, in this paper, we propose to integrate a new optimization objective: limiting the worst case of each individual server's CPU utilization, formulated by a min-max optimization problem. By doing so, the system load balancing performance is greatly improved.
- **Localizing inter-application communication**. We look at communications among applications and introduce the concept of *dependent application*. Since some applications need to communicate with each other, it is desirable to put applications which communicate with each other on the same node. We define system cost as the weighted combination of both placement change and inter-application communication cost. Our application placement framework attempts to maximize the number of instances of *dependent applications* that reside on the same set of servers.
- **Practical approximation algorithm and extensive evaluation results**: In our enhanced framework, the placement algorithm is based on binary search to dynamically probe the optimal application placement solution. It is desirable that our placement algorithm is able to find a near optimal solution within polynomial time. We conduct extensive evaluations, and our results demonstrate that compared with state-of-the-art algorithms, our framework achieves better allocation for clustered web applications, more balanced server load, and less system cost.

The remainder of this paper is organized as follows. Section II discusses related work. Our enhanced framework is proposed in Section III. Performance evaluation of our new algorithm is presented in Section IV. Section V concludes the paper.

II. Related Work

Our work extends the research in application placement described in [6], [10], [11]. With our experiments we demonstrate that a utility-driven system outperforms demand-based approach in terms of application satisfaction fairness. Among them, the framework in [6] for dynamic application placement is a representative example and it outperforms other existing techniques.

Fig. 2 depicts the typical diagram of the Web application control loop for application placement. The system is composed of front-end Request Router, Application Placement Controller, Placement Executor, Back-end Machines, and Applications. The request router receives external requests and forwards them to the application instances. The placement controller periodically calculates a placement solution that optimizes certain objective functions, and then passes the solution to the placement executor to start and stop application instances accordingly.

A popular approach to dynamic server provisioning is to allocate full machines to applications as needed [12], which does not allow applications to share machines. The algorithm proposed in [13] allows applications to share machines, but it does not change the number of instances of an application and only considers one bottleneck resource.

Placement problems have also been studied in the optimization literature, including bin packing, multiple knapsack, and multi-dimensional knapsack problems [14]. The special case of the problem with uniform memory requirements was studied in [7], [8], and some approximation algorithms were proposed. Meta-scheduling algorithms for grid and parallel computing also deal with the placement problem [15].

A disk load balancing criterion which combines a static component and a dynamic component is described in [16]. The static component decides the number of copies needed for each movie by first solving an apportionment problem and then solving the problem of heuristically assigning the copies onto

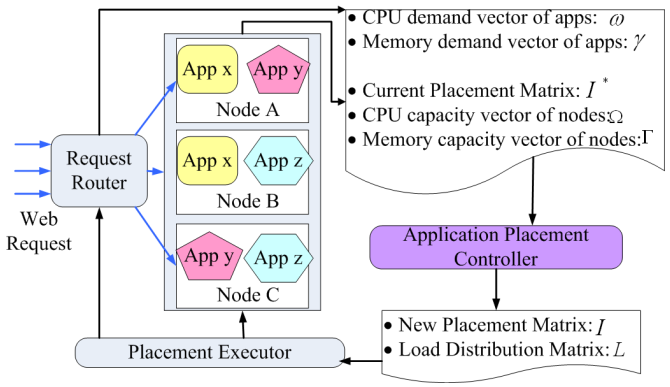


Fig. 2. Control loop for application placement (Adapted from [6]).

storage groups to limit the number of assignment changes. The dynamic component solves a discrete class-constrained resource allocation problem for optimal load balancing, and then introduces an algorithm for dynamically shifting the load among servers (i.e. migrating existing video streams). A placement algorithm for balancing the load and storage in multimedia systems is described in [17]. The algorithm also minimizes the blocking probability of new requests.

Xueyan's work [18] and Hsiangkai's work [19] are focused on content replica placement and request routing in content distribution networks; these works are related to our topic. However, in our scenarios all servers are considered centrally located together instead of geographically distributed.

III. Enhanced Application Placement Framework

A. Symbols

Table I lists symbols used in this paper[4]. The inputs to the placement controller include the current placement matrix I^*, the CPU and memory capacities of each machine (Ω_n and Γ_n), and the CPU and memory demands of each application(ω_m and γ_m). Both values correspond to only the workload controlled by the placement controller. Capacity used by other workloads should be subtracted prior to invoking the algorithm. The outputs of the placement controller are the updated placement matrix I and the load distribution matrix L.

The system has two main costs: one is the start/stop cost of application instances when performing placement changes; the other is the communication cost among *dependent applications*. Let I^* denote the old placement matrix, and I denote the new placement matrix. Let c demote the number of placement changes. We have:

$$c = \sum_{m \in M} \sum_{n \in N} \left| I_{m,n} - I^*_{m,n} \right|$$

It is hard to directly model the inter-application communication cost. There are some non-trial questions: for a specific pair of *dependent applications*, how to specify the cost for communication between instances on the same node and cost for communication between instances on different nodes? for different pairs of *dependent applications* which have different

[4]The table directly inherits symbols of previous works and is extended with new definitions.

TABLE I
Symbols used in this paper

N	The set of machines.
n	One machine in the set N.
M	The set of applications.
m	One application in the set M.
I	The placement matrix. $I_{m,n} = 1$ if application m is running on machine n; $I_{m,n} = 0$ otherwise.
L	The load distribution matrix. $L_{m,n}$ is the CPU cycles per second allocated on machine n for application m. L is an output of the placement algorithm; it is not measured from the running system.
Γ_n	The memory capacity of machine n.
Ω_n	The CPU capacity of machine n.
γ_m	The memory demand of application m, i.e., the memory needed to run one instance of application m.
ω_m	The CPU demand of application m, i.e., the total CPU cycles per second needed for application m throughout the entire system.
ω'_m	The satisfied demand of application m by previous stage.
	New definitions
ρ	the utilization of the entire system.
ρ_n	the utilization of machine n.
p	the utilization fraction parameter of the entire system.
p^*	the optimum utilization fraction parameter of the entire system.
c	the calculated number of needed placement change.
p^-	the lower bound of the utilization fraction parameter.
p^+	the upper bound of the utilization fraction parameter.
R	The application dependence matrix. $R_{m1,m2} = 1$ if application $m1$ and $m2$ have inter communication with each other.
h	the localized fraction of all inter-application communication.
s	the system cost.

sorts of communication patterns, how to differentiate their cost parameters? To make the problem tractable, we exploit an observation of *dependent applications* that an instance's inter-application communication cost is mostly proportional to its workload. Instead of modeling communication costs of all those pairs of *dependent applications* one by one, we record the total wordload of these applications, and calculate how much workload reside in the same set of servers:

$$h = \frac{\sum_{n \in N} \sum_{m1 \in M} \sum_{m2 \in M} I_{m1,n} I_{m2,n} R_{m1,m2} (L_{m1,n} + L_{m2,n})}{\sum_{m1 \in M} \sum_{m2 \in M} R_{m1,m2} (\omega_m 1 + \omega_m 2)}$$

Thus, based on out assumption, h indirectly denotes the localized percentage of all inter-application communication.

It is obvious that, c should be minimized and h should be maximized. The next question is how to combine c and h into a single system cost. We define system cost as the weighted combination of both placement change and inter-application communication cost. Let c be normalized by application number $|M|$, and let $(1 - h)$ as the non-localized percentage of all inter-application communication, we use weight values

α and β to adjust their weights in the system cost s.

$$s = \alpha * \frac{c}{|M|} + \beta * (1 - h)$$

Note that α and β should be adjusted accordingly for individual systems, based on the administrator's estimation of the importance of placement change cost v.s. inter-application communication cost, especially the percentage and magnitude of applications that communicate with other applications [5]. Even with rough modeling, we still get reasonably good results in the evaluation, as shown in Section IV.

In addition, ρ and ρ_n denote the utilization of the entire system and the utilization of an individual machine n respectively. From the view of load balancing, ρ_n should stay close to ρ.

$$\begin{aligned} \rho_n &= \frac{\sum_{m \in M} L_{m,n}}{\Omega_n} \\ \rho &= \frac{\sum_{m \in M} \sum_{n \in N} L_{m,n}}{\sum_{n \in N} \Omega_n} \end{aligned}$$

B. New Formulation of the Application Placement Problem

Application placement addresses the problem of how to distribute applications among multiple servers in order to maximize performance. Apart from [6], this paper is based on the basic idea that *the worst case load performance of individual machines should be minimized first*. From the users' point of view, it is undesirable to experience a long response time due to high load at a server. Intuitively, if the worst case of each individual machine utilization is minimized, applications will be distributed more evenly across machines. Thus, our approach can improve the balance among the servers.

The placement controller should first attempt to find a placement solution that maximizes the total satisfied application demand. Secondly, it also tries to minimize the total system cost, including the number of application starts and stops, because placement changes disturb the running system and waste CPU cycles, and the cost of *dependent application* communications. Our approach also minimizes the worst case load utilization to balance the load across machines. These objectives are listed in the formulated problem below:

$$\begin{aligned} &(i)\text{Maximize: } \sum_{m \in M} \sum_{n \in N} L_{m,n} \\ &(ii)\text{Minimize: } s \\ &(iii)\text{Minimize: } p = \max_{n \in M}(\rho_n) = \max_{n \in M}\left(\frac{\sum_{m \in M} L_{m,n}}{\Omega_n}\right) \\ &(iv)\text{Minimize: } \sum_{n \in N} |\rho_n - \rho| \\ &= \sum_{n \in N} \left| \frac{\sum_{m \in M} L_{m,n}}{\Omega_n} - \frac{\sum_{m \in M} \sum_{n \in N} L_{m,n}}{\sum_{n \in N} \Omega_n} \right| \end{aligned} \quad (1)$$

[5] In our current implementation, we simply assign them equal weight; hence $\alpha = \beta = 100$

Subject to:

$$\begin{aligned} &(a) \sum_{m \in M} \gamma_m I_{m,n} \le \Gamma_n, \forall n \in N \\ &(b) \sum_{m \in M} L_{m,n} \le \Omega_n, \forall n \in N \\ &(c) \sum_{n \in N} L_{m,n} \le \omega_m, \forall m \in M \\ &(d) I_{m,n} = 0 \Rightarrow L_{m,n} = 0, \forall m \in M, \forall n \in N \\ &(e) L_{m,n} \ge 0, I_{m,n} \in \{0, 1\}, \forall m \in M, \forall n \in N. \end{aligned} \quad (2)$$

Constraint set (a) specifies that the memory demand of all applications in machine n should not exceed the memory capacity; (b) specifies that the total CPU cycles consumed in each machine should not exceed the machine's CPU capacity; (c) specifies that the total allocated CPU cycles to an application should not exceed its demand; (d) specifies that an application can be serviced in a machine if and only if it is stored at that machine; (e) define the variables' feasible range.

This problem is NP hard, and we develop an approximation algorithm to solve it. Before presenting its details, we first give a high-level description and outline the key ideas behind the algorithm. Observe that the satisfied CPU demand provided by each single machine is directly confined by the constraints of Equation 2(b). If we scale all Ω_n down by the same ratio $p \le 1$, the constraints of Equation 2(b) are changed to

$$(b) \sum_{m \in M} L_{m,n} \le p * \Omega_n, \forall n \in N, \rho < p \le 1 \quad (3)$$

Then if we solve the placement problem, it is guaranteed that $\rho_n \le p$ from the definition of ρ_n.

Theorem 1: The decrease of p converges to p^*.

Proof: Suppose $\rho \le p_1 \le p_2 \le 1$ and denote a solution of p_1 by I_1 and L_1 respectively. Since all other constraints need to be met, we have:

$$\sum_{m \in M} L_1 \le p_1 * \Omega_n \le p_2 * \Omega_n, \forall n \in N$$

Hence a solution of p_1 is also a solution of p_2. ■

A p value is an *acceptable* value if all constraints are satisfied. Above all, the total satisfied application demand should still be maximized. A lower p value reduces the feasible region compared with the original formulation. To fulfill the demand, more placement changes may be needed.

We use the ***MaxDemandMinChange*** algorithm in [6] as the baseline algorithm of our framework. After running the baseline algorithm first, we get the maximum demand that can be satisfied *max_demand*′ and the corresponding system cost s'. The next step is to obtain p. The original objectives (i) and (ii) now can be transformed to the constraints: max demand should be strictly satisfied and the system cost should be controlled. For each new formulation of p, our framework attempts to optimize objectives (iii) and (iv).

We enhance the baseline algorithm with the optimization of communication cost in the basic load-shift and placement change procedures (extension presented in Section III-C1). Our algorithm repeatedly probes the minimum p in multiple rounds. In each round, it fixes a p value and uses the **MaxDemandMinCost** algorithm to calculate the maximum total

application demand that can be satisfied *max_demand* by the current p value together with the system cost s to see if this p value is acceptable. p will finally converge to the optimum value p^*.

C. The Full Placement Algorithm

The next problem is how to optimize p as quickly as possible. We develop an algorithm based on Binary Search. Instead of a blind probe, the upper bound of p, p^+, and the lower bound of p, p^- are calculated iteratively. In each iteration, either p^+ or p^- are updated. The full high level pseudo code is depicted in Algorithm 1 where the function **MaxDemandMinCost** is shown in Algorithm 2 and the function **BoundAcceptable** is shown in Algorithm 3. The general process is composed of three main building blocks: initialization, iterative optimizing and final rebalancing.

Algorithm 1 PlaceFrame()

Require: : output: $L''_{m,n}$: the load distribution matrix; $I''_{m,n}$: placement matrix.
1: $p^+ = 1$, $p^- = \rho$, $p = 1$;
2: **MaxDemandMinChange**(max_demand', s');
3: **while** $\frac{p^+ - p^-}{p^+} > \varepsilon$ **do**
4: $p = \frac{p^+ + p^-}{2}$;
5: **MaxDemandMinCost**(max_demand, s, $L'_{m,n}$);
6: **if BoundAcceptable**() **then**
7: $p^+ = p$; //decrease the upper bound of p value
8: **else**
9: $p^- = p$; //increase the lower bound of p value
10: **end if**
11: **end while**
12: $p' = p^+$;
13: $\omega'_m = \sum_{n \in N} L'_{m,n}$;
14: **Final_Rebalancing**();
15: $I''_{m,n} = I'_{m,n}$;

Algorithm 2 MaxDemandMinCost()

Require: Input: p: the maximal machine utilization threshold.
Output: calculated max satisfied demand *max_demand* and the cost s.
$L_{m,n}$: the load distribution matrix.
1: **for** i = 0 to K // K=10 by default; **do**
2: calc_max_demand_satisfied_by_current_placement ();
3: **if** all_demands_satisfied **then**
4: **if** worst_case_satisfied **then**
5: //the maximal machine utilization
6: //is less than the given threshold p;
7: break out of the loop;
8: **end if**
9: **end if**
10: ... // we omit the details about placement changes.
11: **end for**

1) Initialization: In the initialization phase(lines 1-2 in Algorithm 1), we set p^- with ρ, and p^+ with 1, $p = 1$. After performing the function **MaxDemandMinChange**() described in [6], we obtain max_demand' and c', which will be used as constraint parameters in the later iterative optimizing phase.

For simplicity, we omit details about the algorithm in this paper. Briefly speaking, it strives to probe the maximal application demand by means of iteratively making placement changes in order to increase the total satisfied demand, for instance, stopping "unproductive" application instances and starting useful ones. We also made some minor modifications

Algorithm 3 BoundAcceptable()

Require: Input: max_demand' and *max_demand*: the demand satisfied before and after updating worst case machine utilization constrains; Cost s' and s: the system cost before and after updating worst case machine utilization constrains.
1: **if** $max\_demand' == max\_demand$ **then**
2: **if** BOUNDED **then**
3: **if** $s > s'$ **then**
4: return FALSE;
5: **end if**
6: **end if**
7: return TRUE;
8: **end if**
9: return FALSE;

to this function. For example, by integrating an additional optimization objective, the system stops making application changes only when the worst case (the maximal machine utilization) is less than a given threshold p.

2) Iterative Optimizing: The system attempts to iteratively decrease the upper bound or increase the lower bound of p (lines 3-11 in Algorithm 1) in order to approximate an optimal solution. The revised problem with parameter p is then addressed by continuously identifying whether the updated value of p is acceptable or not. If the solution is acceptable, p^+ is updated; otherwise p^- is updated. As such, the difference between p^+ and p^- is decreased. That is, the system strives to find the optimal value for p based on binary search. This loop executes until the difference between p^+ and p^- is small enough. For an $\varepsilon-$approximation of the optimum value p^*, the iterative search can be completed in $O(log_2(1/\varepsilon))$ rounds [20]. The final p value is a good approximation to constrain the worst case of the machine utilization across all machines.

MaxDemandMinCost is an extension of the baseline algorithm to support the minimization of inter-application communication cost (in addition to instance start/stop cost). We currently can deal with paired-dependency: two applications have communications with and only with each other; we plan to extend the algorithm to more complex dependency relationships in the future.

The main improvement to **MaxDemandMinChange** is the load-shifting part. For example, application a and b are dependent on each other. The **MaxDemandMinCost** algorithm identifies all those servers that already contain both a and b instances, say, Group 1; and all those servers that contain either a or b instances, say, Group 2. After sort the machines by an increase order of residual memory, our algorithm selects servers in Group 1 and puts them at the head of the list, and those servers in Group 2 in the tail of the list. The intuition here is to shift the load of *dependent application* from Group 2 to Group 1. Instances of *dependent application* in Group 2 servers are more likely to be idle after the load-shifting phase and hence easier to deal with in the placement change phase. In the placement change phase, each time we change at most one instance per dependent pair, i.e., at most one server per pair moved to Group 1. For each pair of *dependent applications*, we find one server in its Group 2 with the most appropriate residual memory and the largest idle CPU power, add the missing application's instance, and move it to Group 1.

3) Final Rebalancing: The last step of the algorithm is Final Rebalancing(line 14 in Algorithm 1). In [6], the final load-balancing component from [10] was used, which moves the new application instances across machines to balance the load, while keeping the total satisfied demand and the number of placement changes the same.

While the basic idea of our Final Rebalancing component is similar to [6], [10], it differs from previous work in the following ways: it not only keeps the total satisfied demand and the number of placement changes the same, but also keeps the maximal machine utilization less than the given threshold for the worst case. That is, the system attempts to find another load distribution matrix L'' that satisfies the same demand for all dynamic clusters while achieving more balanced load across machines. We calculate L'' by solving the following optimization problem:

$$\text{Minimize: } \sum_{n\in N}\left|\sum_{m\in M} L''_{m,n} - \rho * \Omega_n\right| \tag{4}$$

Subject to:

$$\begin{aligned}
&\sum_{m\in M} L''_{m,n} \le p' * \Omega_n, \forall n \in N \\
&\sum_{n\in N} L''_{m,n} = \omega'_m, \forall m \in M \\
&I'_{m,n} = 0 \Rightarrow L''_{m,n} = 0, \forall m \in M, \forall n \in N
\end{aligned} \tag{5}$$

Here p' presents a constraint for the worst case machine utilization and w'_m is the satisfied demand of application m calculated during the first two phases (line 12-13 in Algorithm 1). The goal of the last phase is to reassign the load across machines, accordingly balancing the load and making all ρ_n as close to system load ρ as possible. We transform the problem in the last phase into a min-cost flow problem [21] as shown in Fig. 3. In this figure, from left to right, we can see

- Source node S has outbound edges to every application vertex m, where the capacity of the edge $S \rightarrow m$ is equal to load-dependent requirement ω'_m, and its cost is equal to 0.
- Each application vertex m has outbound edges to machine vertices n representing the machines that the application is placed on, conforming with $I_{m,n}$. The capacity of edge $m \rightarrow n$ is equal to ω'_m, and its cost is equal to 0.
- Each machine vertex n has an outbound edge to the ideal auxiliary machine vertices n' that corresponds to the same physical machine. The capacity of the edge $n \rightarrow n'$ is equal to the desired upper bound usage of the machine $\rho\Omega_n$, and its cost is equal to 0.
- The rebalancing vertex R has inbound edges from machine vertices $n \rightarrow R$, whose capacity is equal to $(p-\rho)\Omega_n$. The cost of these inbound vertices in 1.
- The rebalancing vertex R has outbound edges to ideal machine vertices $R \rightarrow n'$. Each such edge has the capacity $\rho\Omega_n$ and the cost of 1.
- All ideal machine vertices have an outbound edge to the sink node T, with capacity limit equal to $\rho\Omega_n$ and the cost of 0.

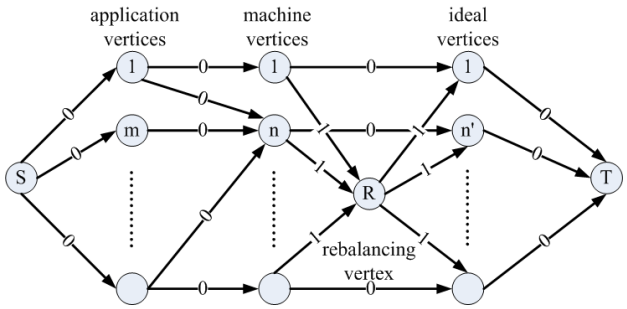


Fig. 3. Transformed Rebalancing Problem

The flow *max_demand* is injected into the network at source node S and leaves the network at sink node T. This network will push each ρ_n close to system load ρ because any deviation from ρ incurs a cost. This min-cost flow can also be solved by linear programming [20], as a linear function is needed to be optimized, and all constraints are linear.

D. Discussion

Next we turn to the time complexity of our placement algorithm. For the first phase of Initialization, the computational time is the same as the previous placement algorithm in [6], whose time time complexity is $O(|N|^{2.5})$. For the second phase of Iterative Optimizing, the computational time is the time of the previous placement algorithm multiplied by the number of iterations of the outer loop (lines 3-11 in Algorithm 1). Note that the outer loop is based on Binary Search: in each iteration, either p^+ or p^- are updated, and the search scope is halved, until the relative error is less than a given threshold, defined by ε. Since for an $\varepsilon-$approximation of the optimum value p^* the iterative search can be completed in $O(log_2(1/\varepsilon))$ rounds [20], the overall computational time of the second phase is $O(|N|^{2.5}log_2(1/\varepsilon))$. In our implementation, ε is set to be 0.01, that is, the outer loop is only performed a few rounds. As such, the time complexity of the second phase becomes $O(|N|^{2.5})$. For the last phase of Final Rebalancing, the time complexity is $O((|N|+|M|)^{2.5})$ [20]. Overall, the time complexity of our placement algorithm is $O(|N|^{2.5})$ when it is assumed that $|N|$ is comparable to $|M|$ value, which is similar to the time complexity of the previous algorithm in [6].

We assume that the placement controller produces a new placement solution that optimizes certain objective functions based on the current inputs such as γ_m and ω_m periodically every T miniatures (e.g., T=15 minutes). That is, during this period, it is assumed that the system with all running application instancex will not change significantly . Accordingly, the optimal worst case value p^* will not change significantly during this period as well.

IV. Performance Evaluation

This section evaluates the performance of our proposed framework. First, we describe the evaluation methodology and explain how experiments are set up. Then we present the load balancing example in Section IV-B as an illustration to our main objective. After that, we present simulation results of the

proposed framework under different scenarios. The cost and sensitivity analysis are given in Section IV-D and Section IV-E respectively.

A. Evaluation Methodology

Experimental Setup: Uniform application demand distribution is assumed because there is no significant difference between uniform and power-law distributions [6]. In addition, the demand of each application is normalized proportionally to the total application demand. The application demands change from cycle to cycle. We conduct experiments with two different demand changing patterns. With a *vary-all-apps* pattern, each application's demand changes randomly and independently within a ±20% range of its initial demand. With a *reset-all-apps* pattern, the demands in two consecutive cycles are independent of each other. This pattern represents the most extreme demand change for severe cases.

All experiments are configured as follows. Define CPU Load Factor L_{cpu} as the ratio between the total CPU demand and the total CPU capacity. That is, $L_{cpu} = \frac{\sum_{m \in M} \omega_m}{\sum_{n \in N} \Omega_n}$, where ω_m is the CPU demand for application m, and Ω_n is the CPU capacity of machine n. Also let Application Load Factor L_{mem} stand for $L_{mem} = \frac{|M|\overline{\gamma}}{|N|\overline{\Gamma}}$, where $\overline{\gamma}$ is the average memory requirement of applications, $\overline{\Gamma}$ denotes the average memory capacity of machines, and $|M|$ is the number of applications. Note that $0 \leq L_{mem} \leq 1$ dictates the number of applications instead of the real memory requirement; an application may need several instances to meet the demand, and the problem is most difficult when $L_{mem} = 1$. We uniformly distribute the configuration of machines over the set (1GB:1GHz, 2GB:1.6GHz, 3GB:2.4GHz, 4GB:3GHz), where the first number is memory capacity and the second is CPU power. The memory requirement of applications is uniformly distributed over the set (0.4GB, 0.8GB, 1.2GB, 1.6GB). Accordingly, the number of machines $|N|$ is set to be 100 in this paper. The number of applications $|M|$ is configured by $|M| = 2.5 * |N| * L_{mem}$ according to $|N|$ and L_{mem}. For example, $L_{mem} = 0.4$ leads to the same number of applications and machines, while $L_{mem} = 0.8$ results in twice as many applications. Higher values of L_{mem} correspond to more applications that need to be scheduled. Below, we concisely represent the system configuration of a placement problem as (L_{cpu}, L_{mem}, demand variability), *e.g.*, ($L_{cpu} = 0.9$, $L_{mem} = 0.4$, vary-all-apps). Among all the applications, we randomly pick 2% and paired them to simulate *dependent applications*.

Performance Metrics: *First*, we measure the maximal machine utilization p since it can result in adverse response time for users. *Second*, it is noted that the most balanced load is a uniform distribution; hence we measure the amount of inequality in the load distribution. Like [10], we consider the Gini index as an alternative metric. Assume the area between the line of perfect (uniform) distribution (45 degree line) and the Lorenz curve of the actual distribution is A and the area below the Lorenz curve of the actual distribution is B. The Gini index is thus referred to as $A/(A + B)$. This Gini coefficient is often used to measure income inequality [22]. A Gini index of 0 indicates perfect equality while a Gini index of 1 indicates complete inequality, or in our case, completely unbalanced load distribution. *Third*, when max demand should be strictly stratified, we present the number of placement changes, c, induced by both algorithms for comparison. *Fourth*, we compare the execution time of our algorithm with previous work [6]. All the reported data are averaged over the results on 100 randomly generated system configurations. For each configuration, the placement algorithm executes for 7 cycles under changing application demands, including an initial placement and 6 dynamic placement executions. Our results show that often the first execution is dominated by the initial placement(Cycle 1 $\rightarrow$ 2). Therefore for most experiments, we exclude this result and consider the five most recent executions. Unless otherwise stated, each reported data point is averaged over 500 placement results.

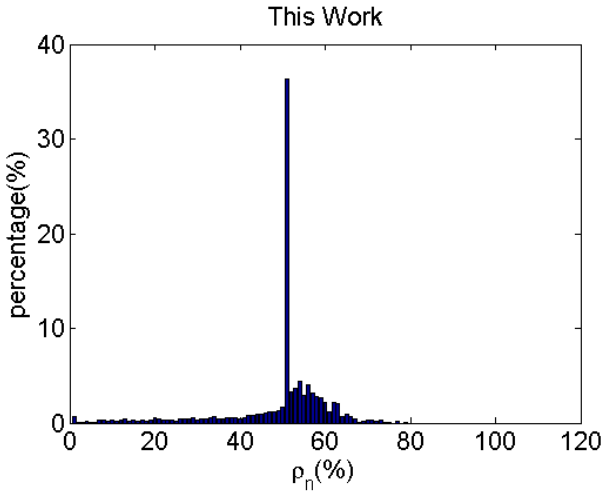


Fig. 5. Server Load Distribution of our work with configuration (L_{mem}= 0.6,L_{cpu}=0.5,reset-all-apps).

Peer Algorithms: As already demonstrated in [6], the application placement algorithm presented in that paper performs better than the two algorithms in [10], [11] with respect to maximizing demand, reducing placement changes, and execution time. Accordingly, we compare our proposed framework with the algorithm presented in [6]. For convenience of presentation, we use *Tang* to stand for the placement algorithm in [6], and *This* for our new placement algorithm.

B. Distribution of Server Utilization

In this illustration, we choose a typical setting where L_{mem} = 0.6 and L_{cpu} = 0.5. The result of Tang's work is already shown in Figure 1: the load balancing is not satisfactory, and some servers have 100% utilization when total system load is only 50%. As a comparison, our work shows much better load-balancing: shown in Figure 5, around 37% of the servers have utilization close to system load; only a few servers have utilization higher than 65%, and the worst case is no more than 80%. Consistent with our expectation, the worst case of individual server utilization is minimized, and load balancing in the whole system is greatly improved.

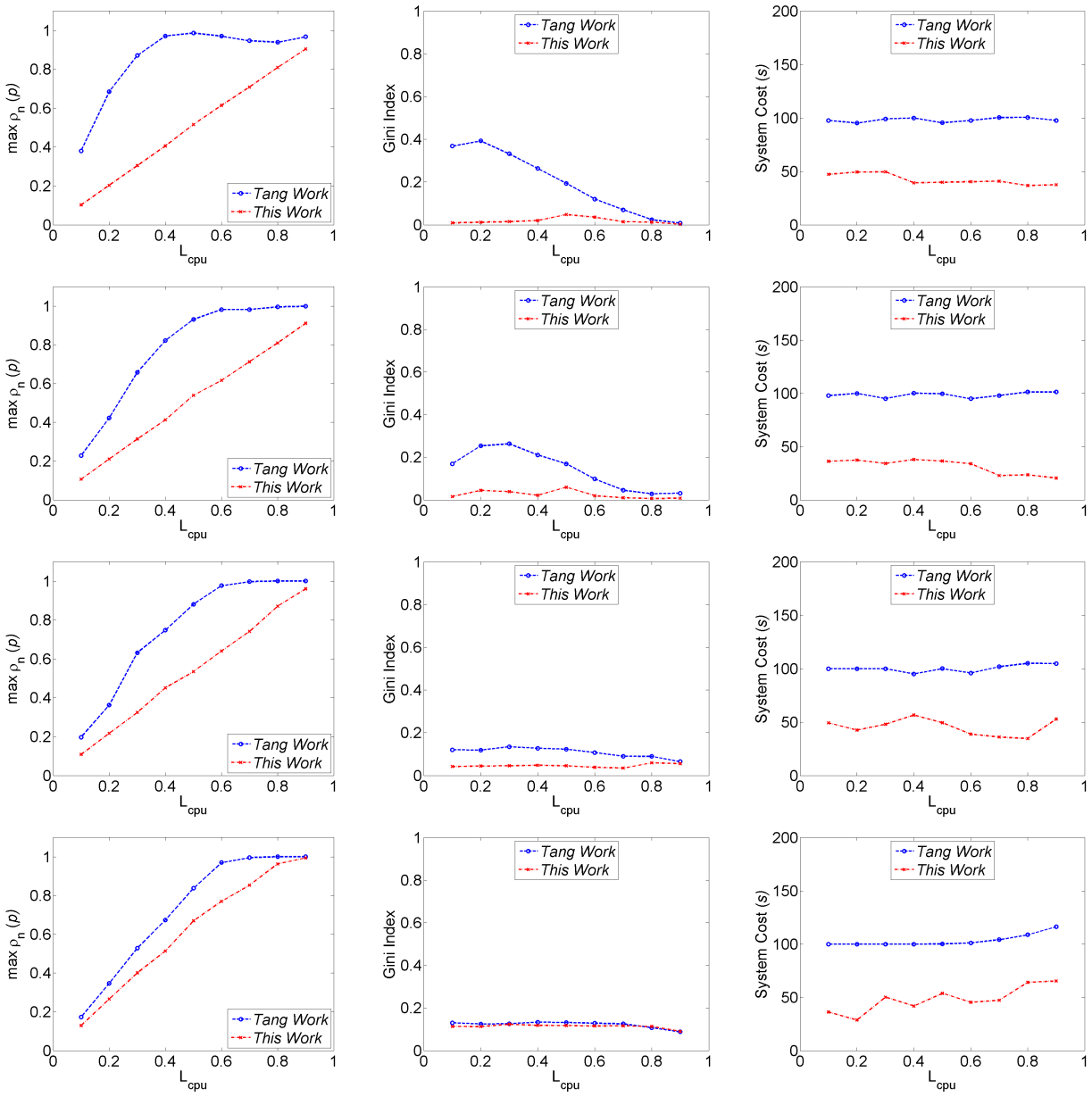


Fig. 4. Maximal machine utilization p value, Gini index, and placement changes with configuration (L_{mem}=0.2,0.4,0.6,0.8,L_{cpu}=x,vary-all-apps)

C. Performance Comparison

1) Vary-all-apps: Fig. 4 shows the experimental results with a variety of CPU load factors (L_{cpu} from 0.1 to 0.9) and application load factors (L_{mem}=0.2, 0.4, 0.6 and 0.8). For all these settings, the new framework proposed in this paper consistently outperforms the previous algorithm in terms of maximal machine utilization and Gini index.

First, Fig. 4 left column shows that the worst case machine utilization p using the *This* framework is less than that using *Tang*, especially in lightly load cases. For instance, when L_{mem} = 0.2 and L_{cpu} = 0.1, the p value using the *Tang* framework is around two times of that using the *This* algorithm, which is close to ρ. *Second*, compared with the *Tang* framework, the *This* framework greatly reduces the Gini index shown in Fig. 4 middle column. We found the Gini index is always less than 0.1. *Third*, the system costs using the *This* framework are stable as shown in Fig. 4 right column. The reason is that our new framework strives to reduce p and is thus capable of evenly distributing the applications. As a result, it will not cause considerable load fluctuations. *Fourth*, since the system costs of the new framework are limited by the previous algorithm *Tang*, it is guaranteed that *This* always introduces less cost than *Tang* while achieving better results in terms of p and Gini index. *Finally*, *This* achieves best results in terms of p, Gini index and cost s. However, there is no limitation on placement changes, so it could incur higher placement change, *i.e.*, when L_{mem} = 0.6, 0.8 in Fig. 4 right column; we will discuss this in the cost part.

2) Reset-all-apps: We next turn to the severe case when the demands in two consecutive cycles are independent of each other. In this case, we reset all applications every execution during all seven cycles. Fig. 6 depicts the results. *First*, most

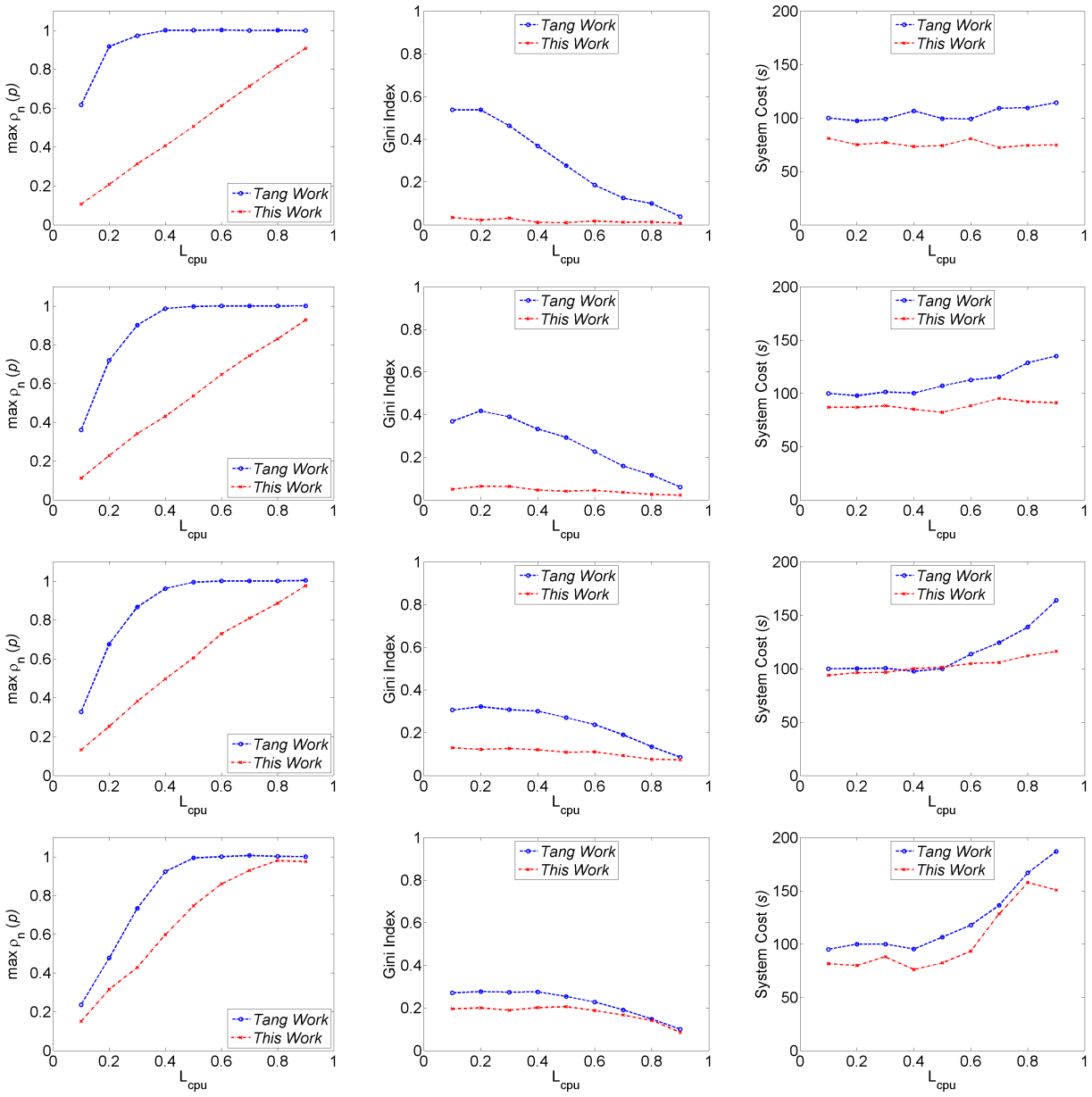


Fig. 6. Maximal machine utilization p value, Gini index, and placement changes with configuration (L_{mem}=0.2,0.4,0.6,0.8,L_{cpu}=x,reset-all-apps)

results are similar to those shown in Fig. 4. For example, the curves of p and the Gini index exhibit similar trends: p values shown in the left column of Fig. 6 are almost the same as those in the left column of Fig. 4; Gini index values shown in the middle column of Fig. 6 are slightly higher than those in the middle column of Fig. 4. *Second*, the *This* costs shown in the right column of Fig. 6 are doubled compared with those shown in Fig. 4(c). The reason is that in the *Reset−all−apps* scenario, there is no correlation of application demands between two consecutive times slots, which results in more cost to achieve a balanced placement if placement changes are not limited. Corresponding to its definition, the system cost of *This* is lower than *Tang*, while its load balancing performance is still comparable.

D. Cost Analysis

System cost is a single objective, as a combination of both placement change cost and inter-communication cost, in our formulation. In the previous part, we can conclude that our framework improves load balancing while maintaining system total cost at a low level. In this part, the impact to individual cost is examined. Execution time is also presented to verify the computational scalability.

In this illustration, we choose a typical setting where L_{mem} = 0.2. The localized communication h, placement change c and execution time are shown in Figure 7.

1) Inter Application Communication: The left column in Figure 7 shows the performance of localization of inter-application communications. The *Tang* algorithm does not consider this cost; hence its h value is randomly low. As a comparison, over 60% of inter-application communications are

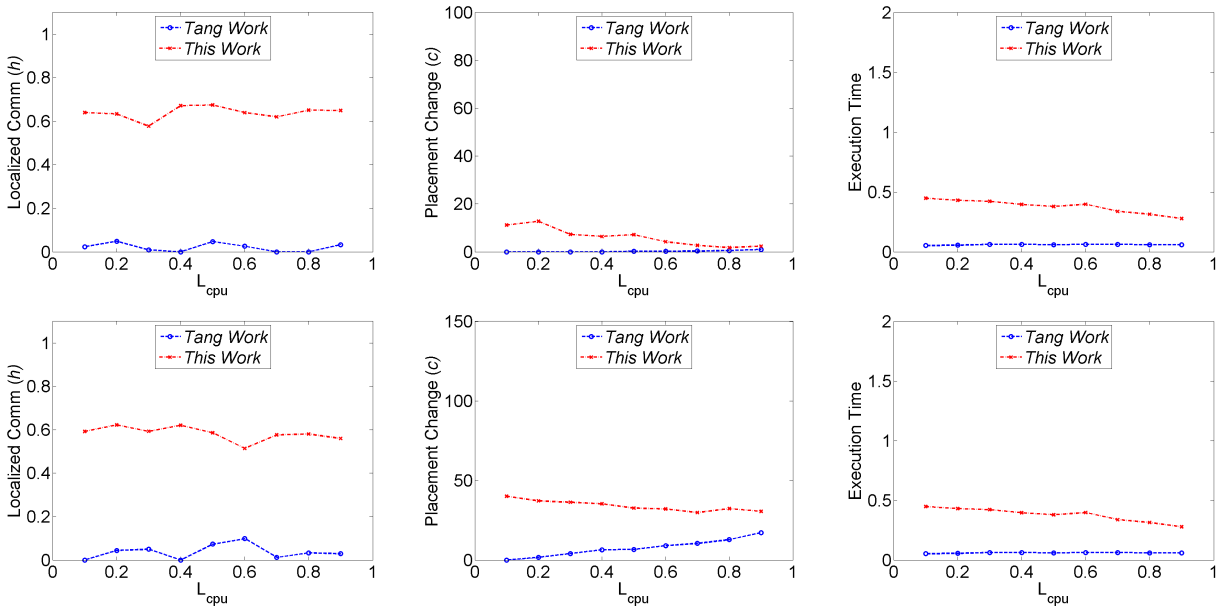


Fig. 7. Localized Communication, placement changes and execution time with configuration (L_{mem}=0.2,L_{cpu}=x, under $vary-all-apps$ (upper half) and $reset-all-apps$ (bottom half)

localized in $This$ algorithm.

2) Placement Change: The middle column in Figure 7 shows the performance of localization of inter-application communications. It is clear that our algorithm has higher placement change cost than $Tang$. In our earlier work, when inter-application communication cost is not considered, we can achieve comparable placement change cost and still optimize the load balancing performance. Hence we argue that higher placement change cost is not an algorithm defect, but a necessary cost for reducing the communication between nodes. In addition, users can modify α and β to adjust the weight of the two costs in the whole system cost.

3) Execution Time: The right column of Figure 7 depicts the relative execution time using the $This$ framework with both *vary-all-apps* both *reset-all-apps* pattern compared with the $Tang$ framework. *First*, as we mentioned in Section III, the execution time of our proposed framework is higher than that of the previous algorithm since the new one includes multiple runs to perform function **MaxDemandMinCost** compared with a single invocation for $Tang$. The execution time depends on the time required for Algorithm 1 to determine an optimal value of p^*. Results show that the ratio between execution time using the $This$ framework is $3 \sim 8$ times that of the $Tang$ framework. That is, the loop in Algorithm 1 only performs a few runs to find a good approximation.

While the proposed framework in this paper incurs overhead in terms of execution time compared with previous work [6], we believe that it provides a viable alternative solution for application placement and is practical for real systems. *First*, with optimizations that we are currently working on and ever-increasing CPU speed, the execution time could be lowered. *Second*, one option if execution time is a problem is to set a higher value for ϵ in Algorithm 1 and use a lighter weight version of the new algorithms in which the number of iterations of function **MaxDemandMinCost** is reduced. That way, one would get some of the benefits of the new algorithm without all of the overhead. *Third*, the placement controller may not always have to execute all that frequently (e.g., every 15 minutes in [6]). In this situation, the execution time using our framework would not be too high.

Other results show that our algorithm often converges after only a few iterations. That is, it does not introduce too much overhead when performing the optimization. The number of iterations will be small when resources are tight, that is, the values of L_{cpu} and L_{mem} are high. It is reasonable because when resources become tighter, the optimal p^* value is closer to 1; hence the probe space of p in Algorithm 1 is smaller.

E. Sensitivity Analysis

In this part, we analyze the sensitivity of both algorithms to system parameters. Although all default evaluation setting parameters are obtained from real world experience, it would be interesting to evaluate the algorithm in more versatile contexts, such as different system hardware context.

1) Fixed Memory v.s. Increasing CPU: In our normal experiments, the CPU/memory capacity is almost linear for each server type. In this experiment, we fix memory and vary the CPU power of each server type. The configuration of machines is now the set (2.5GB:1GHz, 2.5GB:1.6GHz, 2.5GB:2.4GHz, 2.5GB:3GHz). We choose a typical setting where L_{mem} = 0.6 with $reset-all-apps$ scenario. As shown in the first row of Figure 8, the performance is almost the same with the original experiments.

2) Fixed CPU v.s. Increasing Memory: In this experiment, we fix CPU power and vary the memory capacity of each server type. The configuration of machines is now the set (1GB:2GHz, 2GB:2GHz, 3GB:2GHz, 4GB:2GHz). We again choose a typical setting where L_{mem} = 0.6 with the $reset-all-apps$ scenario. As shown in the second row of

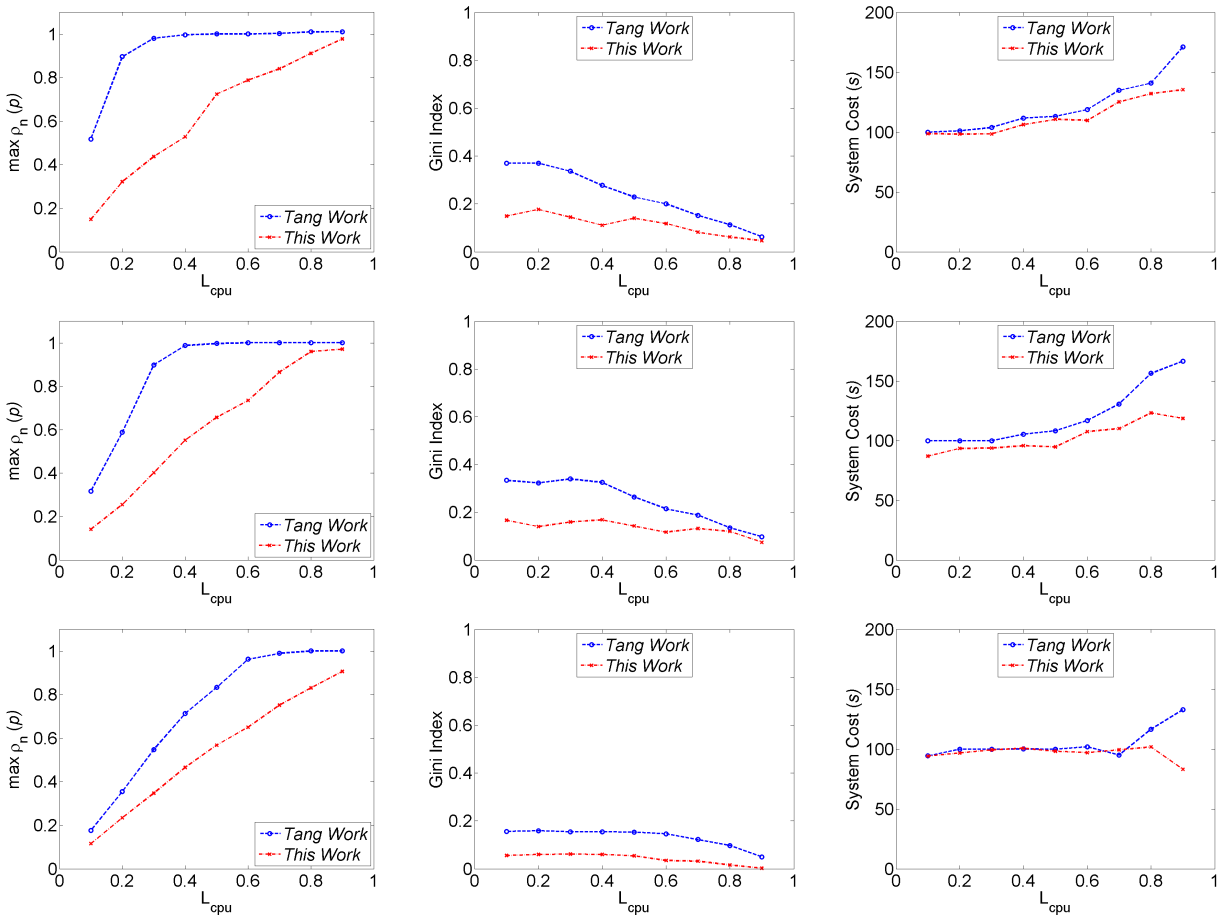


Fig. 8. Maximal machine utilization p value, Gini index, and placement changes with configuration (L_{mem}=0.6,L_{cpu}=x, under $reset - all - apps$

Figure 8, the performance is still almost the same compared with the original experiments.

3) Applications Requiring Less Memory: In this experiment, we use the original configuration of machines. However, we consider applications requring less memory. In these experiments, the memory requirements of applications are uniformly distributed over the set (0.08GB, 0.16GB, 0.24GB, 0.32GB), just 1/5 of the original experiment. Correspondingly, the number of applications in each experiment is 5 times compared with the original experiment. Intuitively, smaller application memory requirements would improve the performance of placement algorithms: it is easier to consolidate instances with smaller memory size.

As shown in the bottom row of Figure 8, the Gini index of both algorithms are significantly lower than the original results. Overall, our algorithm is still results in better load distribution.

V. Conclusion

We have presented a novel framework and a practical algorithm for application placement. This paper has two major contributions. First, motivated by the desire to minimize worst case server utilization and improve load balancing, our algorithm dynamically allocates resources to clustered web applications and balances load across servers simultaneously. Second, we look at communication cost among applications. We define system cost as the weighted combination of both placement change and inter-application communication cost. Our application placement framework attempts to maximize the number of instances of dependent applications that reside in the same set of servers while preserving high throughputs.

We have conducted extensive experiments and demonstrated that: 1) With the proposed framework, applications are evenly allocated among servers, throughput is high, and system cost is low; 2) our new algorithm is able to make application placement decisions in polynomial time. Overall, compared with state-of-the-art algorithms, our framework achieves better load balancing and controls system cost.

Several opportunities exist for future work. These include evaluating our algorithms on larger clusters to further test their scalability and adding a fail-over mechanism to ensure that the load balancer is not a single point of failure.Other topics for future work include: handling applications with different priorities and different quality-of-service requirements; handling more complex dependencies among applications in which more than two applications have communication dependencies among each other; and handling situations in which some applications can only can be allocated to certain types of servers with specific hardware or software.

ACKNOWLEDGEMENT

This work was supported in part through National Natural Science Foundation of China (No.60803115, No.60873127, No.61073147), the Fundamental Research Funds for the Central Universities (No.M2009022), the Youth Chenguang Project of Wuhan City (No.201050231080), the CHUTIAN Scholar Project of Hubei Province, and the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry.

REFERENCES

[1] D. Li, J. Wu, Y. Cui, and J. Liu. QoS-aware streaming in overlay multicast considering the selfishness in construction action. In *Proceedings of IEEE INFOCOM*, 2007.

[2] G. Tan and S.A. Jarvis. Improving the fault resilience of overlay multicast for media streaming. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):721–734.

[3] Alvin T. S. Chan, Jiannong Cao, and C. K. Chan. Webgop: collaborative web services based on graph-oriented programming. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(6):1874–1885.

[4] J. Liu, J. Xu, and X. Chu. Fine-grained scalable video caching for heterogeneous clients. *IEEE Transactions on Multimedia*, 8(5):1011–1020.

[5] K. Shen, H. Tang, T. Yang, and C. L. Integrated resource management for cluster-based internet services. In *Proceedings of OSDI*, 2002.

[6] C. Tang, M. Steinder, M. Spreitzer, and G. Pacici. A scalable application placement controller for enterprise data centers. In *Proceedings of International World Wide Web Conference (WWW)*, 2007.

[7] H. Shachnai and T. Tamir. On two class-constrained versions of the multiple knapsack problem. *Algorithmica*, 29(3):442–467, 2001.

[8] H. Shachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. *Journal of Scheduling*, 4(6):313–338, 2001.

[9] S. Zhou. A trace-driven simulation study of dynamic load balancing. Technical Report UCB/CSD87/305, Univ. California, Berkeley, 1986.

[10] A. Karve, T. Kimbrel, G. Pacici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic application placement for clustered web applications. In *Proceedings of International World Wide Web Conference (WWW)*, 2006.

[11] T. Kimbrel, M. Steinder, M. Sviridenko, and A. N. Tantawi. Dynamic application placement under service and memory constraints. In *Proceedings of International Workshop on Ecient and Experimental Algorithms*, 2005.

[12] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceanosla based management of a computing utility. In *Proceedings of International Symposium on Integrated Management*, 2001.

[13] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of OSDI*, 2002.

[14] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer-CVerlag, 2004.

[15] A. Turgeon, Q. Snell, and M. Clement. Application placement using performance surfaces. In *Proceedings of International Symposium on High Performance Distributed Computing (HPDC)*, 2000.

[16] J. L. Wolf, P. S. Yu, and H. Shachnai. Disk load balancing for video on-demand systems. *ACM Multimedia Systems*, 5(6):358–370, 1997.

[17] D. N. Serpanos, L. Georgiadis, and T. Bouloutas. MMPacking: A load and storage balancing algorithm for distributed multimedia servers. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(1):25–30, 1998.

[18] Xueyan Tang and Jianliang Xu. Qos-aware replica placement for content distribution. *IEEE Trans. Parallel Distributed Systems*, 16:2005, 2005.

[19] Hsiangkai Wang, Pangfeng Liu, and Jan jan Wu. A qoS-aware heuristic algorithm for replica placement. In *Proceedings of IEEE/ACM International Conference on Grid Computing*, 2006.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (2nd edition ed.)*. MIT Press and McGraw-Hill, 1990.

[21] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, New Jersey, 1993.

[22] Gini index. http://en.wikipedia.org/wiki/Gini_coefficient.

Chen Tian received the BS, MS and PhD degrees from the Department of Electronics and Information Engineering at the Huazhong University of Science and Technology, China, in 2000, 2003, and 2008 respectively. He joined the faculty as a lecture in the Department of Electronics and Information Engineering at the Huazhong University of Science and Technology, China. His research interests include distributed networks, wireless networks and network architecture.

Hongbo Jiang received the B.S. and M.S. degrees from Huazhong University of Science and Technology, China. He received his Ph.D. from Case Western Reserve University in 2008. After that he joined the faculty of Huazhong University of Science and Technology as an associate professor. His research concerns computer networking, especially algorithms and architectures for high-performance networks and wireless networks. He is a member of the IEEE.

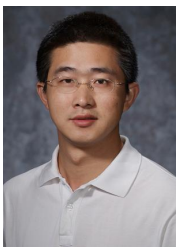
Arun Iyengar received the Ph.D. degree in computer science from MIT. He does research and development into Web performance, distributed computing, and high availability at IBMs T.J. Watson Research Center. Arun is Co-Editor-in-Chief of the ACM Transactions on the Web, Founding Chair of IFIP Working Group 6.4 on Internet Applications Engineering, and an IBM Master Inventor. He is a senior member of the IEEE.

Xue Liu received the BS degree in applied mathematics and the MEng degree in control theory and applications from Tsinghua University and the PhD degree in computer science from the University of Illinois, Urbana-Champaign, in 2006. From 2007 to 2009, he was an Assistant Professor in the School of Computer Science at McGill University in Montreal, Canada. He is currently an associate professor in the Department of Computer Science and Engineering, University of Nebraska-Lincoln. He was briefly with the Hewlett-Packard Laboratories and IBM T.J. Watson Research Center. His research interests include real-time and embedded computing, performance and power management of server systems, sensor networks, fault tolerance, and control. He is the author/coauthor of more than 20 refereed publications in leading conferences and journals in these fields. He is a member of the IEEE.

Zuodong Wu received the BS from the Huazhong University of Science and Technology, China, in 2010. He is for now a M.S. students in the Department of Electronics and Information Engineering at the Huazhong University of Science and Technology, China. His research interest is network architecture.

Jinhua Chen received the BS from the Huazhong University of Science and Technology, China, in 2008. He is for now a M.S. students in the Department of Electronics and Information Engineering at the Huazhong University of Science and Technology, China. His research interest is network architecture.

Wenyu Liu received the PhD and MS degrees from the Department of Electronics and Information Engineering at the Huazhong University of Science and Technology, China. received the BS degree from the Department of Computer Science and Technology at the Tsinghua University, China, in 1986. He is a professor of electronics and information engineering at the Huazhong University of Science and Technology, China. His research interests include image processing, distributed networks and wireless networks. He is a member of the IEEE.



Chonggang Wang received his PhD degree from Beijing University of Posts and Telecommunications (BUPT). He was awarded a National Award for Science and Technology Progress in Telecommunications. He is currently with NEC Laboratories America. His research focuses on Hybrid Optical and Wireless Networks, Sensor Networks and Applications, Cognitive Radio Networks, Ubiquitous and Distributed Computing, and Data Center. He is an editor of ACM/Springer Journal of Wireless Networks and an associate technical editor of IEEE Communications Magazine. He is a senior member of the IEEE.