2015 IEEE International Conference on Big Data (Big Data)

Toward Locality-aware Scheduling for Containerized Cloud Services

Dongfang Zhao, Nagapramod Mandagere, Gabriel Alatorre, Mohamed Mohamed, Heiko Ludwig

Cloud Management Services Department
IBM Almaden Research Center
San Jose, CA 95120, United States

{dzhao, pramod, galatorr, mmohamed, hludwig}@us.ibm.com

***Abstract*—The state-of-the-art scheduler of containerized cloud services considers load-balance as the only criterion and neglects many others such as application performance. In the era of Big Data, however, applications have evolved to be highly data-intensive thus perform poorly in existing systems. This particularly holds for Platform-as-a-Service environments that encourage an application model of stateless application instances in containers reading and writing data to services storing states, e.g., key-value stores. To this end, this work strives to improve today's cloud services by incorporating sensitivity to both load-balance and application performance. We built and analyzed theoretical models that respect both dimensions, and unlike prior studies, our model abstracts the dilemma between load-balance and application performance into an optimization problem and employs a statistical method to meet the discrepant requirements. Using heuristic algorithms and approaches we try to solve the abstracted problems. We implemented the proposed approach in Diego (an open-source cloud service scheduler) and demonstrate that it can significantly boost the performance of containerized applications while preserving a relatively high load-balance.**

***Index Terms*—cloud computing; service computing; containers; load-balance; data locality; optimization**

I. Introduction

A core ingredient in cloud computing is virtualization (for example, Xen [19], Red Hat KVM [15], Microsoft Hyper-V [12], EMC/VMware vSphere [7], IBM z/VM [9]) that enables high resource utilization as well as the isolation of applications. Conventionally, the virtualization is achieved through a hypervisor that manages multiple virtual machines (VM) on the same physical machine. Nevertheless, VMs require a complete stack of software, from the operating system (OS) to application libraries, which takes a significant amount of time to set up (tens of seconds when automated). Recently, lightweight isolation (for example, Docker container [6]) has emerged and been well received since it achieves virtualization without booting a full VM; thanks to the user-level engine, a container usually takes only a couple of seconds to start in its own logical space.

This lightweight, finer-granularity isolation (i.e., containers), however, introduces new challenges to scheduling. The conventional VM-level scheduler considers VMs as standardized and independent units that are maintained by the lower-layer hypervisor. Nevertheless, containers are more proximate to each other in the sense that the controller runs at the same level (all in user space), which means the isolation between containers is not as strong as VMs. Moreover, the state-of-the-art scheduler strives to preserve load-balance and is completely agnostic about the physical layout of the containers, which might result in serious resource contention within and between containers (for example, local disk I/O bandwidth, network bandwidth).

An additional challenge for Big Data applications is the use of a 12-factor application model that underlies some Platform-as-a-Service (PaaS) systems such as Cloud Foundry [3] that typically uses application build packs placed within containers. Instances of an application are deployed in containers but are considered stateless. Container-local file system content is not persisted beyond the life cycle of an individual application instance. For persistence, application instances are bound to persistence services, which can be deployed within or outside the platform. They access data according to the model of the persistence service, e.g. as object, key-value, relational database and so forth. From a scheduler's perspective, this introduces the additional need to consider the relationships of containers to each other and outside data services, which may be crucial for application performance. An application instance should read large amounts of data from services in close network proximity to its own location.

With all this in mind, the following quesiton arises, "How can we make the scheduler of containerized cloud services/Platform-as-a-Service more locality-aware?"

This paper makes the following contributions.

- We identify limitations in the state-of-the-art design of PaaS. We pinpoint that a round-robin policy of containerized cloud services incurs significant performance overhead from both local disk I/O and network traffic.
- We abstract the minimization of performance deficiency into optimization problems and analyze their complexities.
- We implement the proposed locality-aware scheduler by extending Diego — the next-generation scheduler of the open-source PaaS Cloud Foundry [3].

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

- We verify the correctness of the locality-aware scheduler and evaluate its effectiveness. Experiments show up to 60% reduction in network traffic while maintaining load-balance.

The remainder of this paper first (Section II) gives a brief background of containerized Platform-as-a-Service (PaaS) and articulates the unique challenges. We formulate the problem under different scenarios, show that some of them are NP-hard, and devise and analyze heuristic algorithms in Section III. In order to justify the effectiveness of proposed algorithms and optimizations, we implement them in the open-source PaaS framework Diego (Section IV) and report the experimental results (Section V). We review some closely related work of job scheduling in Section VI and finally conclude this paper in Section VII.

II. Background and Motivation

As a motivating example, Figure 1 shows a typical scenario where several deployed applications depend on certain services. In the state-of-the-art PaaS scheduler (for example, Diego [5]) these dependencies are not considered in the decision-making process; by default, a naive round-robin strategy (focused on load-balance) is used for deploying applications. As a result, the network traffic incurred by the placement decision could be large enough to jeopardize an application's performance.

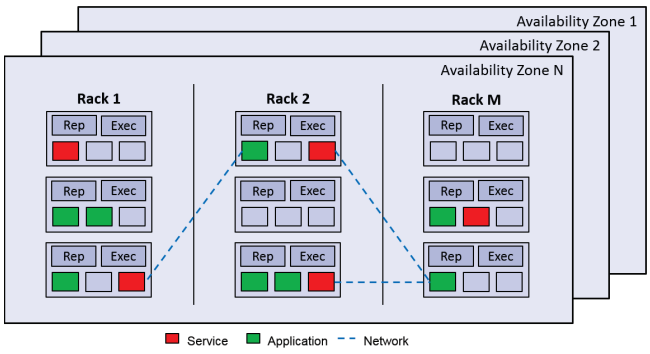

Fig. 1. Application-service affinity placement

In addition to the network overhead, another source of degradation comes from the intra-node resource contention. More specifically, physical resources shared between multiple containers on the same node could be the performance bottleneck. Note that "shared resource" is not necessarily circumvented by parallelization or partitions. For instance, a conventional hard-disk drive (HDD) could serve as multiple virtual block devices for cells or dockers. The partitions of HDD meet the requirement of disjoint spaces for different applications, yet fail to isolate the resources from a performance point of view. Since only a single head exists for disk seek and data access, concurrent accesses to the same HDD cause seriously degraded I/O throughput and increased latency.

As a concrete example for illustrating the resource contention in a containerized cloud service, we show three scenarios of different workload types deployed on the same cell: compute-intensive workloads only, I/O-intensive workloads only, and a mix of both compute- and I/O-intensive workloads. In this oversimplified example, we allocate four containers in a VM. The compute-intensive application is a web application written in Ruby to recursively calculate the Fibonacci series to 40. The data-intensive application is a simple web portal that triggers a writing of 2 GB of data to disk.

Figure 2 shows that a full-scale (i.e., 4 applications) of compute-intensive applications adds about 9% overhead when compared to the baseline. This experiment makes a strong case that the isolation of CPU resources is excellent in the context of containerized applications. After all, multiple cores within the same chip are relatively independent thus attribute small interaction overhead between peers.

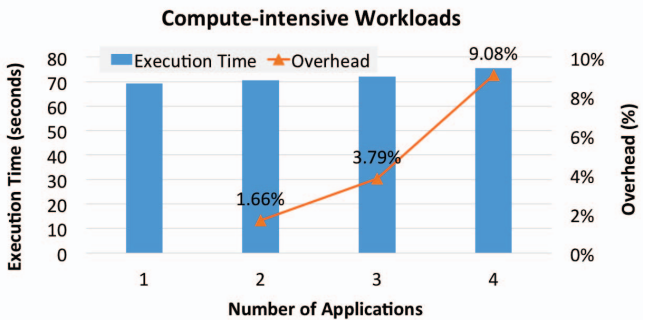

Fig. 2. Multiple compute-intensive applications on the same cell have relatively low contention

Figure 3 shows that two data-intensive applications compete for the I/O bandwidth of the underlying disk and finish in about double the baseline time. The root cause of this phenomena is that the disk I/O is not truly parallelized, although the space is exclusively partitioned for multiple containers. That is, the single head of the underlying disk is busy with the concurrent I/O request from multiple containers thus becoming the performance bottleneck (and single point of failure).

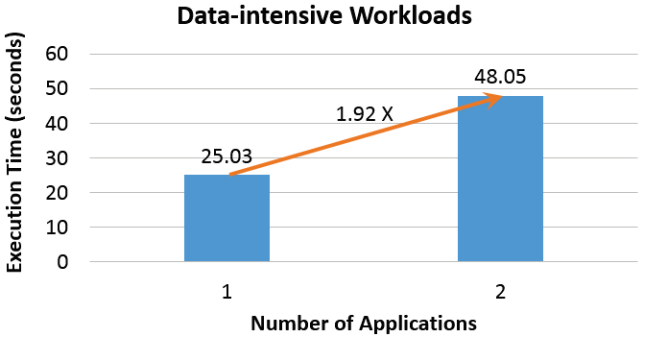

Fig. 3. Multiple data-intensive applications on the same cell cause serious resource contention and degraded performance

Figure 4 shows that a data-intensive and a compute-intensive application minimally intervene each other when both are deployed on the same cell. In other words, the execution time of both applications are completed almost in the same time duration as their baseline cases. These results confirm our

conjecture: the contention between distinct types of resources inside the same cell (could be either a physical node or a VM) is light.

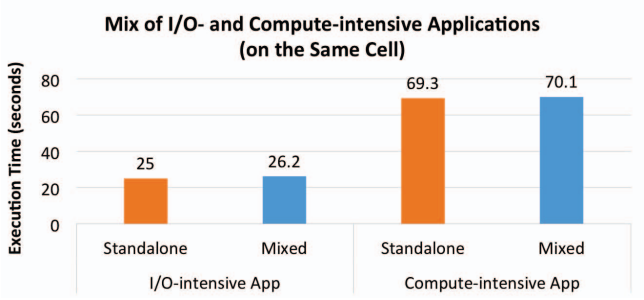

Fig. 4. Compute- and data-intensive applications deployed on the same cell do not cause noticeable contention

III. Locality-aware Application Placement

Most Platform-as-a-Service (PaaS) frameworks typically abstract client applications and service provider applications as jobs for the purposes of planning and management. On the infrastructure side, PaaS abstracts infrastructure components into cells, with each cell characterized by virtual compute capacity (vcpus), virtual memory capacity (vmem) and virtual disk capacity (vdisk). Further these cells can be grouped into higher level aggregates such as zones for purposes of segregating workloads and availability requirements. In this section, we start with our formulation of the placement problem specifically focusing on enabling locality awareness, followed by incorporating both objectives of locality awareness and uniform load balancing in a coordinated fashion.

The list of available cells is denoted by C, where C_j represents the j^{th} cell. The running applications' indexes on a particular cell are represented by $C_j.apps$. Similarly, $Z_j.apps$ indicates the list of applications running on Z_j. Available resources of C_j can be retrieved by $C_j.cpu$, $C_j.ram$, and $C_j.disk$. The total I/O bandwidth of C_j is denoted as $C_j.io$.

The list of applications is denoted by A, where A_i represents the i^{th} application. The cell index where an application has been deployed is represented by $cell(A_i)$. The requested resources (for example, CPUs, memory, disk space) of A_i are represented by $A_i.cpu$, $A_i.ram$, and $A_i.disk$. We denote the required disk throughput of A_i as $A_i.io$. If $A_i.io$ is not explicitly specified, we assume A_i will aggressively take all disk bandwidth of C_j.

The definition for cells can be generalized to other entities as well. For example, if we are interested in the metrics at the zone level, then the above notations are applicable as well by replacing C to Z. Similarly, the zone where an application A_i resides is denoted by $zone(A_i)$, and the available resources of Z_j is denoted by $Z_j.cpu$, $Z_j.ram$, and $Z_j.disk$.

The dependency matrix D specifies the interaction between applications. For example, $(D_{i,j} == 1)$ means application A_i has non-trivial I/O to application A_j. Another matrix T stores the pair-wise traffic of data movement among all location units (for example, cells, physical nodes, zones). The numerical value in $T_{i,j}$ indicates the closeness between C_i and C_j. For instance, the cost of transferring data between two cells on the same physical machine is obviously lower than those on two different racks, which is then lower than those on two different locations (i.e., zones). In practice, the closeness could be network latency, if users have many small I/Os, or network bandwidth, if large volume of data is expected.

One important assumption of some of the following discussions is that containerized applications and services can be migrated to arbitrary locations. This is a required property for the global optimization where deployed applications need to be adjusted. However, how to migrate and how much overhead is involved with the migration is beyond the scope of this paper. In practice, both offline and online (live) migrations are available for VMs and containers (for example, Jelastic [10]).

A. Locality Awareness - Reducing Network Contention

There are various granularities of network contention such as cell-to-cell, machine-to-machine, and zone-to-zone. To make the following analysis neat and clear, we assume zone-to-zone is of the only interest. The idea and validity, however, holds true for other scenarios as well (for example, cell-to-cell, machine-to-machine). We do not differentiate an *application* and a *service* to simplify the following analysis; after all, both are just jobs from the system's point of view. Therefore, the dependency matrix D is also applicable here where $(D_{i,k} == 1)$ means that an application A_i is dependent on service/application A_k. The problem for an application to achieve the minimal network traffic then becomes this: given A_i we find Z_j $(1 \leq j \leq |Z|)$ such that

$$\arg\min_{j} \sum_{k} D_{i,k} \times T_{j,zone(A_k)} \tag{1}$$

subject to

$$Z_j.cpu > A_i.cpu$$

$$Z_j.ram > A_i.ram$$

$$Z_j.disk > A_i.disk$$

It should be clear that Equation 1 only optimizes the network traffic of a single new application. This problem obviously has a polynomial solution. We can iterate all the available zones $(O(|Z|))$, and in each iteration we calculate the aggregate cost between A_i and its dependent services $(O(|A|))$. The minimal cost is also updated during each iteration. Therefore the overall cost takes $O(|Z| \cdot |A|)$.

B. Locality Awareness with Uniform Load Balancing

Admittedly, in the previous Section III-A we only consider network traffic and completely neglect load-balance. The following discussion focuses on how to tune both factors —load-balance and network traffic— in an orchestral manner. That is, we have one more constraint on load-balance when optimizing network traffic.

In the simplest scenario, the coefficient of variance (CV) between the loads on different zones should be under a user-defined threshold (UDT). Formally,

$$\arg\min_{j} \sum_{k} D_{i,k} \times T_{j,zone(A_k)} \tag{2}$$

subject to

$$\texttt{CV}(Z) \leq \texttt{UDT}$$
$$Z_j.cpu > A_i.cpu$$
$$Z_j.ram > A_i.ram$$
$$Z_j.disk > A_i.disk$$

Equation 2 does not add much complexity to the original criterion Equation 1. In essence, in each iteration when looping on $|Z|$ we need to recalculate CV and drop the unqualified candidates. Thus the process takes $O\left(|Z| \cdot (|Z| + |A|)\right)$ in total. However, an obvious limitation exists in Equation 2: what if no qualified zone is available with the required CV?

We propose to normalize both load-balance and network-traffic criteria and apply a weight factor to each. The normalization of network traffic is measured by the ratio of the application's actual traffic divided by the aggregate bandwidth. Ideally, if all applications are executed on local nodes only, the normalized network traffic is simply zero. Formally, the normalization of network traffic (`tr`) for A_i on Z_j (i.e., a mapping matrix M where $M_{i,j}$ indicates A_i is deployed on Z_j) is

$$\texttt{tr}(M_{i,j}) = \frac{\sum_{k=1}^{i-1} D_{i,k} \times T_{j,zone(A_k)}}{\sum_{m=1}^{|Z|} \sum_{n=1}^{|Z|} T_{m,n}} \tag{3}$$

Obviously, we have

$$\texttt{tr}(M_{i,j}) \in [0 \cdots 1], \forall i, j$$

by observing that the total throughput of applications cannot exceed the overall bandwidth.

The normalization of load-balance is defined as the adjusted coefficient of variance of the overall deployed applications on each zone. That is

$$\texttt{cv}(M_{i,j}) = \frac{\sqrt{(1 + |Z_j.apps| - \overline{Z})^2 + \sum_{k \neq j} (|Z_k.apps| - \overline{Z})^2}}{\sqrt{|Z|} \cdot \overline{Z}} \tag{4}$$

where

$$\overline{Z} = \frac{1 + \sum_{j=1}^{|Z|} |Z_j.apps|}{|Z|} \tag{5}$$

We are now ready to define the objective function with both factors. Let α and β denote the weights for the normalizations of `traffic()` and `cv()`, respectively. Our goal is then to find Z_j for A_i (i.e., $M_{i,j}$) such that

$$\arg\min_{M} F(M) \tag{6}$$

where the scoring function is

$$F(M) = \frac{\alpha \cdot \texttt{tr}(M) + \beta \cdot \texttt{cv}(M)}{\alpha + \beta}$$

subject to

$$Z_j.cpu > A_i.cpu$$
$$Z_j.ram > A_i.ram$$
$$Z_j.disk > A_i.disk$$

The complexity analysis of Equation 6 is as follows. Equation 5 can be efficiently updated in constant time assuming zones are not changed. If we maintain the difference square for each zone in memory, Equation 4 can be also updated in constant time. Nevertheless, in general it takes $O(|Z|)$ to update Equation 4. In Equation 3 every dependency is checked that takes $O(|A|)$. Therefore, the overall time is $O(|Z| \cdot (|Z| + |A|))$. That is, although taking both network traffic and load-balance into consideration looks more complicated than the simple threshold-check approach (i.e., Equation 2), it essentially takes time in the same order of magnitude to finish.

The problem becomes significantly harder (even without considering load-balance) with a subtle change when we consider scheduling a batch of applications or all the applications globally. In essence, instead of the linear problem on a single dimension of $|Z|$, we are now faced with a combinatorial problem of placing $|A|$ applications on $|Z|$ zones such that:

$$\arg\min_{M} \sum_{i} \sum_{k} D_{i,k} \times T_{j,zone(A_k)} \tag{7}$$

subject to

$$Z_j.cpu > A_i.cpu$$
$$Z_j.ram > A_i.ram$$
$$Z_j.disk > A_i.disk$$

A brute-force solution would obviously take exponential time, $O(|Z|^{|A|})$. This paper is focused on the optimization problem with regard to a single application. Therefore the algorithm, analysis, and evaluation are all concentrated on how to minimize the contention when a new application is deployed. We leave the optimization of batches of applications as an open question to the community as well as one of our future works.

IV. Implementation

We have implemented the hybrid model discussed in Section III. We modified the Diego source code to allow users to choose whether locality should be considered in the scheduling decision. In particular, a new module called `locality.go` is created among other changes to the existing modules such as `auctionrunner.go` and `scheduler.go`.

A simplified diagram of key components in the proposed locality-aware scheduler is shown in Figure 5. Two major packages are involved: `auctionrunner` and `models`.

- The `auctionrunner` package is responsible for all the auction-related scheduling where most changes were made to the vanilla Diego. It takes the deployment request

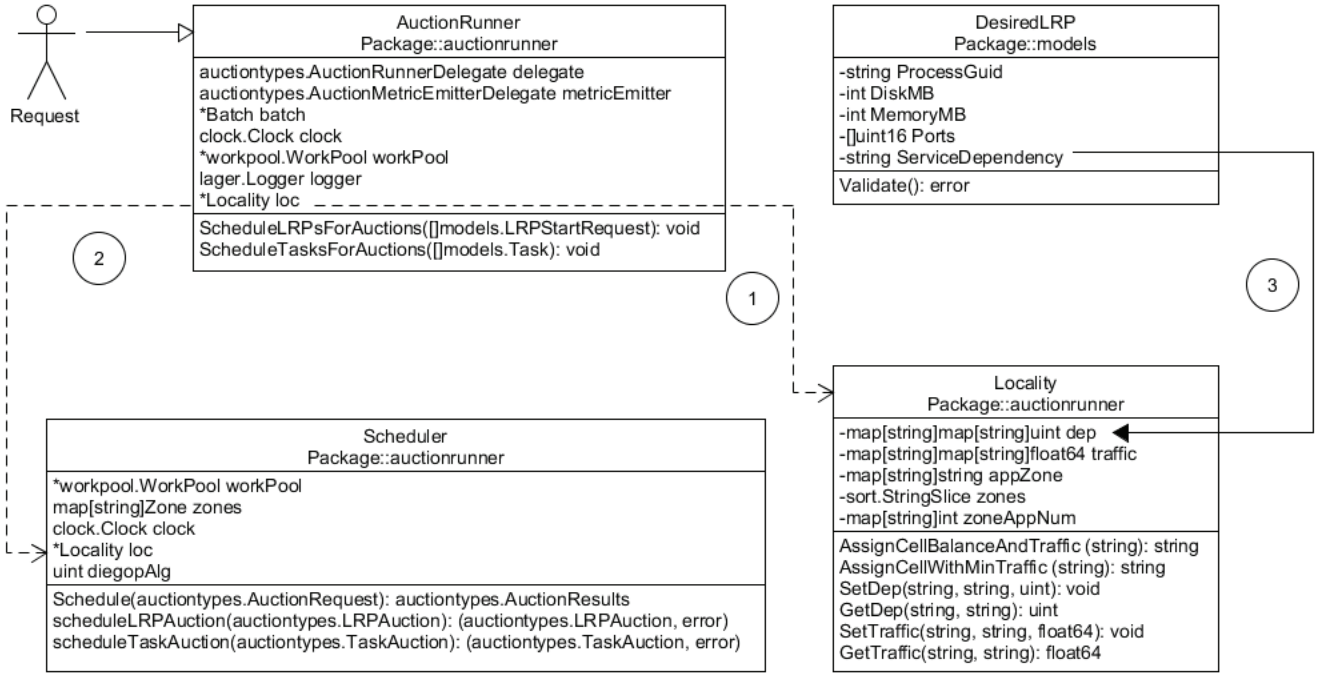

Fig. 5. Introduction of the `Locality` class

and initializes a `Locality` object (step ①), whose reference is then passed to the `Scheduler` class (step ②). The scheduler deploys the application to the cell according to the placement algorithm specified in the `Locality` class.

- The `models` package maintains all the abstraction models in Diego, which was extended in the `DesiredLRP` to also mange the application dependencies. That is, the `Locality` class retrieves the application dependency from the `DesiredLRP` (step ③) that is originally specified in the manifest file by the users.

The `Locality` class[1] is defined in Figure 6. The `dep` and `traffic` fields represent the dependency and traffic matrices discussed in Section III. The `appZone` field is a hashmap from an existing application to its assigned zone. The `zones` field is a sortable array of all the available zones. The `zoneAppNum` field records the number of running applications on a particular zone.

A `Locality` reference is instantiated and initialized in the main scheduling entity named `auctionrunner`. That is, only one instance of `Locality` (for example, the dependency matrix, the traffic matrix, and so forth) is maintained, which is desirable. The `Locality` reference is then passed to the `scheduler` object that could call either the vanilla placement algorithm or the method named `AssignCellWithMinTraffic(app string)` exposed by `Locality`.

[1]Technically, it is a `struct` in the `Go` programing language.

```
type Locality struct {
  dep map[string]map[string]uint
  traffic map[string]map[string]float64
  appZone map[string]string
  zones sort.StringSlice
  zoneAppNum map[string]int
}
```

Fig. 6. Definition of the Locality class

V. Evaluation

We first verified the correctness of the proposed scheduling algorithms leveraging the built-in simulation mode in Diego. Experimental setup includes a set of 8 different application workloads. The simulation module in Diego takes a user specified set of workloads/applications and layout of infrastructure (i.e., cells, zones, etc.) and triggers the Diego scheduler to make placement decisions. This simulation mode provides a light weight mechanism to quickly integrate different types of scheduling mechanisms, mainly with the goal of enabling quick design space exploration.

Using a simplified infrastructure setup, we try to illustrate behavior of both default load balancing scheduler and our proposed locality-aware scheduler. Specifically, we assume two cells (`REP-1` and `REP-2`) exist on two zones (`Z0` and `Z1`), respectively. Each cell has a capacity of eight containers.

Eight applications are to be deployed, namely A0 to A7. The vanilla placement algorithm in Diego, unsurprisingly, assigns four applications on each zone for the sake of load-balance as the simulator output shows in Figure 7 where "+" indicates a deployed application and "." represents a free container.

```
Distribution
  [Z0] REP-1: ++++....
  [Z1] REP-2: ++++....
```

Fig. 7. Distribution of eight applications on two zones with Diego

Now let us consider the same set of applications being deployed with the locality-aware scheduler. Applications A0 to A7 are deployed to our 2-zone cloud in a serial manner where each application is dependent on other applications specified as 1 in Figure 8. For instance, A0 and A1 do not have any dependent applications (or, services), while A2 is dependent on A1 and A3 is dependent on A0. For $i := 4..7$, A(i) is dependent on A(i-1).

Dependency	A0	A1	A2	A3	A4	A5	A6	A7
A0	0	0	0	0	0	0	0	0
A1	0	0	0	0	0	0	0	0
A2	0	1	0	0	0	0	0	0
A3	1	0	0	0	0	0	0	0
A4	0	0	0	1	0	0	0	0
A5	0	0	0	0	1	0	0	0
A6	0	0	0	0	0	1	0	0
A7	0	0	0	0	0	0	1	0

Fig. 8. Dependency matrix of applications

The traffic matrix between zones is shown in Figure 9. Here we assume the network traffic across different zones is symmetric, which is why the matrix is also symmetric. The unit of the numbers does not matter as long as they accurately reflect the relative traffic. In practice, "traffic" could be end-to-end latency for small file accesses, network throughput for data-intensive workloads, and so forth. Because of the huge performance difference between local and remote traffic, we should try to assign dependent applications to the same zone, if possible, to optimize for performance. If either zone yields the same traffic, then we pick the one with fewer running applications (from the zoneAppNum field in Figure 6) for the sake of load-balance.

Traffic	Z0	Z1
Z0	10	100
Z1	100	10

Fig. 9. Traffic matrix of zones

In this example, the traffic across two zones is 10 times larger than the local one. We chose 10 by observing that the inter-zone latency is roughly 10 times larger than intra-zone on Amazon EC2: the latency within North California is 34 ms while the latency between North California and Singapore 385 ms [4]. In the following discussion, we use 10 and 100 for clear presentation instead of the raw measurements (i.e., 34 and 385).

We are now ready to understand why the locality-aware scheduler places six applications in Z0 and two others to Z1, as shown in Figure 10. Since neither A0 and A1 has any dependency, A0 is assigned to Z0 and A1 is assigned to Z1. For A2, because it is dependent on A1 it is deployed to Z1 as well. Yet, because A3 is dependent on A0, it is deployed to Z0. Similarly, A4 $\cdots$ A7 are deployed to Z0 because their dependencies are all on Z0. Therefore, we observe six applications are on Z0: A0, A3 $\cdots$ A7; two applications are on Z1: A1 and A2.

```
Distribution
  [Z0] REP-1: ++++++..
  [Z1] REP-2: ++......
```

Fig. 10. Distribution of the same eight applications on two zones

The overall traffic of applications scheduled by Diego is as follows. Recall that according to the strict load-balance rule of Diego, all even-numbered applications are deployed to Z0 and all odd-numbered applications are deployed to Z1. Unfortunately, A2 $\cdots$ A7 all have dependencies (refer to Figure 8) residing on the a zone different than its own. Therefore, the overall traffic is $100 \times 6 = 600$. On the other hand, in the locality-aware scheduler, all six applications have their dependencies in the same zone, resulting in the overall traffic $6 \times 10 = 60$. That is, with locality-aware scheduling, we can in theory convert costly inter-zone traffic into intra-zone traffic.

While the network traffic is greatly reduced in the locality-aware scheduler, one obvious limitation is on its skewness: Z0 has 6 applications — 300% of Z1. Statistically, it results in huge variance: $\frac{(6-4)^2+(2-4)^2}{2} = 4$. That is, the coefficient of variation (CV) is $\sqrt{4}/4 = 50\%$. An extremely low network traffic is definitely desirable in terms of performance; yet it indicates a low network utilization from the vendor's perspective. Therefore, we are interested in trading off a portion of network traffic to (significantly) improve CV.

We first re-ran the same workloads as discussed above by putting the same weight on load-balance and locality. That is, $\alpha = 1$ and $\beta = 1$. As shown in Table I (first two rows), the allocation is exactly the same as when considering no load-balance at all (i.e., Figure 10). This can be best explained by the coefficient of variance between zones not being significant enough to impact the weighted function that is dominated by the network traffic.

Next, we increased the weight on load-balance (i.e., vc): $\alpha = 1$ and $\beta = 2$. The results are reported in the bottom two rows in Table I. Now we observe A5 and A6 are placed in Z1 rather than Z0. That is, the increased weight on load-balance counters the network traffic on these two applications. Also note that, even though in this scenario both Z0 and Z1 have the same number of applications just like in vanilla Diego

TABLE I
FUNCTION SCORES AND PLACEMENT OF EIGHT APPLICATIONS ON TWO ZONES ($Z0, Z1$) IN TWO SETUPS (F^1, F^2): $\{\alpha = 1, \beta = 1\}$ AND $\{\alpha = 1, \beta = 2\}$ (SELECTED ZONE IS UNDERLINED)

	App	A0	A1	A2	A3	A4	A5	A6	A7
Z0	F^1	.50	.50	.39	.02	.12	.19	.24	.27
Z1	F^1	.50	.00	.19	.02	.12	.19	.24	.27
Z0	F^2	.67	.67	.37	.02	.15	.24	.25	.15
Z1	F^2	.67	.00	.24	.02	.15	.15	.11	.15

(i.e., Figure 7), the applications are different in each zone. It is only a coincidence: the placement is determined by completely different criteria.

In the scenario of $\{\alpha = 1, \beta = 2\}$, the overall network traffic is 240 because `A0 = A1` = 0, `A2 = A3 = A4 = A6` = 10, and `A5 = A7` = 100. While the traffic usage is higher than the strict network-optimal strategy, the ratio is still relatively low: $240/(220 \times 8) = 13.6\%$. In return, the system is perfectly load balanced. Comparing with the vanilla Diego scheduler, our proposed approach achieves the same load-balance but reduces the traffic from 600 to 240 — a $1 - \frac{240}{600} = 60\%$ reduction in network bandwidth.

VI. RELATED WORK

In nature, load-balance and data locality are fundamentally orthogonal to each other. Some of our prior work [17] tried to find a good trade-off between the two metrics in high-performance computing. This work, however, concentrates on the analytical model of load-balance and application performance in the context of containerized cloud services.

Scheduling is actively studied to improve the I/O performance at large scales. For instance, one of our prior works focused on batch scheduling of petascale systems [22]. The objective was to design and evaluate a batch scheduler with a holistic view of both system state and jobs' activities on an extreme-scale production system, Mira [13] at Argonne National Laboratory, a top 5 supercomputer in the world [16]. In contrast, this work is focused on the scheduling part of containerized cloud services with respect to network traffic in addition to disk I/O.

Ahn et. al. [1] proposed VM-level scheduling techniques to migrate micro-architectural resources such as shared cache and memory controller. The micro-architectural resources are not traditionally isolated at the VM layer, but manipulated by the intra-system. They showed that the proposed scheduling approaches are highly effective for cache sharing and non-uniform memory access (NUMA) affinity. There are more studies [2, 14] on the NUMA scheduling at the VM level, as well as adaptive VM scheduling optimized for non-concurrent workloads [18]. This work, on the other hand, targets a different set of micro-architectural resources (disk I/O bandwidth and network traffic) at a finer granularity (i.e., containers).

Scheduling is also researched to improve other aspects of the system. For example, one of the most interesting angles is power consumption [8, 11, 20]. These works showed that a well-tuned job scheduler could significantly reduce the power consumption and electricity bill.

VII. CONCLUSION

This paper identifies the containerized service scheduler as one of the performance limitations in the state-of-the-art design of PaaS. We pinpoint that a round-robin policy of placing applications/services on containers incurs significant performance overhead from both local disk I/O and network traffic. In tackling this problem, we first formulated it into a set of optimization problems and analyzed their complexities. We implemented and incorporated the proposed solution into an open-source PaaS project. We justified the correctness of the proposed work, which shows that the network traffic could be reduced by 60% with no side effect to load-balance.

As part of our ongoing effort, we are working on incorporating more factors into the scheduler such as local resource contention, in addition to evaluating the locality-aware scheduler at larger scales. Batch scheduling of applications and dynamic rebalancing of already deployed applications over their lifetimes are other areas of interest. In the future, we are looking to incorporate data compression (for example, our prior work [21]) into the job scheduler of containerized cloud services.

ACKNOWLEDGMENT

The authors would like to thank Prof. Ioan Raicu (Illinois Institute of Technology, USA) and Prof. Ion Stoica (University of California, Berkeley, USA) for insightful discussions.

REFERENCES

[1] J. Ahn, C. Kim, J. Han, Y.-R. Choi, and J. Huh, "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[2] Y. Cheng, W. Chen, X. Chen, B. Xu, and S. Zhang, "A user-level numa-aware scheduler for optimizing virtual machine performance," in *Advanced Parallel Processing Technologies*, 2013, vol. 8299.

[3] Cloud Foundry, "http://docs.cloudfoundry.org/," Accessed July 17, 2015.

[4] CloudWatch, "http://www.cloudwatch.in/," Accessed July 15, 2015.

[5] Diego Project, "https://github.com/cloudfoundry-incubator/diego-release," Accessed July 16, 2015.

[6] Docker container, "https://github.com/docker/docker," Accessed July 16, 2015.

[7] EMC/VMWare vSphere, "https://www.vmware.com/products/vsphere/," Accessed July 17, 2015.

[8] J. Hikita, A. Hirano, and H. Nakashima, "Saving 200kw and $200 k/year by power-aware job/machine scheduling," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008.

[9] IBM z/VM, "http://www.vm.ibm.com/overview/," Accessed July 17, 2015.
[10] Jelastic, "http://ops-docs.jelastic.com/cluster-features#c," Accessed July 16, 2015.
[11] O. Mammela, M. Majanen, R. Basmadjian, H. De Meer, A. Giesler, and W. Homberg, "Energy-aware job scheduler for high-performance computing," *Computer Science - Research and Development*, vol. 27, no. 4, 2012.
[12] Microsoft Hyper-V, "https://technet.microsoft.com/library/hh831531.aspx," Accessed July 17, 2015.
[13] Mira, "http://www.alcf.anl.gov/user-guides/mira-cetus-vesta," Accessed July 17, 2015.
[14] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu, "Optimizing virtual machine scheduling in numa multicore systems," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.
[15] Red Hat KVM, "http://www.redhat.com/en/files/resources/en-rh-kvm-kernal-based-virtual-machine.pdf," Accessed July 17, 2015.
[16] Top500, "http://www.top500.org/list/2014/06/," Published June 2014; Accessed September 5, 2014.
[17] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proceedings of IEEE International Conference on Big Data (BigData Conference)*, 2014.
[18] C. Weng, Q. Liu, L. Yu, and M. Li, "Dynamic adaptive scheduling for virtual machines," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC)*, 2011.
[19] Xen, "http://www.xenproject.org/," Accessed July 17, 2015.
[20] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka, "Integrating dynamic pricing of electricity into energy aware scheduling for hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.
[21] D. Zhao, J. Yin, K. Qiao, and I. Raicu, "Virtual chunks: On supporting random accesses to scientific data in compressible storage systems," in *Proceedings of IEEE International Conference on Big Data*, 2014.
[22] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/o-aware batch scheduling for petascale computing systems," in *Cluster Computing, IEEE International Conference on*, 2015.