Goldilocks: Adaptive Resource Provisioning in Containerized Data Centers

Liang Zhou, Laxmi N. Bhuyan, K. K. Ramakrishnan
Computer Science and Engineering Department
University of California Riverside, USA
lzhou008@ucr.edu, {bhuyan, kk}@cs.ucr.edu

***Abstract*—Power management in data centers is challenging because of fluctuating workloads and strict task completion time requirements. Recent resource provisioning systems, such as Borg and RC-Informed, pack tasks on servers to save power. However, current power optimization frameworks based on packing leave very little headroom for spikes, and the task completion times are compromised. In this paper, we design Goldilocks, a novel resource provisioning system for optimizing both power and task completion time by allocating tasks to servers in groups. Tasks hosted in containers are grouped together by running a graph partitioning algorithm. Containers communicating frequently are placed together, which improves the task completion times. We also leverage new findings on power consumption of modern-day servers to ensure that their utilizations are in a range where they are power-proportional. Both testbed implementation measurements and large-scale trace-driven simulations prove that Goldilocks outperforms all the previous works on data center power saving. Goldilocks saves power by 11.7%-26.2% depending on the workload, whereas the best of the implemented alternatives, Borg, saves 8.9%-22.8%. The energy per request for the Twitter content caching workload in Goldilocks is only 33% of RC-Informed. Finally, the best alternative in terms of task completion time, E-PVM, has 1.17-3.29 times higher task completion times than Goldilocks across different workloads.**

I. Introduction

Typically, data centers (DC) are over-provisioned so as to satisfy application's Service Level Agreements (SLA) at peak loads. Servers in data center usually operate at 20-30% utilization [1]–[3] and the network link utilizations are around 10% [4], [5]. Running servers and the Data Center Network (DCN) at such low utilizations wastes power [5]. This has prompted a large number of research efforts on power management in data centers, considering both packing tasks on servers and consolidating traffic in the DCN [5]–[8].

However, 'right sizing' the data center resources has proved to be challenging due to workload variability [9] and having to meet strict SLAs [10]. Also, to the best of our knowledge, no solution exists that minimizes power as well as task completion time. The difficulty arises in that a number of factors such as multi-dimensional resource constraints [11], server energy efficiency [12], fluctuating workloads [4], communication affinity [13], etc., have to be considered. A holistic approach is needed to strike a balance between power consumption and application's performance.

State-of-the-art task placement frameworks such as Borg [14] and RC-Informed [15] pack tasks in containers or Virtual Machines (VMs) into a few high utilized servers without violating the resource constraints. To increase the packing efficiency, Borg aims to reduce stranded resources [14] when only some but not all resources on a machine are fully allocated. RC-Informed, on the other hand, over-subscribes CPU resources [15]. Instead of increasing the packing efficiency, the mPP algorithm in pMapper [16] places VMs on servers having the lowest power increase per unit of utilization. On the other hand, E-PVM [17] places VMs on a server with the lowest utilization to leave a large headroom for load spikes and has good task completion time. These approaches improve either power consumption [14]–[16] or task completion time [17], but are unable to achieve improvements in both dimensions. Considering affinity between VMs while packing [13], [18] is a promising approach to improve power consumption as well as task completion time. However, a VM potentially communicates with a very large number of other VMs. For example, in a representative trace from Microsoft for search, the average number of distinct connections per VM is 45 [19]. The rich interactions between VMs makes it difficult to use locality-aware placement policies designed earlier [19].

In this paper, we present a graph-based approach, Goldilocks, to elegantly solve the complex resource 'right sizing' problem in data centers to optimize both power and task completion time. Goldilocks uses containers (as an example) to host tasks instead of VMs as they are more light-weight and flexible [20], [21], but can be easily applied to a VM-based data center as well. Unlike placing stand-alone containers [11], [14]–[18], Goldilocks partitions groups of containers before assigning the container groups to the servers. A prior work [22] maps the VM-clusters to server-clusters based on a clustering algorithm with simplistic assumptions, such as unlimited network bandwidth. Moreover, the data center power is not optimized.

In Goldilocks, two kinds of graphs are considered in the partitioning algorithm: a) a capacity graph, representing data center's total resources and b) a container graph with resource demands as the vertex weights and inter-container communication as edge weights. By running the recursive graph bipartitioning algorithm in METIS [23] with the min-cut objective function, containers with high communication frequency are grouped together. The load for each container group is automatically balanced in the algorithm. Goldilocks significantly improves the task completion time as containers with frequent communication are placed close together in the DCN topology. The power saving is achieved by packing groups of containers into a minimal number of servers, so that unused servers are turned off. We first solve the container group packing problem in symmetric Clos topology. We then extend our algorithm to the asymmetric topology with heterogeneous servers.

In addition to the graph partitioning algorithm, we re-examine the conventional wisdom regarding power management [1], [2], [5]. A common sense strategy for energy efficient data centers has been to run some servers close to 100% utilization (with a small safety margin [5]), and turn off idle servers for maximal power savings [9], [24]. However, SLA violations for latency-sensitive applications [8] can be a major concern when servers become over-loaded due to burstiness of the workload. In Goldilocks, we operate servers at *Peak Energy Efficiency* (60-80% utilization) [12], which is defined as the point to achieve the maximum number of operations completed per watt. Such a strategy saves more total server power and leaves a much larger headroom to deal with instantaneous load fluctuations.

Packing traffic in the network [5], [6], [25] has been proposed to save DCN power. However, the flow completion time [26] and scalability [6] becomes a major concern. Also, since the DCN contributes to only 15-20% of the total power consumption in a data center [5], [6], [25], we turn off the idle switches and links only after packing the containers, while avoiding the complexities of traffic scheduling. A few extra backup paths [6] are reserved for bursty traffic.

To demonstrate the feasibility of Goldilocks, we implemented a seamless Docker container migration framework on a 16-server testbed. For the Twitter content caching application, Goldilocks saves 11.7% power, with the variation seen in Azure cloud trace [15] and 22.7% power on the variability seen in Wikipedia trace [27]. Comparatively, the best of alternatives (i.e., Borg) saves 8.9% and 21%, respectively. Although Borg and Goldilocks have somewhat similar power saving results, the energy consumption per request for Borg is 3.5 times that of Goldilocks. Because of the locality-focused partitioning and large headroom for load fluctuation, Goldilocks produces at least 2.56 times of better task completion times, compared with any of the implemented alternatives. Our large scale trace driven simulation, based on Microsoft traces [19], reiterates the significant power saving and reduced task completion time of Goldilocks. Our major contributions include:

- We propose a holistic graph-based approach for resource provisioning in containerized data centers. This approach places containers closely to minimize power consumption and task completion time.
- A number of resources and performance considerations are taken into account, such as CPU, memory, network, migration overheads and workload fluctuation.
- We propose to achieve *Peak Energy Efficiency* for servers that maximizes performance for the energy consumed.
- We implement Goldilocks in a data center testbed, including seamless Docker container migration between servers. We show results both from the implementation and a large scale simulation based on a Microsoft Azure trace.

The rest of this paper is organized as follows. Section II analyzes the *Peak Energy Efficiency* of servers. Section III and Section IV present the graph-based resource provisioning algorithm on symmetric and asymmetric topology, respectively.

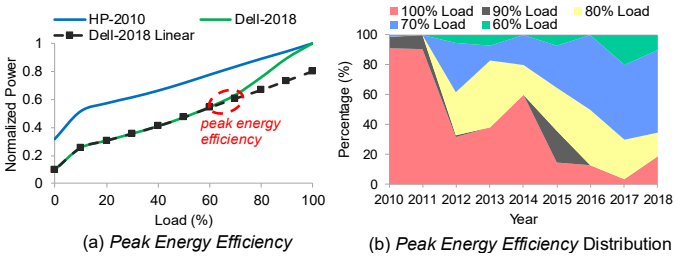


Fig. 1. The distribution of *Peak Energy Efficiency* utilization for the SPEC power benchmark [28].

Section V gives the implementation details. Section VI shows the implementation and evaluation results. Finally, Section VII concludes the paper.

II. Peak Energy Efficiency

In energy efficient computing, the basic assumption is that server power increases linearly as we increase the utilization [37], [38]. This linear power curve assumption is the basis for a number of efforts [3], [7], [8] to consolidate server load, turn off idle servers and thus save static server power. Usually, the servers are packed to 100% maximal utilization. However, the latency of application might increase due to longer queuing time. The SLA violation becomes a concern as there is little room to accommodate workload variation [10].

We observed that the server power increases quickly beyond a point on the load vs. power curve. Load means the quotient of current request rate and maximum possible request rate on the server. In this paper, we define *Peak Energy Efficiency* as achieving the maximum number of operations per watt [12]. When servers operate at higher utilization than the *Peak Energy Efficiency*, power increases faster than the load because of automatic frequency boosting [26] and increase in temperature requiring higher fan speeds [8]. Fig. 1 (a) shows the normalized power of 2 recent servers, as we vary the load. The power is normalized to maximum power consumption at 100% load, to avoid focusing on the differences among systems from different vendors. The major take-away of this result is that the power consumption vs. load was mostly linear until 2010, but not any more. As also observed with the Dell-2018 curve, power increases much faster beyond the *Peak Energy Efficiency* point [12]. For comparison, the dotted line shows what would be a linear increase if it was strictly power proportional.

The cubic power curve after *Peak Energy Efficiency* utilization in Fig. 1 (a) is due to Dynamic Voltage Frequency Scaling (DVFS) [2], [3], [26], [37] on modern server. In DVFS, power

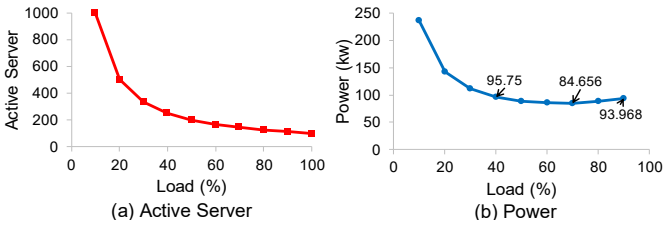


Fig. 2. Operating servers at the *Peak Energy Efficiency* utilization consumes the least total server power.

TABLE I
THE CONFIGURATION OF 5 DIFFERENT DATA CENTERS.

	# of Server	# of Switch	# of Link	Power Model
Google [29]	98304 (40G)	2048 ToR (32X40G up, 32X10/40G down) 3584 Fabric (32X40G up, 32X40G down)	147456	96W SoC server (Facebook 1S [30]), 630W ToR/Fabric switch (2 HPE Altoline 6940 [31])
Facebook [32]	184320 (10G)	4608 ToR (4X40G up, 48X10G down) 576 Fabric (48X40G up, 48X40G down)	36864	96W SoC server (Facebook 1S), 282W ToR (Facebook Wedge [33]), 1400W Fabric (Facebook 6 Pack [33])
VL2(96) [34]	46080 (10G)	2304 ToR (2X40G up, 20X10G down) 144 Fabric (96X40G)	9216	250W Microsoft blade server [30], 282W ToR (Facebook Wedge), 1400W Fabric (Facebook 6 Pack)
Fat-tree(32) [35]	32768 (10G)	1280 (32X40G)	2048	250W Microsoft blade server 315W switch (HPE Altoline 6940)
Fat-tree(72) [35]	93312 (10G)	6480 (72X10G)	10368	250W Microsoft blade server 315W switch (HPE Altoline 6920 [36])

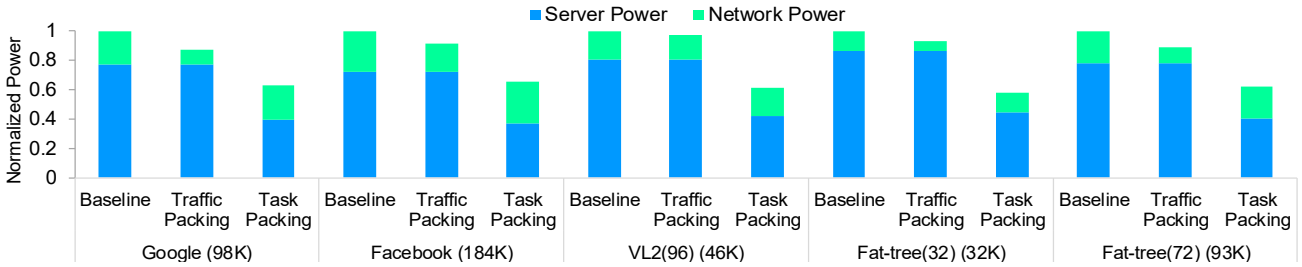


Fig. 3. The power breakdowns on 5 different data centers.

P equals $C * V^2 * f$, where C is capacitance, V is voltage and f is frequency. At low loads, the voltage V does not scale down further because it is already low. Only the frequency f scales down. This explains the linear power curve below the *Peak Energy Efficiency*. At high loads, both voltage V and frequency f will scale up, which results in the increase in power according to the cubic law. Fig. 1 (b) shows the *Peak Energy Efficiency* utilization results of 419 servers subjected to the SPEC request/response transaction workloads [28]. The *Peak Energy Efficiency* utilization for each server is obtained by analyzing the SPEC power benchmark results uploaded by vendors. This benchmark exercises the CPU, Cache, Memory, I/O as well as the operating system. 100% load is defined as the maximum number of requests that can be supported by the server. The Y-axis on Fig. 1 (b) is the share of each listed *Peak Energy Efficiency* utilization (i.e., 100% to 60% load) for a specific year. Each color represents one kind of *Peak Energy Efficiency* utilization. We can observe that most of the servers in 2010 have the *Peak Energy Efficiency* at 100% server utilization. During recent years, the *Peak Energy Efficiency* utilization of servers has already moved to the range of 60-80% utilization.

Let us examine how running servers at *Peak Energy Efficiency* can be beneficial in a data center context. There are two benefits: better power saving and higher tolerance for workload fluctuation. Consider placing a group of containers in a cluster of 1000 servers. Fig. 2 (a) shows that fewer servers are needed as we increase the load per server. The corresponding total power consumption is depicted in Fig. 2 (b). Servers have the same power model as Dell-2018 in Fig. 1 (a), which has *Peak Energy Efficiency* at 70% utilization. We can observe an obvious 'U' curve for total server power. The maximum power is saved when servers are packed only up to 70% utilization.

Lowering the utilization allows us to tolerate burstiness of data center workloads [4], [15], [39] and have shorter latency. Workloads across applications in a cloud data center might also be correlated [40]. We calculated the Pearson correlation [41] of 1500 VMs in the Microsoft Azure trace [15]. 99.8% of time the Pearson correlation is between 0.6 and 0.8, indicating (pair-wise) that VMs might 'burst' at the same time. Packing servers to the *Peak Energy Efficiency* leaves sufficient headroom to accommodate burstiness in the workload. Using just bin packing with a target of 100% server utilization [7], [8], tasks have to be migrated when the server becomes overloaded to avoid violating the application's SLA requirement.

In the DCN, we turn off idle switches and links. Table I gives the configuration of 5 different data centers such as Google's Jupiter [29] and Microsoft's VL2 [34]. Because the power models are not given [29], [32], [34], [35], we carefully select power models from the Open Compute Project [30], [33] to match the switch's port density and server's network bandwidth. For example, the Facebook's DCN topology has 10G servers, 16X40G Top of Rack (ToR) switch and 96X40G Fabric switch. So, they are mapped to Facebook 1S System on Chip (SoC) server [30], Facebook Wedge ToR switch and Facebook 6 Pack Fabric switch [33] accordingly.

Power breakdowns for the 5 different data centers are depicted in Fig. 3. At the baseline, all the servers are uniformly loaded at 20% utilization [1] and link utilization between ToR switch and next stage switch in the Clos topology [35] is 10% [4]. To ease the comparison, all the power results are normalized to baseline. The first take-away in Fig. 3 is that DCN only contributes around 20% of the total power for all the 5 data centers. The results are in line with prior works [5], [6]. Next, we focus on the power saving results of Traffic Packing in the network and Task Packing on servers. By saying Traffic Packing, we mean moving all traffic to the fewest number of

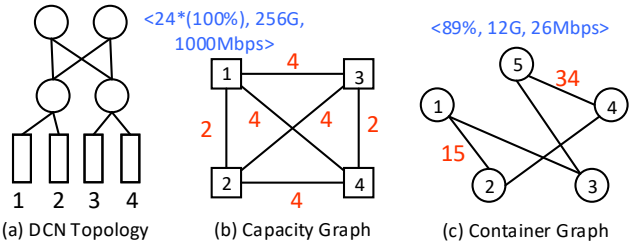


Fig. 4. Example for capacity graph and container graph.

links and switches as long as they are not overloaded. In Task Packing on servers, all the loads are packed into the fewest servers as long as the total utilization of the server is below a threshold. The results are obtained through mathematical analysis of bin packing [42]. The second take-away is that Traffic Packing on average can only save 8% of the entire data center's power, while Task Packing saves as much as 53% of total power. Thus, we should better do the Task Packing on servers to save most of the power.

III. Provisioning on Symmetric Topology

In prior works, such as E-PVM [17], RC-Informed [15] and Borg [14], each container is allocated to a server independently, ignoring the affinity or dependency between containers. Goldilocks leverages a holistic graph-based approach to solve the container placement problem, while minimizing power *and* task completion time. The algorithm is centered around the partitioning of two graphs: the capacity graph and container graph. Goldilocks partitions the containers into different balanced (in terms of aggregate resource demands) groups before assigning each container group to a subtree in the topology. The grouping is achieved by running the recursive bipartitioning algorithm on the container graph. With the min-cut objective function, the containers having high communication frequency are grouped together and then placed closer together in the data center.

A. Graph Construction

Capacity Graph: We construct a capacity graph, representing the total resource capacity in the data center. The vertex weight is a 3 dimensional vector *<CPU utilization, Memory usage, Network bandwidth>*. The edge weight is the length of the shortest path (i.e., number of links) between server pairs in the topology. As an example in Fig. 4 (a), we show a simple topology with 4 switches (circles) and 4 servers (rectangles). Its capacity graph is shown in Fig. 4 (b). The vector <24*(100%), 256G, 1000Mbps> means that each server has 24 CPU cores (with 100% utilization), 256GB memory and 1000Mbps network bandwidth (we choose to not factor in the disk size for the moment, assuming it is not likely to be a limiting factor). With the Clos topology allowing line rate communication between any server pair [35], we set the *Network bandwidth* in vertex weight as the link capacity of server's Network Interface Card (NIC). We relax this assumption in Section IV. We set the edge weight as path length, because each substructure in the Clos topology will automatically be grouped together during the graph partitioning for the max-cut, since inter-substructure edges always have the largest edge weight.

TABLE II
Vertex weight and edge weight of 4 data center workloads.

	CPU (%)	Memory (GB)	Network (Mbps)	Flow Count
Twitter Content Caching (Memcached)	33	4	24	4944
Web Search (Apache Solr)	32	12	1	50
Naive Bayes Classifier (Hadoop)	376	2	328	2
Media Streaming (Nginx)	54	57	320	25

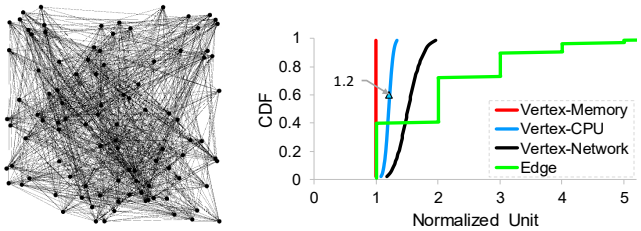


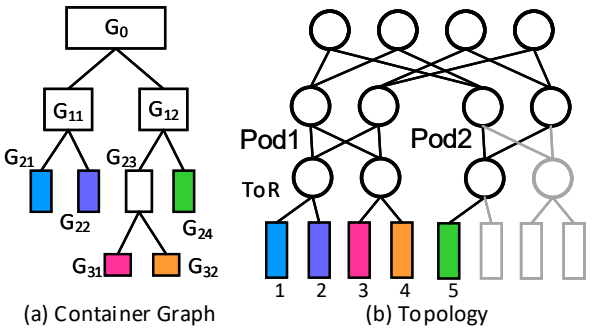
Fig. 5. Vertex weight and edge weight distribution (part b) in the Microsoft search trace [19] graph (part a, 100 vertices snapshot). Both vertex weight and edge weight in part (b) are normalized to the smallest value.

Container Graph: Each vertex in the container graph corresponds to a container in the workload. The vertex weight is the resource demand, in terms of CPU utilization, memory usage and network bandwidth. The edge weight is the number of distinct flows between the pair of containers. We aim to place together the containers with the most frequent communication. An example of the container graph is given in Fig. 4 (c).

We present the characteristics of different containerized applications, deployed in our testbed, in Table II. Each application instance is hosted in a Docker container. Columns 2-4 are the vertex weights and the last column is the edge weight. The method of obtaining these weights is described in Section V. Next, we show a part of the container graph for the Microsoft search trace [19] (which has 5488 vertices and 128538 edges) in Fig. 5 (a) for 100 vertices (IP range: 10.0.0.1 to 10.0.0.100 in the trace). The vertex weight and edge weight distribution are plotted in Fig. 5 (b). All the weights are normalized to the smallest value in the distribution. For example, the dot on the Vertex-CPU line with the value of 1.2 on the X-axis means its CPU utilization is 1.2 times larger than the smallest CPU utilization in the trace. In the trace, all the nodes performing the search indexing occupy 12GB memory for in-memory index accessing. So, the vertex weight is 1 for memory usage, after normalization for all vertices.

B. Container Partitioning and Assignment

At a high level, Goldilocks assigns a group of containers that have inter-dependencies together, rather than assigning each stand-alone container. The group is assigned to a substructure (a machine, a rack, a pod or a subtree) in the DCN topology. The substructure can be automatically found by recursively bipartitioning the capacity graph, using the max-cut objective function. The resource capacities of every server in the



(a) Container Graph (b) Topology

Fig. 6. Recursively bipartition the container graph until the resource demands of leaf nodes (in part a) can be satisfied by the resource capacity of servers (in part b).

substructure are factored as vertex weights in capacity graph. The group of containers are assigned to a substructure only if the containers' resource demands can be satisfied by the substructure.

Consider a container graph $G^c = (V^c, E^c)$, where V^c are the vertices and E^c are the edges. The number of vertices is given by $m = |V^c|$. A^c is the vector representing the resource demands of container vertex V^c. Similarly, $G^t = (V^t, E^t)$ is the capacity graph with N vertices . B^t is the vector representing the resource capacity of server V^t. The goal of partitioning on the container graph is to find n partitions P_1 to P_n such that:

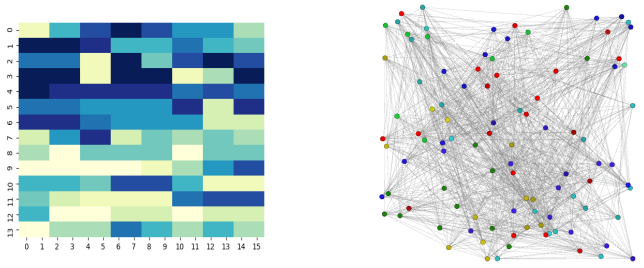
$$minimize \sum_{1 \le i < j \le n} |E^c_{ij}| \tag{1}$$

$$\forall\ P_i, \sum_{\forall j \in Pi} A^c_j \le B^t_i, where\ 1 \le i \le n \tag{2}$$

$$U_{P_1} \approx U_{P_2} \approx ... \approx U_{P_n} \tag{3}$$

n is determined at run-time of the partitioning algorithm rather than as a pre-defined static parameter. U_{P_i} refers the utilization of server V^t_i, where container group P_i is hosted. In equation (2), the algorithm stops bipartitioning the container graphs until the container group's resource demands can be satisfied by the server's resource capacity. At the final step of the partitioning algorithm, the container groups are assigned to servers. Equation (3) guarantees that the containers are uniformly distributed among n partitions. E^c_{ij} is the set of edges between partition P_i and P_j in container graph G^c. $|E^c_{ij}|$ refers the sum of edge weights for E^c_{ij}. Equation (1) guarantees that the cut is minimized when partitioning the container graph. Equation (2) guarantees the resource constraints are satisfied.

Fig. 6 shows the workflow for locality partitioning. G_0 in Fig. 6 (a) is the initial container graph and Fig. 6 (b) is the DCN topology. G_0 is partitioned into G_{11} and G_{12}. In the next iteration, G_{11} and G_{12} are bipartitioned because their resource demands exceed the server's resource capacity. Because the graph partitioning algorithm in open-source software, such as METIS [23], can tolerate some imbalances between container partitions, the number of vertices or total resource demands in G_0 does not need to be the power of 2. For example, the partition G_{23} can not be assigned to a server without violating resource constraints. But, its counter-part G_{24} is a little smaller than G_{23}, as the parent partition G_{12} cannot be ideally partitioned into two equally balanced partitions. Thus G_{24} can be successfully allocated to a server. The partition G_{23} has to be bipartitioned again to obtain G_{31} and G_{32}.



(a) Twitter Content Caching (b) Microsoft Search Trace

Fig. 7. Partition results for the Twitter Content Caching with 224 containers (part a) and Microsoft search trace with 100 vertices (part b).

Finally, the container groups to be assigned to the servers in Fig. 6 (b) are G_{21}, G_{22}, G_{31}, G_{32}, G_{24}. In addition to maximizing the intra-group locality, the partitions G_{21} and G_{22} that have the same parent partition are also assigned to the same rack in Fig. 6 (b) to maximize inter-group locality as well. By using the bipartitioning algorithm recursively, the inter-container communication is localized to a server, a rack, a pod or a subtree in the topology.

The partitioning algorithm in Goldilocks is implemented by using METIS [23]. METIS produces the optimal balanced min-cut partitioning. The computation time is reasonably small. For example, it just takes 285s to partition a graph with 1 million vertices. That means the epoch length in Goldilocks from one execution of container placement to the next can be short enough to quickly adapt to load changes in data centers. It is practical to deploy the algorithm for a large topology with millions of containers. In Fig. 7, we present two real partitioning results from our testbed experiment and also in the Microsoft trace graph. In Fig. 7 (a), there are a total of 224 Memcached containers (to simulate the Twitter content caching) which are represented by a cell. Each color in the results means a unique partition. Similarly, for the Microsoft trace graph [19] in Fig. 5 (a), there are 5 different container partitions shown in Fig. 7 (b).

IV. Provisioning on Asymmetric Topology

In Section III-B, we assumed a symmetric network topology with full bisection bandwidth and homogeneous servers. Although symmetric DCN topology is widely used [4], [5], [25], [26], [29], [34], [35], [42], switch and link failures can make the DCN topology asymmetric. Thus, servers are not inter-changeable because of imbalanced network bandwidth in various parts of the DCN [43]. The homogeneous server assumption might not hold as well [14], [44] because of legacy equipment even in a data center with custom-built servers [29].

In order to relax the symmetric topology and homogeneous server assumptions, we have to overcome two problems. First, only checking the network bandwidth usage at the server's NIC is not enough to ensure that resource demands are met,

as the full bisection bandwidth assumption doesn't hold with an asymmetric DCN topology. Second, Goldilocks partitions the containers into a few balanced groups and maps them to homogeneous servers in Section III-B. But now, each server has different computing and networking capabilities.

We now solve the problem of allocating m containers to an asymmetric tree topology without violating the resource constraints for each server and network link. The servers have various CPU cores, memory capacity and NIC bandwidth. Containers are labeled with a particular Group id for the sake of minimal power consumption and task completion time. Containers with same Group id should be placed close to each other in the topology. This is an NP-hard problem [3], [45]. So, we propose a heuristic algorithm in Goldilocks. In Section IV-A, the algorithm is first analyzed under the assumption that there is no inter-group communications between containers. Subsequently, we consider the more realistic case that container groups communicate with each other.

A. Assignment without Inter-Group Communication

The heuristic algorithm reuses the locality-focused partitioning in Section III-B. The bipartitioning algorithm stops when the resource demands of every container group can be satisfied by the average capacity of the heterogeneous servers. If m containers are partitioned into n groups, the initial number of active servers to place the m containers is n. Because the average capacity is used, the final number of active servers n' might be larger than the initial value n after allocating all the m containers. The abstraction of a Virtual Cluster, introduced in Oktopus [46] and Proteus [47] is leveraged by Goldilocks to represent a container group.

In Fig. 8 (a), a total of m containers are partitioned into n groups. Containers 1 to m/n are grouped together into a Virtual Cluster, and these containers are connected by a virtual switch. The virtual switch would represent a group of physical switches and links of the DCN, and it is expected that it can support the communication requirements among the containers in the Virtual Cluster. Consider the bandwidth requirement between the virtual switch and container i as being B_i. Conservatively, B_i should be larger than the sum of the intra-group container traffic and inter-group traffic for container i. To successfully place a container group in the topology, we have to meet the container group's CPU and memory demands on the servers as well as the bandwidth requirements on every underlay links of the Virtual Cluster. Validating the server side resource requirement is relatively straightforward. Therefore, we focus on the network link bandwidth reservation.

If we ignore the inter-group communications in Fig. 8 (a), the problem becomes placing n independent container groups (i.e., Virtual Clusters) in the data center without violating resource constraints. Considering a subtree structure T_i in the DCN topology as shown in Fig. 8 (b), the bandwidth of outbound link(s) between T_i and the remaining data center is the bisection bandwidth (for multiple paths) between T_i and the remaining data center. Under the general assumption, some containers have already been placed on subtree T_i. The

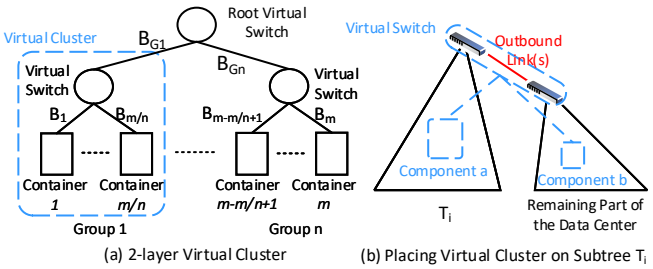


Fig. 8. In (a), each container group is abstracted as a Virtual Cluster [46]. All the running containers are represented by a 2-layer Virtual Cluster. When placing a virtual cluster of containers on the DCN topology, any underlay physical link divides the virtual cluster into 2 components (part b).

pending container group to be placed on subtree T_i is group j (i.e., Virtual Cluster VC_j). Because of resource constraints at subtree T_i, the maximum number of containers that can be placed at subtree T_i is component a as shown in Fig. 8 (b). The component b of VC_j has to be placed at the remaining part of the data center. The maximum size of component a for VC_j is bounded by the residual bandwidth at the outbound link(s) [46]. If we know the residual bandwidth at the outbound link(s) of subtree T_i, we can calculate the maximum size of component a for VC_j.

With the knowledge of maximum component a for VC_j at subtree T_i, the VC_j is placed at the smallest left-most subtree that can support all m/n containers of group G_j. In such a way, all the independent container groups are successfully placed in the data center. The subgraph of the DCN with n active servers might not be able to support the demands of n Virtual Clusters, because the average server capacity is fully used, or because we assumed full network bisection when determining the number of groups n. Goldilocks increases the value of n by the size of one pod when there are one or more Virtual Clusters that have to be placed. Because containers with same Group id are placed in the smallest subtree, locality is assured.

B. Assignment with Inter-Group Communication

Subsequently, we place the n container groups as a whole. The ignored inter-group bandwidth B_{G1} to B_{Gn} in Fig. 8 (a) should be considered now. Suppose we try to place a 2-layer Virtual Cluster with only 2 container groups (G_1 and G_2), similar to Fig. 8 (a). Using the same example in Fig. 8 (b), the Group 1 has already been placed in the data center with component a (G_{1a}) in the subtree T_i and component b in the remaining data center. The next step is to place container group G_2 with component G_{2a} in subtree T_i. The containers of G_2 that can not be placed at T_i because of resource constraints are called component G_{2b}. The problem we try to solve is deciding the maximum size of G_{2a} based on the residual bandwidth at outbound link(s) [46] of subtree T_i.

In the independent group placing, the size of G_{2a} is only limited by the intra communication between component G_{2a} and G_{2b}. But now we need to consider the inter-group communication between G_{2a} and G_{1b}. Because G_{1b} of group 1 is placed outside of subtree T_i. The bidirectional bandwidth to

be reserved on the outbound link(s) of subtree T_i, to satisfy group G_2, is given by R_{G_2}:

$$R_{G_2} = min\ (\sum_{\forall q \in G_{2a}} B_q, (\sum_{\forall r \in G_{2b}} B_r + \sum_{\forall s \in G_{1b}} B_s)) \quad (4)$$

$\sum_{\forall q \in G_{2a}} B_q$ is the sum of bandwidth for all the containers located in component G_{2a}. The required bandwidth at outbound link(s) to support G_2 could never be larger than the total bandwidth of component G_{2a}. $\sum_{\forall r \in G_{2b}} B_r$ is the intra-group communication between G_{2a} and G_{2b}. The inter-group communication between G_{2a} and G_{1b} is represented by $\sum_{\forall s \in G_{1b}} B_s$. If the sum of communication to the outside of subtree T_i (i.e., $\sum_{\forall r \in G_{2b}} B_r + \sum_{\forall s \in G_{1b}} B_s$) is smaller than the total bandwidth of component G_{2a}, we should reserve $\sum_{\forall r \in G_{2b}} B_r + \sum_{\forall s \in G_{1b}} B_s$ bandwidth at the outbound link(s). Otherwise, we should reserve $\sum_{\forall q \in G_{2a}} B_q$ bandwidth.

Next, we consider the case with n container groups (G_1 to G_n). If our algorithm already placed G_1 to G_{k-1}, the next one to be placed is the container group G_k. The total bandwidth of component G_{ka} and the intra-group communication for G_k are similar to the terms at equation (4). Now, we focus on the inter-group communication to the outside of subtree T_i, given by the following equation:

$$\sum_{1 \le y \le k-1} \sum_{\forall r \in G_{yb}} B_r + \sum_{k+1 \le z \le n} \sum_{\forall s \in G_z} B_s \quad (5)$$

$\sum_{1 \le y \le k-1} \sum_{\forall r \in G_{yb}} B_r$ is the communication between component G_{ka} and all the component b of placed groups (G_1 to G_{k-1}). For the pending container groups G_{k+1} to G_n, to be conservative, component b has all the containers in that group and the size of component a is 0. So, $\sum_{k+1 \le z \le n} \sum_{\forall s \in G_z} B_s$ is the communication to all the unplaced container groups. By knowing the required bandwidth at the outbound link(s), the 2-layer virtual cluster can be assigned to the data center. Note that the bandwidth constraints on any links l inside T_i has already been satisfied when placing container groups on the subtree T_l inside tree T_i, as the container groups are first placed on the smallest tree in the topology [46].

C. Discussion

Failure Resilience. In a data center, there are different fault domains for the sake of service surveillance [48]. Individual micro service of an application might be unavailable because of server failure, ToR switch failure or power supply failure [49], [50]. Large distributed systems such as Hadoop or the Google File System places replicas across different fault domains [13], [51]. In Goldilocks, we automatically place the replicas of the same service on different fault domains by giving negative edge weights for replica-replica or replica-primary edges in the container graph. We assume that the replica containers have already been labeled as such by the service owner based on prior knowledge. Using the min-cut graph partition algorithm, where the communication between vertices are positive in the container graph, the containers (replicas) with negative edge weights are thus partitioned into different container groups. Different container groups with the replicas are assigned to different fault domains in the DCN.

Migration Cost. The placement of containers from one execution to the next should be stable because of concerns on migration cost [16] such as application freeze time [52], migration traffic and the task startup latency [14]. Goldilocks is an epoch based scheduling system. The number of container migrations is the 'difference' between prior container grouping results and the current grouping results. In order to reduce the migration cost, we can leverage the incremental graph partitioning algorithm [53] to trade off the partitioning quality *and* the number of vertice migrations needed from old partition to the new partition. This topic is beyond the scope of this paper and will be our future work.

V. Implementation

Our testbed consists of 16 compute nodes and a management node. All the compute nodes have a 32 core AMD Opteron 6272 CPU, 64G memory, 1G Ethernet NIC, 50G SSD and 12TB shared RAID file system. All the servers run CentOS 7.2. In order to support the Docker container migration, we customized the Linux kernel version 4.14.24 and CONFIG_CHECKPOINT_RESTORE is enabled in the kernel. In the network, a leaf-spine topology is achieved by using 3 HPE 3800 series switches with 48 1G ports each. One physical switch is divided into 8 virtual switches (distinct VLANs) to simulate the leaf switches. Every pair of physical servers is connected to a leaf switch. Leaf switches are connected by 2 spine switches in a full mesh.

All the applications are hosted in Docker containers (version 17.09.1-CE). Docker container migration is an essential function for the placement of containers from one execution to the next. At the end of each epoch, the containers are migrated to the new servers based on the algorithms in Section III and IV. Goldilocks leverages the process checkpoint and restore technique [54], [55] to migrate containers between servers. The footprint of a process is typically stored as a disk image and then restored in the destination server. This process checkpoint & restore is a challenging task because a number of different pieces of information: namespace, iptables, cgroup and memory pages have to be obtained from the application's process tree. In our implementation, we use the Checkpoint Restore In Userspace (CRIU) [56] to achieve the Docker container checkpoint & restore.

The current CRIU has several limitations for container checkpointing. CRIU only checkpoints the file descriptor and inode information. We need to copy disk files and the Docker volume separately, for which we use rsync. Also, CRIU freezes the network connection by using iptable rules. It is essential that the same application-specific IP address exists at the destination server. In order to achieve seamless container migration, we adopt a similar policy as in VL2 [34]. The location-specific IP address is in subnet 192.168.0.0/16, but the application-specific IP address is in range 10.0.0.0/16. Node 16 in the cluster functions as a Docker swarm manager to also maintain the mapping between location-specific IP address and

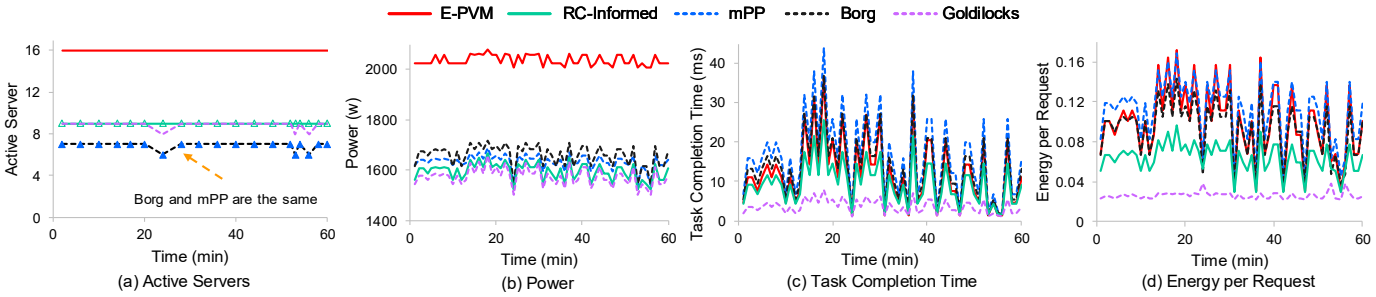


Fig. 9. Twitter content caching on Wikipedia trace pattern. Compared with the best alternative for each performance metric, Goldilocks on average has 4.6% less entire data center's power consumption, 2.57 times shorter task completion time and 3 times less energy consumption per request. Note that the Y axis for part (b) starts at 1400w.

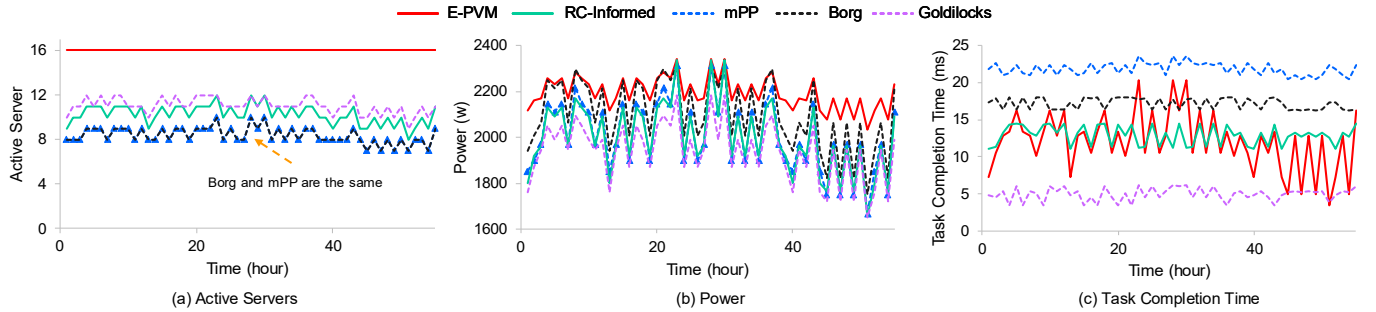


Fig. 10. Rich mixture of applications on Azure trace pattern. In part (b) with high data center load, the power consumptions of the alternatives sometimes are close to baseline E-PVM (only 1% power saving). Under same load, Goldilocks consumes 6.6% less power compared with E-PVM, because of operating servers at *Peak Energy Efficiency*. In part (c), Goldilocks on average has 2.6 times shorter task completion time than RC-Informed (best one in alternatives). Note that the Y axis for part (b) starts at 1600w.

application-specific IP address. All the Docker containers are attached to the same overlay network, with tunnelling achieved by using VxLAN.

Apart from the 16 compute nodes, we have a distinct management node to implement Goldilocks. The management node has an out-of-band connection to the switch and in-band connection to all the compute nodes. Prior to mapping containers to servers, the real-time server and network utilization have to be measured. The server utilization is obtained by polling the Docker metric pseudo-files. Because packet encapsulation is used in the VxLAN overlay network, the inter-container communication pattern is obtained by using the IPTraf tool [57] on servers to monitor the virtual Ethernet port for each container. We developed a migration controller in python to enforce migration rules, defining messages to orchestrate container checkpoint & restore. Servers can be remotely turned ON/OFF using an additional IPMI port. We estimate network power savings by determining which of the switches are idle.

VI. Evaluation

We evaluate Goldilocks on a testbed implementation and compare it with a number of alternatives published in the literature, viz., E-PVM [17], mPP [16], Borg [14] and RC-Informed [15]. We also use large scale trace-driven simulation to further evaluate Goldilocks. For E-PVM, containers are placed on the least utilized machines. In mPP, containers are placed on servers in a First Fit Decreasing manner, where items are considered in decreasing order of resource demand size. If the resource constraints are satisfied, the container is allocated to the server with least power increase per utilization unit. The difference between mPP and Goldilocks is that Goldilocks stops packing containers to the server if we reach the *Peak Energy Efficiency* (70% in the experiments), but mPP stops at the maximum server utilization (95%).

Borg is a cluster scheduling system from Google, including the capabilities of priority-based scheduling, task preemption, and task packing etc. [14]. In this paper, we only implement the task packing algorithm of Borg, meant to reduce stranded resources. The last algorithm we compared is the bucket-based RC-Informed policy. In RC-Informed, the CPU resource is oversubscribed because the allocated resource for containers are usually not full utilized. Currently, the CPU resource is 125% oversubscribed in RC-Informed [15]. To be fair, only the placement algorithms in the alternatives are implemented on our Docker container-based testbed, independent of whether it is a virtualization-based system (e.g., mPP and RC-Informed) or a Linux container-based system (e.g., Borg and E-PVM).

A. Testbed Results

1) Twitter Content Caching on Wikipedia Trace Pattern: We start with an experiment using the Twitter content caching workload. We set up a fixed, total number of 176 containers in the testbed, based on the goal of ensuring that the average server utilization for E-PVM is 32%, in line with prior observations [1]–[3]. In this experiment, the front-end container queries for a set of twitter terms from the Memcached container. If there is a hit for the query in the memory of Memcached, a success occurs for the get operation.

Fig. 9 shows the time-varying results across the alternatives and Fig. 11 shows the average values. The query pattern for the request per second (RPS) rate follows the Wikipedia

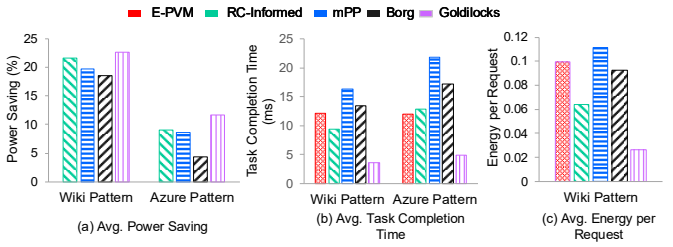


Fig. 11. On average for 2 trace patterns, Goldilocks achieves up to 31% better power saving, 2.62 times shorter task completion time and 3 times less energy consumption per request, compared with the best one among alternatives. Note that all the power saving results are calculated based on E-PVM. So, there is no power saving results to show for E-PVM in Part (a).

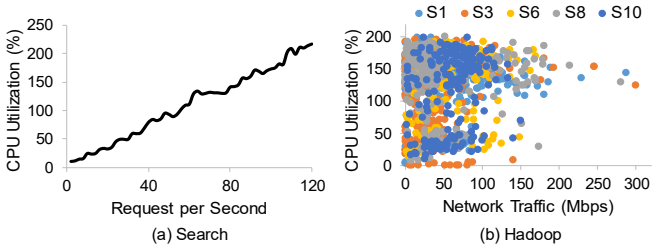


Fig. 12. Part (a), measured CPU utilization for Apache Solr search engine. Y axis is defined as the sum of all CPU core's utilization. Part (b), CPU utilization for varying traffic rates, with 5 servers in a 16-node Hadoop cluster running Facebook trace [58]. Each color is the data from one server S_i.

trace [27]. The RPS ranges from 44K to 440K across the entire testbed. In Fig. 9 (a), all the servers are active in E-PVM. Most of the time, Goldilocks and the bucket-based RC-Informed need 9 active servers, compared with 7 active servers in Borg and mPP, because servers in Goldilocks are limited to a maximum of 70% utilization. All the placement policies, except E-PVM, might need less active servers at low RPS, as the resource demands per container decrease. Goldilocks in Fig. 9 (b) consumes the least amount of power over the 60-mins experiment, because it seeks to achieve the *Peak Energy Efficiency*. The average power savings of mPP, Borg, RC-Informed and Goldilocks, compared with E-PVM on the Wiki pattern, are given in Fig. 11 (a). We can observe that Goldilocks saves 22.7% of the entire data center's power and outperforms all the alternatives.

More importantly, task completion time of queries improves significantly with Goldilocks in Fig. 9 (c). Goldilocks has the smallest task completion time during the 60-mins experiment. The average task completion time results are plotted in Fig. 11 (b), with Goldilocks giving the lowest result of 3.67ms. Among the alternatives, RC-Informed has the smallest task completion time of 9.44 ms. It is a bucket-based scheduling policy and has the lowest server utilization, compared with Borg and mPP. The average task completion time of Goldilocks is only 33% of RC-Informed, as we place the query generator and responder closely in the testbed.

The next important consideration is the energy consumed per request. Because of *Peak Energy Efficiency* (i.e., less power consumption) and locality-focused graph partitioning (i.e., shorter task completion time) in Goldilocks, it has the lowest energy per request results in Fig. 9 (d). We first take a look at the alternatives. On average, RC-Informed has the best energy per request result of 0.06, as shown in Fig. 11 (c). The energy per request result of Goldilocks is 3 times of better than RC-Informed, because of least power consumption and shortest task completion time in Fig 9 (b) and (c).

2) Rich Mixture of Applications on Azure Trace Pattern: In the previous experiments, we had a single Twitter workload and the number of containers was fixed at 176 and we vary the RPS from 44K to 440K. We now consider a rich mixture of applications, as is typical in a cloud data center and reflected in workload trace from the Microsoft Azure cloud [15]. We compare Goldilocks with the alternatives under this workload mix in terms of total number of containers needed in the testbed. The RPS for the containers supporting Twitter content caching is set at 2K for each connection. The total RPS for the entire set of services depends on the number of containers in the data center. In addition to the Twitter caching workload, we add 6 other background applications: the Apache Solr Search Engine, a movie recommendation system on Spark, Hadoop, a page rank on Spark and Cassandra database. The total number of containers ranges between 149 and 221, following the pattern found in Microsoft Azure trace [15]. We achieved this by stopping existing containers and launching new containers in the testbed. For the baseline E-PVM, the average server utilization gets to be as high as 54%.

Fig. 10 (a) plots the number of active servers for E-PVM, mPP, Borg, RC-Informed and Goldilocks. Because of the mix of applications, packing of containers does result in higher fragmentation. Goldilocks needs 2 more active servers to serve the same number of containers compared to Borg and mPP, because of 70% server utilization in Goldilocks. In Fig. 10 (b), Goldilocks again consumes the lowest total data center power. Compared with the baseline E-PVM, the power saving of Goldilocks increases from 6.6% to 18.8% when the total number of containers in the testbed decreases to 149. We observe that mPP, Borg and RC-Informed consume similar amount of power as the baseline E-PVM, at about 55% average server utilization. The reason is that these scheduling policies keep packing containers until reaching maximum server utilization, where the server power increases non-linearly. At the same average server utilization, Goldilocks consumes 6.7% less power compared with the base line E-PVM. Similarly, we plot the average power saving results compared with E-PVM in Fig. 11 (a). On average, Goldilocks achieves 11.7% power saving and the best one in alternatives (RC-Informed) has power saving of 8.9%. Fig. 10 (c) compares the task completion times. Goldilocks has the shortest task completion time, by far, for Twitter queries. As shown in Fig. 11 (b), the average task completion time for Goldilocks is 4.9 ms. The best, among the others compared is E-PVM, but even that has 2.5 times higher average task completion time.

B. Simulation Results

In addition to the testbed implementation, we also performed a flow-level, large scale simulation with a 28-ary fat tree topology, with a total of 5488 servers and 980 switches. The power model we used for servers is Dell Power Edge R940 [28] and the power model for switch is HPE Altoline

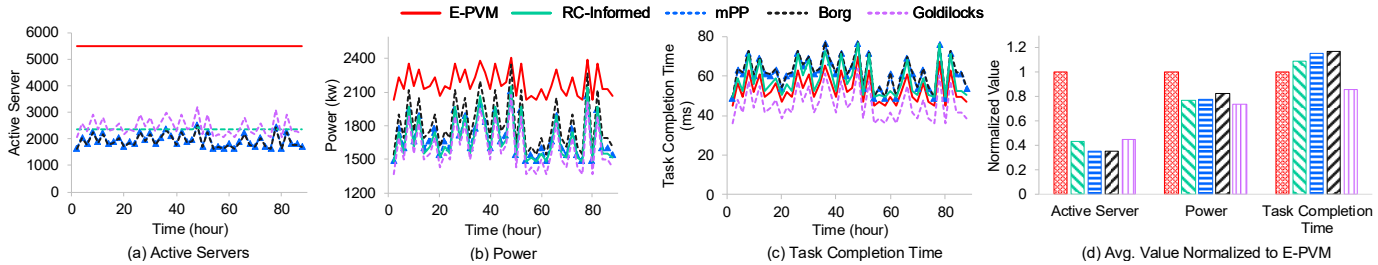


Fig. 13. Trace-driven simulation proves that the power saving (5% better) and task completion time (22% better) improvement of Goldilocks still holds at the data center with 5488 servers and 49392 containers. Note that the Y axis for part (b) starts at 1200 kw.

6940 [33]. The flow level simulation uses the processing time distribution for search queries based on a micro-benchmark of measurements made in our testbed. There are a total of 49392 containers in the topology, targeting a 20-30% server utilization [1]–[3] for the baseline E-PVM. We then simulated Goldilocks, mPP, Borg and RC-Informed to get the power savings and improvements in task completion time.

Since the trace [19] used for the simulation has only the flow level information, we obtain the resource demands on the servers through experiments in our testbed with a workload matching the trace's network traffic pattern and then measured the server resource utilization. In the trace, there are two kinds of traffic: search queries with flow sizes ranging from 1.6KB to 2KB and background update traffic with 1MB to 50MB flow sizes. We assume for the purposes of our simulation that the background traffic is Hadoop traffic because the URL crawling in search is based on the Map-Reduce framework.

In Fig. 12 (a), we deploy the Apache Solr search engine in the testbed and vary the search request rate. Since the maximum number of connections per Index Serving Node (ISN) in the trace is 120, we increase the request rate up to 120 RPS. The memory usage stays at 12G throughout. Fig. 12 (b) shows the background update traffic generated in our testbed deployed as a 16-node Hadoop cluster running the Facebook job trace [58]. The aggregate traffic and corresponding CPU utilization are then measured. Each color in the figure represents the data from one of the slave nodes in the Hadoop cluster. As shown in Fig. 12 (b), there are multiple dots with the same network traffic rate (X-axis value). In the simulation, a randomly chosen CPU utilization (Y-axis value) is chosen for a selected network traffic rate. The resource demands on the servers is the result of the sum of these 2 applications.

Fig. 13 (a) shows the number of active servers over a period of 88 hours. Because E-PVM always chooses the least utilized server, all of the 5488 servers in the topology are active. Borg and mPP pack the containers to achieve 95% server utilization and both of them have the least number of active servers. RC-Informed is a bucket-based scheduling policy, and the number of active servers is constrained by the total reserved resources for each container rather than the real-time resource utilization in the simulation. That is why RC-Informed needs 2358 active servers most of the time. Although Goldilocks needs more active servers compared with the other container packing policies, it consumes the least amount of power, as shown in Fig. 13 (b). Goldilocks stops packing containers when the server reaches its *Peak Energy Efficiency* (70% in this simulation). The baseline, E-PVM, consumes the highest data center power because of no saving in static server power. The power consumption of mPP, Borg and RC-Informed are similar, within 150kw of each other.

Finally, we take a look at the task completion time of search queries in Fig. 13 (c). Goldilocks has the shortest task completion time because of large server headroom and good locality. The average server load on E-PVM is around 26% to 40%. mPP, Borg and RC-Informed target at 95% [15] server utilization. This high server load contributes to their increased task completion time compared with E-PVM. In Fig. 13 (d), we see the average values of active server, power consumption and task completion time. All the values are normalized to the values of baseline E-PVM. Goldilocks has the lowest power consumption (27% power saving compared with E-PVM) and shortest task completion time (0.85 of E-PVM). Compared with the best alternative for each performance metric, Goldilocks consumes 5.2% less power than RC-Informed and achieves 15% shorter task completion time than E-PVM.

VII. Conclusion

Goldilocks is a holistic framework that solves the complex 'right sizing' provisioning problem in containerized data centers. As part of Goldilocks, a novel graph-based locality aware container placement scheme is proposed, yielding minimal power consumption *and* task completion time. Instead of directly taking the existing linear power model assumption in the literature, we found that current servers are more power efficient when operating at the *Peak Energy Efficiency* point. This *Peak Energy Efficiency* also leaves adequate headroom for burstiness in the workload. Our testbed implementation on the Twitter content caching workload proves that Goldilocks saves up to 22.7% of the entire data center's power. The energy per request result in Goldilocks is 3 times better than RC-Informed, which consumes the least energy per request among the compared alternatives. Because of Goldilocks' locality aware placement and large headroom for load spikes, it has at least 2.56 times better task completion time than any of the compared alternatives. Simulations of a large scale data center environment also shows that our power and performance improvements are still achieved at scale.

Acknowledgment

This research was partly supported by NSF grants 1815643 and 1763929.

REFERENCES

[1] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," in *Proc. ACM ASPLOS*, 2009.

[2] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proc. IEEE/ACM MICRO*, 2015.

[3] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proc. IEEE ISCA*, 2014.

[4] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM SIGCOMM*, 2015.

[5] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks," in *Proc. USENIX NSDI*, 2010.

[6] N. Vasić, P. Bhurat, D. Novaković, M. Canini, S. Shekhar, and D. Kostić, "Identifying and using energy-critical paths," in *Proc. ACM CoNEXT*, 2011.

[7] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," in *Proc. IEEE INFOCOM*, 2011.

[8] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion, "Energy proportionality and workload consolidation for latency-critical applications," in *Proc. ACM SoCC*, 2015.

[9] M. Shahrad, C. Klein, L. Zheng, M. Chiang, E. Elmroth, and D. Wentzlaff, "Incentivizing self-capping to increase cloud utilization," in *Proc. ACM SoCC*, 2017.

[10] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar, "Timetrader: Exploiting latency tail to save datacenter energy for online search," in *Proc. IEEE/ACM MICRO*, 2015.

[11] W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive resource provisioning for the cloud using online bin packing," *in IEEE Transactions on Computers*, vol. 63, pp. 2647–2660, 2014.

[12] D. Wong, "Peak efficiency aware scheduling for highly energy proportional servers," in *Proc. IEEE ISCA*, 2016.

[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *in Proc. USENIX OSDI*, 2004.

[14] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. EuroSys*, 2015.

[15] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proc. ACM SOSP*, 2017.

[16] A. Verma, P. Ahuja, and A. Neogi, "pmapper: Power and migration cost aware application placement in virtualized systems," in *Proc. Springer International Conference on Middleware*, 2008.

[17] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, "An opportunity cost approach for job assignment in a scalable computing cluster," *in IEEE Transactions on Parallel and Distributed Systems*, vol. 11, pp. 760–768, 2000.

[18] K. Zheng, X. Wang, L. Li, and X. Wang, "Joint power optimization of data center network and servers with correlation analysis," in *Proc. IEEE INFOCOM*, 2014.

[19] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proc. ACM SIGCOMM*, 2010.

[20] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove, "Picocenter: Supporting long-lived, mostly-idle applications in cloud environments," in *Proc. ACM EuroSys*, 2016.

[21] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proc. USENIX NSDI*, 2008.

[22] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM*, 2010.

[23] "Metis - serial graph partitioning and fill-reducing matrix ordering," http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[24] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes," in *Proc. ACM SoCC*, 2016.

[25] X. Wang, Y. Yao, X. Wang, K. Lu, and Q. Cao, "Carpo: Correlation-aware power optimization in data center networks," in *Proc. IEEE INFOCOM*, 2012.

[26] L. Zhou, C. Chou, L. N. Bhuyan, K. K. Ramakrishnan, and D. Wong, "Joint server and network energy saving in data centers for latency-sensitive applications," in *Proc. IEEE IPDPS*, 2018.

[27] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *in Elsevier Comput. Netw.*, vol. 53, pp. 1830–1845, 2009.

[28] "Specpower_ssj 2008," https://www.spec.org/power_ssj2008/.

[29] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *Proc. ACM SIGCOMM*, 2015.

[30] "Open compute project server design specification," http://www.opencompute.org/wiki/Server/SpecsAndDesigns.

[31] "Hpe altoline 6940 switch series," https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04741125.

[32] "Introducing data center fabric, the next-generation facebook data center network," https://code.facebook.com/posts/360346274145943/.

[33] "Open compute project switch design specification," http://www.opencompute.org/wiki/Networking/SpecsAndDesigns.

[34] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009.

[35] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, 2008.

[36] "Hpe altoline 6920 switch series," https://h20195.www2.hpe.com/v2/getpdf.aspx/c04680998.pdf.

[37] D. Wong and M. Annavaram, "Knightshift: Scaling the energy proportionality wall through server-level heterogeneity," in *Proc. IEEE/ACM MICRO*, 2012.

[38] Y. Liu, S. C. Draper, and N. S. Kim, "Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers," in *Proc. IEEE ISCA*, 2014.

[39] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM IMC*, 2010.

[40] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "Hug: Multi-resource fairness for correlated and elastic demands," in *Proc. USENIX NSDI*, 2016.

[41] "Pearson correlation coefficient," https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[42] Q. Yi and S. Singh, "Minimizing energy consumption of fattree data center networks," *in ACM SIGMETRICS Perform. Eval. Rev.*, vol. 42, pp. 67–72, 2014.

[43] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM SIGCOMM*, 2014.

[44] A. R. Curtis, S. Keshav, and A. Lopez-Ortiz, "Legup: Using heterogeneity to reduce the cost of data center network upgrades," in *Proc. ACM CoNEXT*, 2010.

[45] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A flexible model for resource management in virtual private networks," in *Proc. ACM SIGCOMM*, 1999.

[46] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. ACM SIGCOMM*, 2011.

[47] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *Proc. ACM SIGCOMM*, 2012.

[48] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *Proc. ACM SIGCOMM*, 2012.

[49] J. Dean, "Designs, lessons and advice from building large distributed system," in *ACM LADIS keynote talk*, 2009.

[50] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM*, 2011.

[51] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proc. ACM SOSP*, 2003.

[52] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. USENIX NSDI*, 2005.

[53] C.-W. Ou and S. Ranka, "Parallel incremental graph partitioning," *in IEEE Trans. Parallel Distrib. Syst.*, vol. 8, pp. 884–896, 1997.

[54] J. Ansel, K. Arya, and G. Cooperman, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *Proc. IEEE IPDPS*, 2009.
[55] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *Proc. USENIX ATC*, 2007.
[56] "Checkpoint/restore in userspace," https://www.criu.org/Main_Page.
[57] "Iptraf," http://iptraf.seul.org/.
[58] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *in Proc. VLDB Endow.*, vol. 5, pp. 1802–1813, 2012.