
Proceedings Paper:
Dziurzanski , Piotr and Indrusiak, Leandro Value-Based Allocation of Docker Containers. In: Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing. The 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2018.03.21 - 2018.03.23, Cambridge, UK. (In Press)

Value-Based Allocation of Docker Containers

Piotr Dziurzanski, and Leandro Soares Indrusiak
Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK
{Piotr.Dziurzanski, Leandro.Indrusiak}@york.ac.uk

***Abstract*—Recently, an increasing number of public cloud vendors added Containers as a Service (CaaS) to their service portfolio. This is an adequate answer to the growing popularity of Docker, a software technology allowing Linux containers to run independently on a host in an isolated environment. As any software can be deployed in a container, the nature of containers differs and thus assorted allocation and orchestration approaches are needed for their effective execution. In this paper, we focus on containers whose execution value for end users varies over time. A baseline and two dynamic allocation algorithms are proposed and compared with the default Docker scheduling algorithm. Experiments show that the proposed approach can increase the total value obtained from a workload up to three times depending on the workload heaviness. It is also demonstrated that the algorithms scale well with the growing number of nodes in a cloud.**

***Keywords*—Container orchestration, Market-based heuristics, Containers as a Service, Cloud computing.**

I. Introduction

The release of the Docker software in 2013 changed software deployment and orchestration in Linux-based cloud systems. Since that time, the operating system virtualisation has gained popularity due to much higher performance and flexibility in comparison with a traditional, hypervisor-based virtualisation, offering sufficient isolation for numerous applications [4]. It is not surprising then that the most popular cloud vendors, as Amazon Web Service, Google Cloud Platform, Microsoft Azure or Red Hat OpenShift added Containers as a Service (CaaS) to their cluster management systems (CMSs). These services offer container engines, orchestration and the underlying computing resources.

Initially, Docker containers were executed on a single machine, but soon a few of orchestration software managing a number of nodes in a cluster emerged, such as Docker Swarm [5] or Google Kubernetes [1]. These systems, however, perform best with the most typical cloud usage patterns, such as Internet services' high availability or load balancing for microservices [8]. It is assumed that Docker Swarm has no a priori information regarding the workload or the containers' resource requirement. Thus its only available allocator at the time of writing this paper is called *Spread*, which basically replicates a container on different nodes in a round-robin like fashion. More sophisticated allocation strategies are possible by defining CPUs or memory reservations and limits. Additionally, some affinity rules can be added to run a container on a computer with certain user-defined labels. However, all the extra configuration has to be performed manually and is rather laborious [18].

In [14], based on an example of distributed machine learning, it was shown that the default Docker allocating solutions are inferior to custom ones if a workload differs from the typical usage patterns mentioned above. Similarly, the workload considered in this paper is comprised of a relatively large set of containers that perform transformational computations, i.e. compute results from input values and then stop, which is in contrast to interactive or reactive services omnipresent in the Internet or IoT. Such usage pattern is characteristic to numerous real-world problems related to distributed optimisation [19]. For such workload type, the features of the native clustering for Docker, named Docker Swarm, such as creating replicas of a certain container, rolling updates or load balancing based solely on the ingress traffic are of limited usage [15]. Kubernetes, on the other hand, is rather a complicated system and requires essential manual configurations to tie together its components (e.g. etcd, flannel), but its high availability, scalability and service discovery features again suit more to the interactive Internet services than transformational computations [14]. Those orchestrator tools are not capable of benefiting from an extra knowledge originating from the prior execution of the same containers, which can be used for the execution time estimation [17], [13]. The traditional Docker orchestration tools are not capable of prioritising the container execution considering the value of the container execution results to the end users. In this paper, this problem is addressed using market-based heuristics.

The above considerations lead to the conclusion that, despite the popularity of various container orchestration software, there is still a need for creating new ones, aiming at executing specific workload types.

II. Related Work

The most prominent container orchestrate systems, such as native Docker Swarm [5] or popular Google Kubernetes [1] follow the monolithic architecture rather than the shared-state one, where parallel schedulers compete for cloud resources in a free-for-all manner [12], or two-level, in which a number of independent subclusters with their own schedulers requests resources from the only second-level scheduler [6]. Similarly, the majority of the orchestrators proposed in academia, e.g. ACO [9] or DORM [15], applies the same approach. The simplicity of the monolithic architecture allows the researchers to analyse and evaluate the influence of various aspects of scheduling algorithms, as dominant fairness policy and resource adjustment overhead in [14] or fuzzy-logic-based approaches in [15]. Similarly, in this paper, a monolithic cluster scheduler is used to facilitate the analysis of value-based container allocation.

The main goal of market-inspired heuristics is to allocate jobs to processors in a way that the overall value is maximal. In [16], jobs' values have been assumed to be fixed, whereas

in [3] it has been assumed that a job value can change over time. In this case, the value can be described with a so-called *value curve* of a job, a function whose domain represents the computation time with the origin at the release time of the container, whereas the codomain represents the values themselves [10]. A number of value-related heuristics have been compared in [7], highlighting the benefits originating from the access to historic execution data (profiling). In this paper, a similar assumption is made: the estimated execution time of each container, based on historically measured executions, is known and used to support the allocation decisions.

From the literature survey, it follows that there were no prior works related to the benefits of value-based scheduling of Docker containers. Since the market-based heuristics proved to be beneficial in a traditional task scheduling problem in the high performance computing domain [2], it may be expected that they will be similarly advantageous when applied to containers. This hypothesis is investigated in this paper.

III. System Model and Problem Formulation

Cluster platform CP contains a set of k nodes $N = \{N_1, \ldots, N_k\}$ capable of executing containers (i.e. running a container engine). Each node can execute one or more containers of workload $\Gamma = \{C_1, \ldots, C_n\}$. The nodes are heterogenous and their computation speed difference is expressed with so-called *calibration coefficient* ζ_i, $i \in \{1, \ldots, k\}$, denoting a ratio between empirically measured execution time of a set of container benchmarks on node N_i divided by the execution time of the same set of container benchmarks on a reference unit.

Each container C_j, $j \in \{1, \ldots, n\}$, is executed on node N_i in a time slot proportional to the so-called *CPU shares* $\xi_j \in \{1, \ldots, 1024\}$ of this container (the value of the maximum share is taken directly from the Docker's *–cpu-shares* flag). Assuming that the sum of all the CPU shares of containers Γ_{N_i} executed on node N_i equals Ξ_i, container C_j gets $\xi_{j,i} = \frac{\xi_j}{\Xi_i} \cdot 100\%$ of the CPU time of node N_i.

In CP, there is a global resource manager, named *DockerManager* that coordinates the execution of containers submitted by the users. It is responsible for serving the incoming requests and allocates the containers to nodes.

The estimated execution time of each container C_j is known and equals ET_j. Such information can be obtained from the previous executions of the containers in the cluster. The assumption of the existence of such data is quite common in the cloud computing domain, e.g. [17], [13]. The arrival time of container C_j is denoted as AT_j.

A value curve VC_j of container C_j is a function of the value of the container's job to the end user depending on the completion time of the job [3], [10]. A value curve has its maximum value $Vmax_j$ from the moment of the container release time and does not increase, as shown in Figure 1. At certain time the value curve assumes zero value and since that time there is no benefit from computing the task. The job can be then dropped in order not to waste resources. We assume a value curve is given for each container, as this reflects its business importance as assessed by the end user. The description of the value curve generation is out of scope of this paper. Some guidance can be found, for example, in [7].

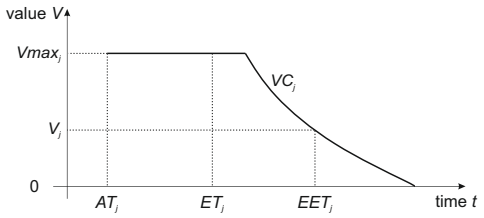

Fig. 1. An example value curve of container C_j

The reduction in the container value due to delay can be determined by observing the value of the value curve at the delayed completion time. Late completion of a container execution may result in zero value and thus the computation becomes useless for the end user. Further, the energy spent on such computation can be considered as wasted. Therefore, the container's job request may be rejected if the zeroth value is expected from executing it.

The concept of estimated execution time ET_j of container C_j is different from expected execution time EET_j of the same container. The latter one denotes the expected real computation time of C_j and considers the influence from other containers allocated to the same node and CPU share ξ_j. It is then expected that the value V_j obtained from container C_j would be equal to the value of the associated value curve VC_j at time point EET_j, $V_j = VC_j(EET_j)$.

With β_i we denote the sum of expected values of all the containers executed on node N_i (i.e. the value from the value curve at the time point of the expected computation completion) divided by the sum of their maximal values, which can be expressed with formula

$$\beta_i = \begin{cases} 0 & \text{if } \Gamma_{N_i} = \emptyset, \\ \frac{\sum_{j:C_j \in \Gamma_{N_i}} V_j}{\sum_{j:C_j \in \Gamma_{N_i}} Vmax_j} & \text{otherwise.} \end{cases} \quad (1)$$

Notice that $\beta_i \in [0, 1]$.

A node N_i is treated as overloaded if β_i is below a certain threshold $\lambda \in (0, 1)$, i.e.

$$\beta_i < \lambda. \quad (2)$$

For example, if Γ_{N_i} includes two containers with $Vmax_1 = 100$ and $Vmax_2 = 50$, but due to the expected execution time of these containers $V_1 = 50$ and $V_2 = 40$, the value of $\beta_i = 0.6$. If λ is set to, e.g., 0.7, then N_i is treated as an overloaded node. The influence of various values of λ on the total workload value is discussed in the experimental result section.

In CP, containers $\Gamma = \{C_1, \ldots, C_n\}$ are submitted by end users at various time points and need to be allocated to the processing cores of the cluster nodes $N = \{N_1, \ldots, N_k\}$ in a way that the total value obtained from the executed containers, $\sum_{j=1}^{n} V_j$, is maximised.

To recap, the problem considered in this paper can be briefly described with the following features:

- Input: Workload, i.e., container set $\Gamma = \{C_1, \ldots, C_n\}$, value curve of each container VC_j, arrival time of each container AT_j, $j \in \{1, \ldots, n\}$, nodes of the CaaS platform $N = \{N_1, \ldots, N_k\}$ with different *calibration coefficients* ζ_i, $i \in \{1, \ldots, k\}$.
- Constraints: Limited computational power on each node N_i of the cluster platform CP.
- Objective: Maximize the total value $\sum_{j=1}^{n} V_j$ obtained from the execution of containers in Γ.

For each container submitted by an end user, the allocation process conducted by *DockerManager* selects the node for executing container C_j and its CPU share $\xi_j \in \{1, \ldots, 1024\}$. The container may be also not allocated to any node if the allocation algorithm finds the value obtained after its execution inferior to the value obtained from CP without execution of C_j. In certain situations (detailed later in this paper), an earlier allocated container may be preempted from the selected node to provide more computational power to the newly allocated container.

IV. Value-Optimising Container Allocation Approaches

To determine the baseline, we assumed that the whole workload Γ is known in advance and thus it can be scheduled statically to the cluster platform CP. This assumption is rarely applicable to practical scenarios and the purpose of this approach is solely for comparison with other, dynamic approaches, described later in this section.

As the computation of this static allocator is performed off-line, its execution time is not crucial and thus heuristics with even high computational complexity may be employed. In this paper, a genetic algorithm, one of the most popular but resource intensive heuristics, is used for this purpose. Each chromosome in the genetic algorithm contains genes of two types. The odd n genes indicate the target nodes for n containers or the rejection of it, $N_{C_j} \in \{\emptyset, N_1, \ldots, N_k\}$, whereas the remaining n genes specify the containers' CPU share, $\xi_j \in \{1, \ldots, 1024\}$.

In real-world HPC scenarios, it is quite unlikely that the workload is known a priori [13]. Then the target node together with the CPU share of a container need to be determined just upon the container's release.

The first proposed dynamic approach is greedy as for each released container C_j it searches for the mapping N_{C_j} and CPU share ξ_j maximising the overall value at the given time point. For each released container, $1023 \cdot k + 1$ possibilities are evaluated (extra "+1" for the possibility of the container rejection). The computational complexity of the algorithm is then $O(n \cdot k)$ where n is a number of containers. In an implementation of this algorithm, some larger stride (the amount by which the index is increased each loop iteration) for the for loop browsing through the CPU shares can be applied to increase the speed of the approach. The influence of the stride size on the total value and execution time is evaluated experimentally in section VI.

In order to go beyond the search space of both the algorithms presented above, we propose to remove the already allocated containers whose expected value is relatively low. This additional functionality may be beneficial when the system is overloaded and thus the majority of containers would be executed well beyond their estimated execution times [13]. Then the resources regained by removing one container may be used in a more beneficial way when allocated to another container.

The main difference between the greedy dynamic allocation with preemption and the previous approach is the fact that, in case of a node overload, during allocation of container C_j to each node $N_1, \ldots, N_k$, one particular container already allocated to this node is selected as a victim. This container is to be removed from the cluster. It may be either suspended (using the *docker pause* command) or killed (using the *docker stop* command).

In section VI, two criteria for victim selection are experimentally evaluated: the lowest expected value and the lowest value density. These victimisation criteria among others, such as the minimal remaining value, are described in [13].

V. Implementation Issues

The dynamic algorithms described in the previous section have been implemented and used with the original Docker engine in form of two software modules, namely *DockerManager* and *DockerWorker*. The former one is run on a machine where Docker may or may not be installed, whereas the latter requires the presence of the Docker daemon, which is depicted in Fig. 2. In this figure, VFS denotes Virtual File System, an abstraction layer on top of a concrete file system used in Linux OS and the remaining blocks are described below.

The responsibilities of DockerManager are mainly related to the selection of a DockerWorker instance for executing a particular container. The DockerManager module can allocate/deallocate a container into/from a certain DockerWorker instance. A container can be started and stopped. Such parameters as: a period of a container, a quota of a container and container CPU shares can be defined. A number of statistics are available, such as a container CPU usage since its start, total value for all containers executed by a certain DockerWorker, the remaining execution time of a certain container (based on its execution time), an execution time ratio between a certain DockerWorker and a reference DockerWorker instance (used for determining its calibration coefficient ζ_i). There is a possibility of checking how these statistics would change if a certain container would be added to the particular instance of a DockerWorker.

DockerManager keeps a list of all containers and DockerWorker instances registered in the system. With each container, its estimated execution time and value curve are associated. DockerManager also includes information about all container allocations to DockerWorker instances. Its functionalities are communicated to an appropriate DockerWorker instance via its DockerAgent using a RESTful-based interface.

Each DockerWorker instance executes a DockerAgent and the original Docker daemon. DockerAgents communicate with the corresponding Docker daemon using its official Engine API. Currently, only three functions are available in DockerWorker: creation of a container, its start and stop. DockerAgent accesses (both for reading and writing) the Linux

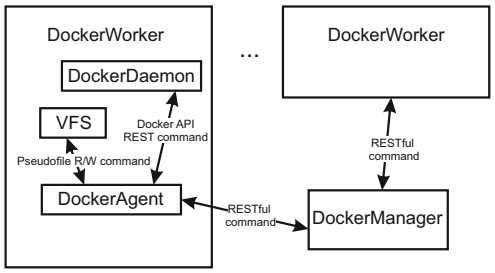

Fig. 2. General scheme of the proposed approach implementation

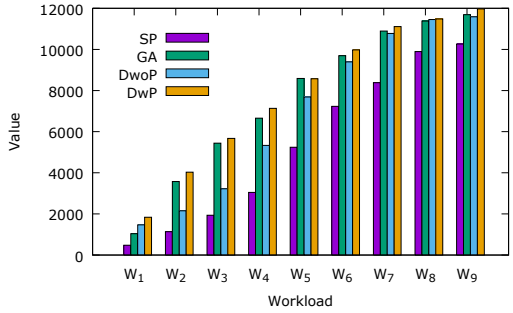

Fig. 3. Total value obtained with different allocation algorithms (SP - Spread, GA - Genetic Algorithm baseline, DwoP - Dynamic Allocation without Preemption, DwP - Dynamic Allocation with Preemption)

cgroups pseudo-files to get specific statistics and send certain properties of a container [11]. It allows DockerWorker to get or set a container CPU share, period and quota. There is also functionality for getting the total CPU usage of a particular container.

VI. Experimental Results

To experimentally evaluate the algorithms proposed in this paper, they will be first used to allocate workloads of assorted heaviness. Then the influence of various parameters of these algorithms is analysed to identify their most promising values for the considered scenarios. Finally, the scalability of the proposed solutions is addressed.

To check the system response to tasksets of various levels of load, nine sets of 10 random workloads, $W_1, \ldots, W_9$, have been generated. Each workload is comprised of 150 containers to be executed. The maximal value of each container equals 100. Execution times of these containers vary from 100s to 800s on a reference machine. There are three DockerWorkers, two of them can process with the speed of the reference machine whereas the third one is about 50% faster. The arrival time AT_{j+1} of the subsequent containers is selected randomly between AT_j and $AT_j + T_{max}$. The following T_{max} values (in seconds) have been selected for the workloads: W_1 - 10, W_2 - 45, W_3 - 80, W_4 - 115, W_5 - 150, W_6 - 185, W_7 - 220, W_8 - 255, W_9 - 290 to cover a wide spectrum of workload heaviness.

The total values obtained from the executed containers while using different algorithms are presented in Fig. 3. In all cases, the default Docker Swarm algorithm (SP - Spread) leads to significantly lower values than the remaining approaches. This observation is not surprising, as the Spread algorithm is not aware of the underlying containers' values and do not maximise the total value.

As a rather small CP is used for all workloads, for the heavier workloads (e.g. W_1) the total value is much lower than for the lighter ones (e.g. W_9), as there is not enough computational power and thus more containers are executed for much longer than their estimated execution time ET_j or rejected by the scheduler, as no positive value is predicted. For the heavier workloads, the variance of the values obtained with different algorithms is significantly higher than in case of the lighter ones. For example, for the heaviest workload W_1, the Dynamic Allocation with Preemption algorithm (DwP) is 3.85 times better than SP, whereas it is only about 17% better for the lightest workload, W_9. This behaviour has been expected as in the second case the cluster has almost enough resource to execute all the containers close to their estimated execution times. The different behaviour between W_1 and W_9 means that the workloads have been well selected to cover the spectrum of possible workload heaviness.

Not surprisingly, the knowledge of future container release leads to higher total values than the lack of such knowledge. Thus the value obtained with the static, baseline genetic algorithm is significantly better than Dynamic Allocation without Preemption. This difference is particularly visible for the workloads with heavy loads. For example, the value obtained with the Dynamic Allocation without Preemption algorithm (DwoP) is about 40% worse than the baseline for workloads W_2 and W_3, whereas the dynamic algorithm almost equated with the baseline for the lighter workloads.

The benefits resulting from applying preemption for the dynamic strategy are clearly visible. This approach extends the search space and can easily overperform the baseline approach especially for heavier workloads. Depending on the workload, DwoP improves the DwP result from 87% (for W_2) to less than 1% (for W_8). The arbitrarily selected value of the overload parameter $\lambda = 0.5$ and the chosen minimal remaining value victimisation criterion have appeared to be quite an appropriate choice, as discussed below.

In case of the heaviest workload W_1, the baseline static approach seems inferior to the proposed DwoP solutions. This counter-intuitive result is caused by the fact that the baseline genetic algorithm does not converge early in the majority of cases and finding the right termination criterion is problematic, especially when a workload is heavy.

According to the previous experiment, employing DwP leads to the highest values in all considered cases. However, earlier in this paper two techniques for victim selection have been mentioned: the lowest value density (LVD) and minimal remaining value (MRV). The results of the previous experiment have been obtained with the latter. Its comparison with the LVD criterion using the same workloads W_1-W_9 and cluster platform as before has been performed with no visible impact on the total value. LVD has been slightly better in total, but the difference (4 per mille) is negligible.

Next, different values of the λ parameter have been eval-

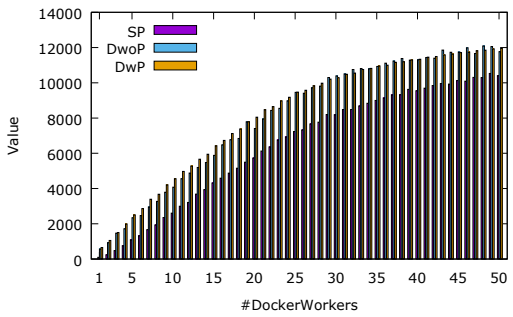

Fig. 4. Average values obtained for heavy workload set W_1 executed on different number of DockerWorkers (SP - Spread, DwoP - Dynamic Allocation without Preemption, DwP - Dynamic Allocation with Preemption)

uated. This parameter determines the situation if a node is overloaded according to equation (2). Workloads W_1-W_9 have been then allocated using the DwP algorithm with parameter λ ranging from 0.1 to 0.9. Low values of this parameter lead to about 10% worse results in terms of the total workload value. The choice of $\lambda = 0.5$ seems to be the most advantageous, especially in cases of workloads W_2, W_3 and W_4, for which the standard deviations of the total value with respect to parameter λ are significantly higher than for heavier or lighter workloads. For workloads of the heaviness similar to W_2-W_4 selecting the appropriate value of λ is then crucial.

Another parameter, applicable to both DwoP and DwP algorithms, is the CPU share stride size. For example, for the stride equal to 18, i.e. by analysing 57 CPU share values instead of 1023 per container per node, the resulting total workload value is from 25% (workload W_2) to 4% (workload W_8) worse than with the stride size equal 1, while the allocation time decreases about five times on average. In general, the stride size has a relatively low impact on the total value of lighter workloads (W_6-W_9) and then its larger value can be applied.

To evaluate the scalability of the proposed approaches, the heaviest workload set W_1 has been deployed to CP with the number of DockerWorkers ranging from 1 to 50. The values obtained in this experiment are presented in Fig. 4. An almost linear growth is observed for the three analysed approaches up to the point close to 20 DockerWorkers. After this point, the number of containers in W_1 per DockerWorker is too low to keep this linear trend and some saturation around the total value 12000 can be observed, what is 80% of the maximum possible value of the workload. In general, DwoP and DwP are 27% and 29% better than Spread, respectively. In the linear growth region (3 to 20 nodes) it is 53% and 65%, correspondingly. The proposed algorithms seem then to be particularly advantageous in the overloaded systems. Even for the largest analysed cluster (50 nodes), the average container allocation time is lower than 0.25s.

VII. Conclusion and Future Work

We have proposed a value optimizing container allocation approach for cluster platforms. We have shown that this approach is beneficial in cases when the container's value to end users varies over time. A baseline static and two dynamic allocation algorithms have been proposed. The dynamic algorithms have been implemented and used with the original Docker engine. The experiments have shown that the proposed approach can significantly increase the total value of a workload, especially in case of heavy workloads. The proposed approaches scale well with the growing number of nodes in a cloud.

In future, we plan to add metrics related to memory footprint size and network utilisation to improve the allocation of the memory or communication intensive containers. Additionally, we plan to investigate various container relocation algorithms to benefit from stateful container migration technologies, such as Flocker.

Acknowledgement

The research leading to these results has received funding from the the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 723634).

References

[1] Kubernetes Authors. Kubernetes, 2017.
[2] A.M. Burkimsher. *Fair, responsive scheduling of engineering workflows on computing grids*. PhD thesis, University of York, 2014.
[3] A. Burns et al. The meaning and role of value in scheduling flexible real-time systems. *Journal of systems architecture*, 46(4):305–325, 2000.
[4] M.T. Chung et al. Using Docker in high performance computing applications. In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 52–57, July 2016.
[5] Docker Contributors. Swarm: a Docker-native clustering system, 2014.
[6] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.
[7] L.S. Indrusiak et al. *Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing*. River Publishers, 2016.
[8] A. Jaison et al. Docker for optimization of Cassandra NoSQL deployments on node limited clusters. In *2016 International Conference on Emerging Technological Trends (ICETT)*, pages 1–6, Oct 2016.
[9] C. Kaewkasi et al. Improvement of container scheduling for docker using ant colony optimization. In *9th International Conference on Knowledge and Smart Technology (KST)*, pages 254–259. IEEE, 2017.
[10] B. Khemka et al. Utility functions and resource management in an oversubscribed heterogeneous computing environment. *IEEE Transactions on Computers*, 64(8):2394–2407, 2015.
[11] P. Ondrejka et al. Red Hat Enterprise Linux, Managing System Resources on Red Hat Enterprise Linux 6, Edition 6, 2016.
[12] M. Schwarzkopf et al. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, 2013.
[13] A.K. Singh, P. Dziurzanski, and L.S. Indrusiak. Value and energy optimizing dynamic resource allocation in many-core hpc systems. In *IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 180–185. IEEE, 2015.
[14] P. Sun et al. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6. IEEE, 2017.
[15] Y. Tao et al. Dynamic resource allocation algorithm for container-based service computing. In *IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*, pages 61–67. IEEE, 2017.
[16] T. Theocharides et al. Hardware-enabled dynamic resource allocation for manycore systems using bidding-based system feedback. *EURASIP Journal on Embedded Systems*, 2010:3, 2010.
[17] H. Topcuoglu et al. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar 2002.
[18] K. Ye and Y. Ji. Performance tuning and modeling for big data applications in Docker containers. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6, Aug 2017.
[19] M. Zhu and S. Martínez. *Distributed Optimization-Based Control of Multi-Agent Networks in Complex Environments*. Springer, 2015.