WattsApp: Power-Aware Container Scheduling

Hemant Kumar Mehta
University College Cork, Ireland
h.mehta@cs.ucc.ie

Paul Harvey
Rakuten Mobile, Japan
paul.harvey@rakuten.com

Omer Rana
Cardiff University, UK
ranaof@cardiff.ac.uk

Rajkumar Buyya
University of Melbourne, Australia
rbuyya@unimelb.edu.au

Blesson Varghese
Queen's University Belfast, UK
b.varghese@qub.ac.uk

ABSTRACT

Containers are popular for deploying workloads. However, there are limited software-based methods (hardware-based methods are expensive) for obtaining the power consumed by containers to facilitate power-aware container scheduling. This paper presents `WattsApp`, a tool underpinned by a six step software-based method for power-aware container scheduling to minimize power cap violations on a server. The proposed method relies on a neural network-based power estimation model and a power capped container scheduling technique. Experimental studies are pursued in a lab-based environment on 10 benchmarks on Intel and ARM processors. The results highlight that power estimation has negligible overheads - nearly 90% of all data samples can be estimated with less than a 10% error, and the Mean Absolute Percentage Error (MAPE) is less than 6%. The power-aware scheduling of `WattsApp` is more effective than Intel's Running Power Average Limit (RAPL) based power capping as it does not degrade the performance of all running containers.

ACM Reference Format:
Hemant Kumar Mehta, Paul Harvey, Omer Rana, Rajkumar Buyya, and Blesson Varghese. 2020. WattsApp: Power-Aware Container Scheduling. In *UCC '20: IEEE/ACM International Conference on Utility and Cloud Computing, December 07–10, 2020, Leicester, UK.* ACM, New York, NY, USA, 10 pages.
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Container technology is a lightweight virtualization technique that has low overheads [19]. Therefore, it is becoming popular for deploying workloads on clusters and clouds [20, 24] and for upcoming edge computing systems [22, 23]. Container scheduling is an important avenue explored in this context. Existing container scheduling strategies consider resource demand, service level agreements and hardware/software requirements [11, 26]. However, container scheduling needs to be power-aware so that the total power consumption of a system does not exceed predefined power cap limits.

UCC '20, December 07–10, 2020, Leicester, UK

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Modern processors are equipped with power capping techniques, such as Dynamic Voltage and Frequency Scaling (DVFS) and Running Average Power Limit (RAPL) [13, 14, 25]. These are hardware-based and reduce the CPU frequency and voltage to lower processor power consumption. However, this degrades the entire system performance and consequently the deployed application.

It is valuable to gather the power consumption of individual containers running in a system to schedule them in a power-aware manner. However, software-based methods, such as cWatts [17], cWatts++ [18] and SmartWatts [7] are either CPU architecture specific, do not capture all system components that contribute to container power, and are intrusive methods. This paper develops `WattsApp` underpinned by a six step software-based (hardware-based are expensive and require hardware level changes), architecture agnostic and a non-intrusive power-aware container scheduling method. The aim is to estimate container power consumption and schedule containers to stay within safe power budgets.

The contributions of `WattsApp` are as follows:

(i) A six-step software-based power-aware container scheduling method with negligible overheads that accurately predicts container power consumption. Additionally, nearly 90% of all data samples can be estimated using the power model with less than a 10% error. The Mean Absolute Percentage Error (MAPE) is observed to be between 1%-6%, which is relatively low. `WattsApp` is the first prototype that builds power models and enforces power capping for parallel applications that execute on a cluster of containers.

(ii) A power capped scheduling approach for single and multiple containers which is more beneficial than when no power cap or Intel's RAPL power cap is employed (the performance of all running containers on the system is not degraded, only containers that violate the budget are penalized). The power cap is achieved by migrating the container to another server or deallocating resources of the container that violates the power cap.

This paper is organized as follows. Section 2 discusses the motivation for `WattsApp`. Section 3 proposes the `WattsApp` method and presents the underlying power model and the power capped container scheduling approach. Section 4 presents experimental studies. Section 5 discusses the related work. Section 6 concludes the paper.

2 BACKGROUND

Predicting container power is complex because it depends on the resource allocated and the workload running in the container. It is different to the power prediction of VMs, other processes and hardware (processors, memory etc) because of the limited availability of resource utilization data and hardware performance counters specific to containers. This is because containers create multiple

Table 1: Scientific workloads used in this paper

Name	Description	Type
KMEANS	Clustering algorithm	DCBench
FUZZY-KMEANS	Clustering algorithm	DCBench
KPCA	Principal component analysis	DCBench
PCA	Principal component analysis	DCBench
BFS	Graph mining breadth-first	DCBench
MD	Molecular dynamics	MPI-C
HEATED	Steady heat equation solver	MPI-C
POISSON	Poisson equation solver	MPI-C
PRIME	Counting prime	MPI-C
SGEFA	Linear algebra solver	MPI-C

processes on the host operating system. The number of processes varies depending on the activity within the containers.

This paper observes that containers with more allocated resources consume more power for the same workload than on containers with fewer resources. However, in all cases the power does not correspond to the increase in resources (CPU cores, memory). For example, the average power of an application running on twice the resources, may not necessarily directly correspond to a factor of two.

This hypothesis is verified on 10 different scientific workloads that are listed in Table 1. These workloads are obtained from: (i) DCBench [10] from which five MPI based workloads are chosen, and (ii) A collection of C/C++ based scientific programs[1] that are a combination of CPU-bound, I/O-bound and memory-bound scientific workloads. This paper does not consider alternate classes of workloads, such as Internet-of-Things, stream processing, or sensor-based applications. The workloads considered in this paper may have different power consumption patterns during execution. This is captured in the resource utilization and power data that is collected at a fine granularity and used for building the power model. This ensures that estimation is carried out for different workload phases.

Figure 1 highlights the average, minimum and maximum power consumed by a container with 3 CPU cores and 4GB RAM. Figure 2 provides results for the same workloads for twice the resources (6 CPU cores and 8GB RAM) Although the resources allocated are doubled the average power consumed does not necessarily double for all workloads (for example, refer to the workloads HEATED, MD, POISSON, PRIME and SGEFA).

Similar trends are obtained when the power consumption is noted for the above workloads over time (the results are exhaustive and are not presented in this paper). When more resources are added parallel workloads (applications running within a single container) execute faster, but reach their peak power consumption at different times. This paper does not aim to explain individual power profiles of workloads, but notes that a server that executes multiple containers could violate the power cap; specifically, when multiple large size containers are multi-tenant on a server. These large containers may consume high power and when they are multi-tenant their combined total power consumption could be higher and close to the maximum power consumption of the server.

Power cap violations are undesirable and need to be effectively managed on servers running different workloads. They occur when the total power consumed by a server exceeds a threshold defined by the server administrators. When power cap violations occur, the

[1] http://people.sc.fsu.edu/~jburkardt/c_src/c_src.html

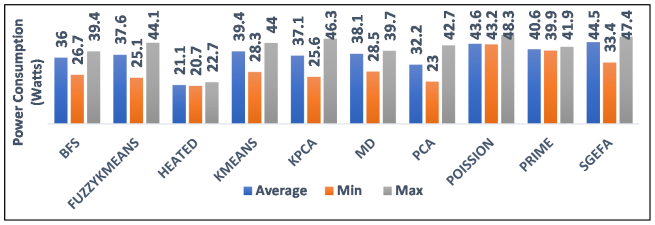


Figure 1: Power consumption (Watts) of workloads deployed in containers with 3 CPU Cores and 4GB RAM.

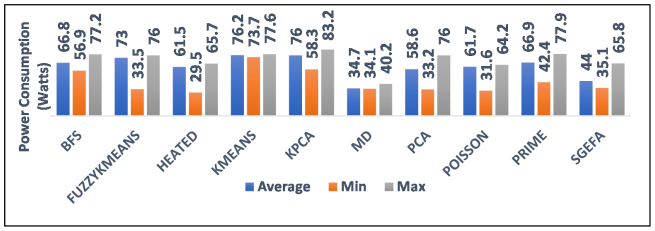


Figure 2: Power consumption (Watts) of workloads deployed in containers with 6 CPU Cores and 8GB RAM.

server performance starts degrading since power management techniques like Dynamic Voltage and Frequency Scaling (DVFS) that are bundled with processors come in to play. DVFS reduces the server power consumption by using two power saving techniques, namely dynamic voltage scaling and dynamic frequency scaling [14]. Power is saved by lowering the frequency/voltage of the CPU and other system resources. This reduction negatively impacts performance of workloads executed on the server (performance of a container running on a server will drop when there is a power cap violation).

A power aware container scheduling strategy is required to avoid the above. Power/energy saving benefits of techniques like DVFS are diminishing because of the complexity of multi-core processors and memory [14]. Therefore, a software-based power capping technique is desirable in addition to specific hardware-based techniques.

3 THE WATTSAPP METHOD

This section presents a method for software-based power aware scheduling of containers to minimize power cap violations on a server, which is fundamental in developing `WattsApp`. *Power aware container scheduling* is the distribution/consolidation of containers such that the total power consumed by a server does not cross a predefined threshold (or cap) specified by an administrator.

A primary requirement for the `WattsApp` scheduling approach is to obtain information on the power consumed by a container. Resource utilization statistics of each container is used to calculate its power consumption. This information is used for container scheduling, such that the maximum power consumed does not violate any power restriction on an individual server.

Currently, there are no hardware methods for obtaining the power consumed by containers. Moreover, there are a few software methods to measure the container power consumption and these methods have concerns like they are architecture specific, ignores essential system resources or intrusive as discussed in Section 5. This article

aims to bridge this gap. Hardware-based methods will require modification of the hardware (such as additional probes) resulting in more expensive processors. Hence, a *software-based method* is adopted to develop a model of container power consumption that depends on resource utilization information of the container. The model uses linear regression-based neural network to correlate container resource utilization statistics with system power consumption information to estimate the container power consumption (refer Section 3.2).

The WattsApp method relies on two activities: (i) *Container power prediction*, which is estimating the power consumed by an individual container using software-based methods, and (ii) *Power capped container scheduling*, which is using the estimated power values to place containers equitably on a server.

Any power cap violation is periodically detected on each server by observing the total power consumption of the server. If the power consumed is above a threshold, then power capping is performed on the server without significantly affecting the run time performance of all running containers. Power capping in WattsApp is performed using two approaches, namely container migration and container resource (CPU cores) reduction (considered in Section 3.2). The method firstly attempts to migrate a container causing the server power cap violation onto another server. If no servers are available to migrate a container, then the resources allocated to the container are reduced (the processor consumes nearly 85% power of the total system power [16]). Hence, the CPU cores allocated to the container are firstly reduced one at a time until the power cap limit is restored. Commands, such as *docker update*, are used to change the cores allocated to a container; the change is immediately reflected.

The proposed method of WattsApp shown in Figure 3 comprises six steps: namely Data Collection, Model Building, Power Estimation, Power Capped Allocation, Violation Detection and Power Capping. The first three steps are for container power prediction, and the remaining steps are for power capped container scheduling.

Step 1 - Data Collection: The training data for supervised learning is collected to estimate container power consumption. It collects system power consumption and resource usage statistics for each container from the host OS. This is correlated with system power consumption to obtain container power consumption. The system power information is necessary as the regression techniques require labeled data for building a model.

System power consumption data is collected from Watts Up .net hardware power meter. It facilitates the power consumption sampling at a one second granularity. Real time system power consumption data can be obtained by connecting it through a USB interface. It is reported that the accuracy of this hardware meter is +/- 1.5%+0.3W[2] [9]. The readings provided are also considered to be generally reliable although a high error rate is observed for readings below 1 Watt [9]. A series of power meters from Watts Up are used in research reported in the literature [15][1][4][3].

Step 2 - Model Building: The collected data is used to train data and build a neural network for individual containers running on the server. A model is built for each container as different applications exhibit different properties. The models are used during run time for predicting the power consumed by the container. The input to the neural network is container resource utilization statistics (including

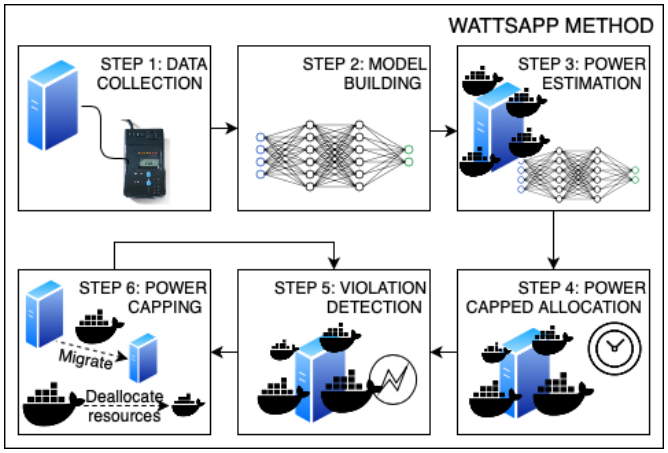


Figure 3: WattsApp **method for power-aware scheduling.**

the host CPU and memory the container is using, the total memory used by the container and the maximum allocated memory, the data the container sent/received on its network interface, and the amount of the data read from/written to the block I/O devices) and system power consumption. The model was developed using Keras[3].

Step 3 - Power Estimation: Consider there are n containers (C_1, C_2, C_3, $\cdots$ C_n) running on a server. The resource utilization statistics (CPU usage, memory usage, amount of block I/O and network data transfer) are collected for each container. The models developed (previous step) and the run-time statistics are used to predict the power consumed by the container.

Step 4 - Power Capped Allocation: The models are used for power capped container scheduling. It uses the predicted power consumption of the container and current total power consumption of the server before scheduling the container on the server. The power consumption information of the containers are obtained by using the power estimation model that uses the power profile of each container for power capped allocation.

Step 5 - Violation Detection: Power capped container scheduling is performed when a power cap violation is detected. After initial scheduling, this step is executed at a five minute interval to check the server for any power cap violation. If there is a violation, then the final step enforces the power capping limit on the server.

Step 6 - Power Capping: This final step adopts two techniques to enforce power capping. The first is migration - another server that can accommodate the container causing the power cap violation on a current server is identified; migration should not violate the power cap of the recipient server. If such a server is available, then the power cap violating container is migrated to the identified server. If no such server is identified, then a second technique, referred to as resource deallocation, is performed in which the number of CPU cores allocated to the container is reduced until the power cap limit is reached. This first prototype of WattsApp only considers a single container causing power cap violations. However, if multiple containers cause power cap violation, then a priority based container selection approach is required (not considered in this paper).

[2] https://www.vernier.com/files/manuals/wu-pro.pdf

[3] https://keras.io/

Algorithm 1: Data Collection

Input : Container name
Output : Combined resource utilization and power consumption data

1 **while** *workload in container is running* **do**
2 Obtain resource utilization from docker stat command once per second
3 Add time stamp to the output of docker stat
4 Collect Watts Up power data once per second
5 **end**
6 Combine resource utilization and power data on the basis of the timestamp

The first three steps are presented in Section 3.1 and the last three steps are discussed in Section 3.2.

3.1 WattsApp Power Estimation

This section describes Data Collection (Step 1), Model Building (Step 2), and Power Estimation (Step 3).

The ***Data Collection*** step gathers (i) the system power consumption data, and (ii) resource utilization data of running containers. This is a black box technique as the data is collected from the host operating system and no profiling data is obtained from within the container. The other approaches (referred to as white box) collect profiling information inside the container and should be avoided to maintain the integrity of the containers [8].

Power consumption data is collected from the Watts Up .net power meter. The data obtained contains the time stamp and power consumption (in Watts). The resource utilization data of the container is collected using the `docker stats` command, which provides the following data: (i) Id of the container and the name of the container, (ii) Percentage of host CPU and memory the container is using, (iii) Total memory the container is using and the maximum allotted memory, (iv) The amount of data the container has sent and received on its network interface, (v) The amount of the data read from and written to the block input/output devices, and (vi) The number of processes/threads created by container.

Resource utilization and system power consumption data are obtained once per second during benchmark execution. The sequence of steps is presented in Algorithm 1.

In the ***Model Building*** step, a neural network is used that takes as input the container resource utilization (including percentage of host CPU and memory the container is using, total memory the container is using, maximum allotted memory, amount of data the container has sent and received on its network interface, amount of the data read from and written to the block I/O devices), and system power consumption. The output is container power consumption.

In the ***Power Estimation*** step, the power consumption of containers is firstly modeled by `WattsApp`. The power consumption of a server (P_{server}) comprises static power and dynamic power. Static power (P_{static}) is defined as the power consumption of system when there is no active container. This power is measured by the Watts Up .net power meter. If there is only one running container then the total power consumption is sum of idle power and dynamic power consumption of the server. The dynamic power consumption ($P_{dynamic}$) is defined as the power consumption of the running container.

$$P_{server} = P_{static} + P_{dynamic} \tag{1}$$

If there are n active containers on a server, then the dynamic power consumption is the aggregate power of all the containers.

$$P_{dynamic} = \sum_{k=1}^{n} P_{container_k} \tag{2}$$

where $P_{container}$ is the power consumption of the container.

The dynamic power consumption of the system is the sum of the power consumed by the CPU, memory, disk and network.

$$P_{container_k} = a_k * Ucpu_k + b_k * Uram_k + c_k * Udisk_k + d_k * Unet_k \tag{3}$$

where $a, b, c,$ and d are constants, n is the number of running containers, l represents the number of CPU cores allocated, $Ucpu$ is the CPU utilization factor ($Ucpu = \sum_{i=1}^{l} Ucpu_{core_i}$), $Uram$ is the RAM utilization factor, $Udisk$ is the disk utilization factor, and $Unet$ is the network utilization factor.

$$P_{server} = P_{static} + \sum_{k=1}^{n} a_k * Ucpu_k + \sum_{k=1}^{n} b_k * Uram_k + \sum_{k=1}^{n} c_k * Udisk_k + \sum_{k=1}^{n} d_k * Unet_k \tag{4}$$

A single workload can be executed across a cluster of containers. In this case, the workload's power consumption is the aggregate power consumption of all containers of the cluster.

$$P_{workload} = \sum_{i=1}^{n} P_{Container_k} \tag{5}$$

where $Container_k$ is element of C, the set of containers in the cluster $C = \{Container_1, Container_2, \cdots Container_n\}$.

3.2 WattsApp Power Capped Scheduling

This section presents the use of the estimated power values at run time for the power capped container scheduling method. The approach is to initially allocate containers using a best fit strategy, and when there is a power cap violation on the server, migrate the container elsewhere or reduce the allocated CPU cores of the running containers until the power cap is not violated. Power Capped Allocation (Step 4), Violation Detection (Step 5), and Power Capping (Step 6) from Section 3 is considered in this section.

Power Capped Allocation uses the estimated power to schedule containers by calculating the total power consumption of the candidate server after adding the estimated power of the container ready for deployment. If the power consumed by the server is anticipated to be below the power cap, then the container is placed on the candidate server. This is repeated for all containers that are ready for placement. Algorithm 2 highlights this and Table 2 presents the notation used by the algorithms presented in this section. It is assumed that there are n servers, and each server may have up to m containers.

Algorithm 2 executes for all containers ready for placement (line 1). The flag variable is initialized to false (line 2); this variable

Algorithm 2: Power Capped Scheduling of Containers

Input : S, C_{i_j}, md

1 **for** $\forall\ c_i\ \varepsilon\ C$ **do**
2 $flag = false$
3 **for** $\forall\ s_i\ \varepsilon\ S_i$ **do**
4 $PS_i = PS_i + containerPower_i$
5 **if** $serverPower_i < cap$ **then**
6 allocate(c_i, s_i)
7 $flag = true$
8 **else**
9 do nothing
10 **end**
11 **end**
12 **if** $flag == false$ **then**
13 Power Capped allocation is not possible
14 Select the i^{th} server with minimum PS_i to place the current container
15 **end**
16 **end**

Algorithm 3: Power Cap Violation Detection

Input : S, C_{i_j}, md

1 **for** $\forall\ s_i\ \varepsilon\ S$ **do**
2 $totalPower_i = 0$
3 **for** $\forall\ c_j\ \varepsilon\ C_{i_j}$ **do**
4 data = collect_stats(c_j)
5 PC_{i_j} = md.predict(data)
6 $totalPower_i = totalPower_i + PC_{i_j}$
7 **end**
8 **end**
9 **for** $\forall\ s_i\ \varepsilon\ S_i$ **do**
10 **if** $totalPower_i > cap$ **then**
11 powerCap(c_j, ps_j,s_i, PS)
12 **else**
13 do nothing
14 **end**
15 **end**

Table 2: Notation used in the Power Capping Method

Notation	Description
S	Set of all the servers $s_i\ \varepsilon$ S for $i = 1, \cdots, n$
Cr_i	Number of CPU Cores in server s_i
M_i	Available memory in the server s_i
C_{i_j}	List of m containers deployed in server S_i
C	List of all the containers ready to be placed
ACr_i	List of CPU core allocated to containers
AM_i	List of memory allocation to containers
md	Power consumption model
PC_{i_j}	Power consumption of container c_{i_j}
PS_i	total power consumption of server s_i, $ps_i\ \varepsilon$ PS
c_j, pc_j	candidate container causing server power cap violation and its power consumption
cap	Power cap

will be used to identify the case when no suitable server for power capped placement. Each server is checked one by one whether it can accommodate the container under consideration (line 3). The container power consumption is added to the candidate server power (line 4) to check if it can accommodate the container (line 5). If the candidate server can deploy the container, then it is allocated to the server (line 6). When container placement is successful, the flag is updated to true (line 7). If it is not possible to allocate on a given server, then the remaining servers are processed. When no suitable server is found for power capped placement (line 12), the container is allocated to the server with the lowest power consumption (line 13 and 14). Algorithm 3 detects the possibility of power cap violation, and if required, the power cap is applied using Algorithm 4.

A process to determine any power cap violation is executed on the servers, five minutes after initially scheduling containers (profiling data is collected for the first five minutes). This process uses the power prediction model to estimate the power consumption of each running container. If the power consumption of any server (sum of power consumed by all running containers) is beyond the power cap, then there is a violation due to the newly placed container). This container is considered for migration or core reduction.

Power capping is achieved in two ways. The first is by *migrating the candidate container* from the source to a destination server whose power consumption is below the cap. A stateful migration method, namely 'CRIU (Check-point/Restore In Userspace)' is employed for migrating containers. The second is by reducing the resources allocated to the candidate container, specifically the number of CPU cores (reduce one at a time) as CPU usage significantly affects the power consumption of Docker containers [21]. The Docker update command is used to change the number of allotted CPU cores to the container. The performance of the container will be degraded when using this approach (further considered in Section 4).

Currently, there is support for power capping on the hardware. However, most hardware power capping techniques tweak the voltage and processor frequency, which affects the potential performance of the entire system and is detrimental to **all** containers running on the system [14]. However, the proposed software power capping technique achieves the power cap without significantly affecting the entire system's performance and only negatively impacts the container that causes the power cap violation.

Violation Detection detects a power cap violation and runs Algorithm 3 on each server. When a violation is detected, Algorithm 4 performs power capping. The model used to compute the power consumption of the container at run-time is considered in Section 3.1.

The detection algorithm firstly computes the power consumption of each server (line 1) indirectly by calculating the power consumption of every container (line 3) deployed on the server. This is achieved by collecting the resource usage statistics for each container (line 4) and then passing the data to the power model for predicting power consumed (line 5). The power consumption of all containers running on a server are summed to obtain server power consumption (line 6) after which power cap violation (if any) is checked for on each individual server (line 9 and 10). If a server crosses the power cap limit then Algorithm 4 is performed with the required inputs. If

Algorithm 4: Apply Power Capping

Input : $S, C_{i_j}, c_j, ACr_i, cap, PS, md$
Output : true if power capping is successful, false otherwise

```
1  for ∀ s_j ε (S-s_i) do
2      if PS_j + pc_j < cap then
3          migrate(c_j, s_j)
4          return true
5      else
6  end
7  while ps_j > cap do
8      reduceCoresByOne(c_j)
9      data = collect_stats(c_j)
10     predictedpower=md.predict(data)
11     if ps_j < cap then
12         return true
13     end
14 end
15 return false
```

the power consumed by all containers on the servers is below power cap, then no changes are made (line 13).

The last step is ***Power Capping*** that uses Algorithm 4 (input provided by Algorithm 3). The container that violates the power cap on a source server is migrated to a destination server (whether such a server is available is checked by a network process; lines 1-3). If no candidate destination servers are found, the algorithm reduces the cores allocated to a container until the server power consumption drops below the power cap (line 7). The cores are first reduced one by one (line 8) and then the power consumption is predicted (line 9, 10). Again, if the power consumption does not fall below the power cap (line 11), then the above process is repeated.

CPU core reduction to achieve power capping will degrade performance of the selected container. This can be compensated for by increasing the CPU cores at a later stage when it may be feasible to do so without exceeding the power cap. This scenario is considered in experimental studies to demonstrate that the impact of CPU core reduction of a container can be compensated when running the parallel component of an application in a cluster.

There may be a delay in enforcing the power capping limit since the detection algorithm is only executed every five minutes (this is a configurable parameter; experimentation on the impact of this limit is not presented in this article). It was empirically observed that after a violation was detected, time was required for migrating the container to another server due to: (i) Creating checkpoints - freeze a running container and save its state on disk, (ii) Compressing the checkpoint and transferring it to the selected server, and (iii) Creating a new container and restoring the checkpoint. The majority of the migration time (depends on the container image size) is for transferring the checkpoint to the destination server.

4 PERFORMANCE EVALUATION

The experiments highlight that for WattsApp: (i) the overheads for data collection to estimate power do not significantly impact system power consumption, (ii) there is limited error in estimating power using the model, (iii) WattsApp operates across multiple processors, and (iv) the proposed power capping method is more effective for scheduling than alternate methods, such as Intel's RAPL.

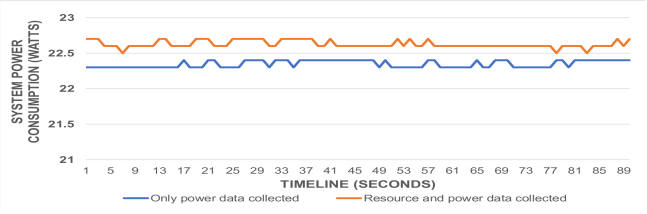


Figure 4: Data collection overhead for 89 seconds on Intel Xeon.

The experiments are carried out on two systems with different form factors. The first is workstation Dell Precision 3630 with an Intel Xeon E-2174G processor and 16GB memory. The second is a small form factor Odroid N2 Board with a quad-core ARM Cortex-A73 CPU cluster and a dual core Cortex-A53 cluster and 4GB memory. The system power data is collected using Watts Up .net hardware power meter. The systems run Ubuntu 18.04. The containers are created with Ubuntu 18.04 LTS image. Each container is allocated three CPU cores, 4GB memory on workstation and 2 CPU cores and 2GB memory on Odroid. Docker 17.12.0-ce version is used to deploy the containers. Keras 2.2.0 that runs on TensorFlow 1.8.0 is used to build the power model. The hardware power meter Watts Up .net is used to obtain system power consumption in real-time. Watts Up .net power is used to collect instantaneous power readings using USB from the host OS.

The workloads defined in Table 1 are used for evaluating WattsApp. These are scientific workloads that execute to completion. This paper does not consider Internet-of-Things, stream processing, or sensor-based applications. The experiments are carried out for single workloads on single containers, multiple workloads on multiple containers, and single workload across multiple containers (a cluster of containers) to thoroughly evaluate the WattsApp method.

Results: The data collection overheads and estimation error in container power prediction is firstly presented. Then, the results from scheduling for power capping obtained for single and multiple containers are considered. The average system power consumption overhead when collecting data (CPU, memory, and disk utilization along with power) for a 89 second time period is shown in Figure 4. The blue plot shows the average system power consumption when only power data is collected, and the orange plot shows when both resource utilization and power metrics are collected. On an average, nearly 0.2 Watts are spent. The graph illustrates that the overhead in terms of system power is negligible (less than 1% of system power consumption). This is an indicator that Step 1 of the proposed power-aware scheduling method is feasible. Figure 5 shows the data collection overhead on ARM to be nearly 1.7% of the system power.

Figure 6 shows the error distribution on the Intel processor in the power estimated for 444 samples using the neural network model. In this experiment, data collected from all workloads (Table 1) is consolidated to build the model that is validated using repeated random sampling by splitting data into 75% and 25% as training and testing dataset respectively. More than 90% of the samples have an error of less than 10% and nearly 49% of the samples have less than

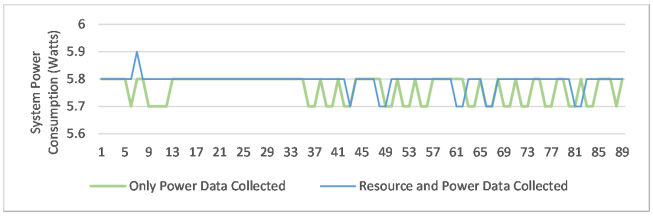


Figure 5: Data collection overhead for 89 seconds on ARM.

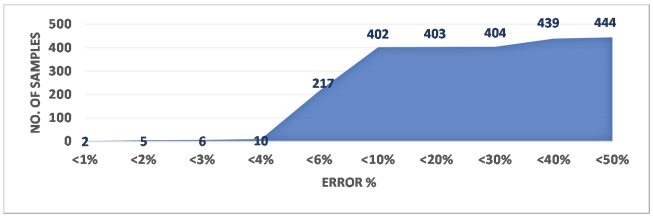


Figure 6: Error distribution for 444 data samples on Intel Xeon.

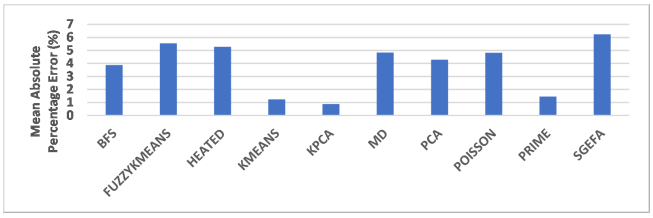


Figure 7: Mean Absolute Percentage Error (MAPE) in estimating system power for workloads running in containers on Xeon.

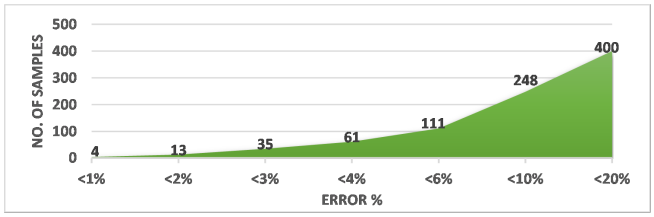


Figure 8: Error distribution for 400 data samples on ARM.

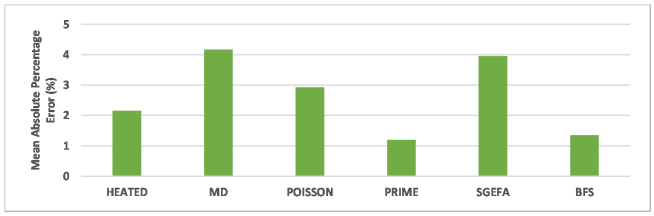


Figure 9: Mean Absolute Percentage Error (MAPE) in estimating system power for workloads running in containers on ARM.

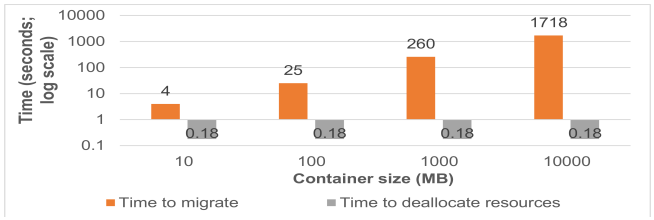


Figure 10: Time taken to migrate containers and deallocate resources for different sizes of containers.

a 6% error. This highlights that the power model built in Step 2 of the method will have a reasonable accuracy for prediction in Step 3.

Figure 7 highlights the Mean Absolute Percentage Error (MAPE) of the model for estimating power of individual workloads executing in a container on Intel Xeon. For this experiment each workload is executed in a single container and the power is estimated for each container. MAPE indicates the average of percentage errors (a lower value means the model estimates with a higher accuracy). The average percentage error is between 1% and just over 6%.

Similar experiments are performed on the ARM processor using six workloads from (Table 1) (FUZZYKMEANS, KMEANS, KPCA and PCA are from DCBench with the binaries for the x86 platform). Figure 8 shows the error distribution of estimating power values for 400 samples using the neural network model. In this experiment, data collected from the six workloads is consolidated to build the model that is validated using repeated random sampling by splitting data into 75% and 25% as training and testing dataset, respectively. Over 80% of the samples have an error of less than 15% and over 60% of the samples have less than a 10% error. This highlights that the power model will have a reasonable prediction accuracy.

Figure 9 highlights the MAPE of the model for estimating power of individual workloads executing in a container on the ARM processor. For this experiment each workload is executed in a single container and the power is estimated for each container. MAPE indicates the average of percentage errors (a lower value indicates that the model estimates the power consumed with a higher accuracy). The average percentage error is between 1% and just over 4%.

Figure 10 shows the overheads associated with the two techniques adopted in power capping, namely migration and deallocation of resources for different sizes of containers. The time taken to migrate using the Checkpoint/Restore in Userspace approach provided by Docker is directly proportional to the size of the container as the container needs to be checkpointed and migrated to an alternate server. However, using the time taken to deallocate resources on a container on the server takes approximately 180 milliseconds. Although migration is a potential option to achieve the power cap, the results show that deallocating resources is a more viable option given the inherent overheads in container migration. In the next set of experiments, power capping results based on only resource deallocation is presented. Migration using containers is a less viable option based on existing technology (if a critical application has to be executed) given large migration overheads although it may be lower than VMs (also not suited for single parallel application executed across a cluster of containers).

In another experiment, a cluster of containers (four on Intel and two on ARM) was created for running the MPI applications from

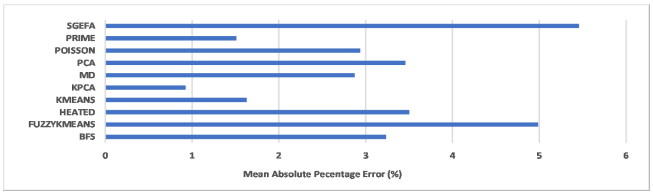


Figure 11: MAPE of estimating system power for parallel workloads running across a cluster of four containers on Intel Xeon.

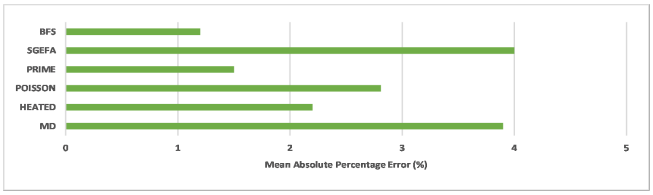
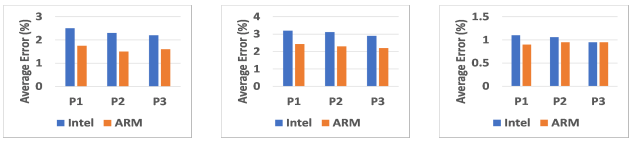
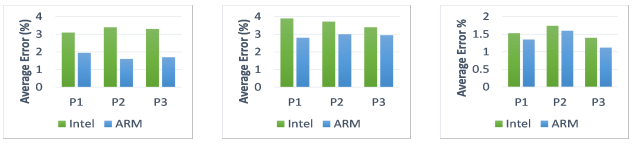


Figure 12: MAPE of estimating system power for parallel workloads running across a cluster of two containers on ARM.



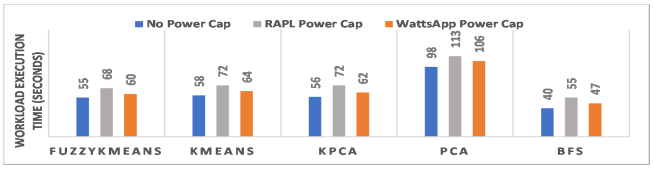
(a) BFS (b) POISSON (c) MD

Figure 13: Average power estimation error (%) for different input configurations of benchmarks on a single container.

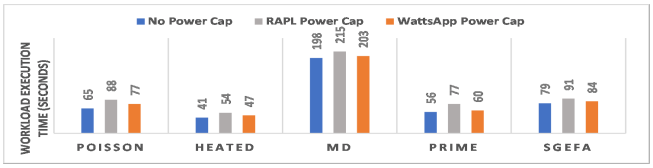


(a) BFS (b) POISSON (c) MD

Figure 14: Average power estimation error (%) for different input configurations of benchmarks across two containers.



(a) For DCBench programs from Table 1



(b) For MPI-C programs from Table 1

Figure 15: Execution time when a single container executes using the power cap technique on Intel Xeon.

Table 1. Figure 11 shows the results on Intel to demonstrate the feasibility of container power prediction for parallel applications executed across multiple containers. The average MAPE is 3 with error between 1 % and around 5.5%. Figure 12 show the MAPE on ARM. The average MAPE is 2.6 with error between 1 % and 4 %.

The accuracy of WattsApp power estimation is considered for different input parameters when the benchmark is executed in a single container (Figure 13) and in cluster of two containers (Figure 14). Only three benchmarks (BFS, POISSON, MD) are presented with three different input parameters (P1, P2 and P3). The input to BFS is the parameter scale for which P1, P2, and P3 values are 8, 12, and 16 respectively. POISSON takes as input the number of interior vertices in one dimension, for which we chose P1, P2, and P3 as 16, 32, and 64 respectively. MD requires parameters: spatial dimension, number of particles, number of time steps and time step size; P1 = {2, 500, 500, 0.2}, P2 = {3, 500, 500, 0.2}, and P3 = {3, 750, 500, 0.2}. The data for the input parameters were not used during training. The results highlight that the average error percentage is between 0.8% and 4% for both Intel and ARM processors.

Two further experiments were carried out on Intel Xeon to identify benefits of the scheduling approach when compared against no power caps or Intel's RAPL. The first experiment is when a single container executes on the server with a given workload; only one container running on the server is likely to violate the power cap. The second experiment is when multiple (three) containers that run the same workload execute on the server; any container may violate the power cap. All containers executes the same workload.

Figure 15 shows the results for the first experiment in which a single container executes on the server with a given workload. The graph shows the workload execution time for the proposed power cap method, no power cap, and RAPL's power cap is adopted. The proposed power capping method is more effective since the total workload execution time is lower than RAPL's power cap.

Figure 16 shows the peak power consumption on the Intel Xeon processor of applications (from Table 1) when there is no power capping, under the WattsApp power capping regime and the RAPL power capping technique. This experiment uses the power cap limit of 55W. The average peak power consumption of the proposed power capping technique is 56.4W which is close to the power cap limit where as the average peak power of RAPL's power cap is 60.2W and significantly higher than the power capping limit. The peak power consumption for WattsApp is 60.3W in comparison to the peak power consumption of RAPL's power cap is 65.9W.

Similar experiments are carried out on ARM using six workloads from Table 1. As RAPL is specific to Intel, these experiments only compare the WattsApp power cap with no power cap.

Figure 17 shows the results when a single container executes on the ARM processor. The workloads running under WattsApp takes slightly longer time and executes within the power budget of 7W.

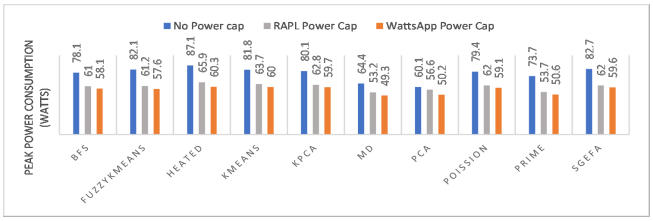


Figure 16: Peak power under power cap techniques on Xeon.

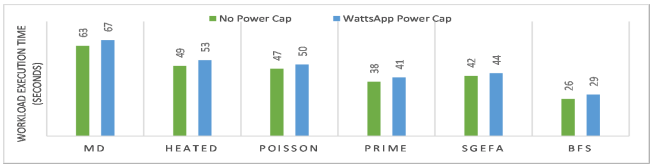


Figure 17: Execution time when a single container executes using the power cap technique on ARM.

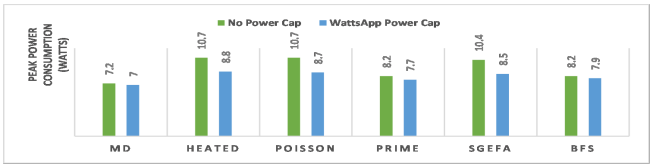


Figure 18: Peak power under power cap techniques on ARM.

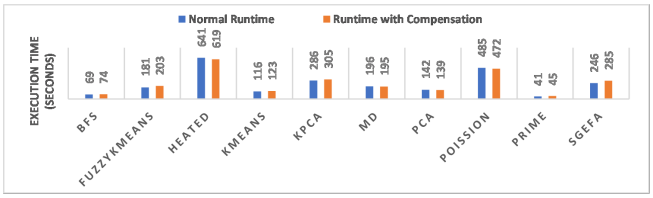


Figure 19: Effect of power capping with compensation on parallel workloads on Xeon.

Figure 18 shows the peak power consumption of six workloads when there is no power capping and under the WattsApp power capping regime. A power cap limit of 9W is used. The average peak power of the proposed power capping technique is 8.1W (below the power cap limit), but is 9.2W when there is no power cap.

Figure 19 and Figure 20 show performance gain when a single MPI application is executed across a cluster of containers. This experiment considers that workloads are running on a cluster of two or more different servers. When the CPU cores of the workload need to be reduced on one server, then it is compensated for by increasing the CPU cores allocated on the other server for the workload. Power capping on one server with compensatory allocation on another server does not significantly impact performance.

Summary: The experimental results highlight that: (i) The data collection overhead in the proposed power-aware container scheduling method of WattsApp only affects the system power consumption negligibly. (ii) Nearly 90% of data samples are estimated with

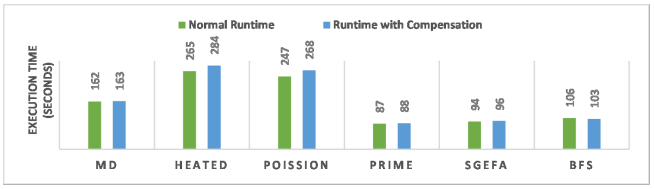


Figure 20: Effect of power capping with compensation on parallel workloads on ARM.

less than 10% error. (iii) The MAPE of power estimation using the model that is employed by the proposed power-aware container scheduling method of WattsApp is between 1%-6%. This is relatively low and accurate estimations can be expected from the model. (iv) The power estimation method of WattsApp is also validated on the parallel applications across a cluster of containers. The proposed model estimate the power consumption with MAPE between 1% to 5.5%. The impact of power capping (CPU core reduction) for parallel workloads is minimized during the workload runtime by applying compensation. (v) Deallocation of resources are found to be a more feasible approach than migrating containers for the power capping technique given that the overheads for migration increase with the size of the container. The overheads for deallocating resources is negligible. (vi) When both single and multiple containers are executed, the proposed power cap method is more beneficial than when no power cap or RAPL's power cap is employed since the proposed method does not degrade the performance of all running containers.WattsApp's power capping is also effective since the peak power allowed by WattsApp is less than RAPL's power cap and does not violate any soft power cap imposed by administrators.

5 RELATED WORK

Power modeling of processors and VMs are well explored, but power modeling of containers is in early stages.

Container Power Modeling: SmartWatts [7] is a self calibrating software power model for containers that relies on hardware performance counters and RAPL's power measurement of CPU and DRAM for estimating power. RAPL limits the applicability of SmartWatts to Intel architectures. It also does not capture the impact of disk access and network usage on power that may be the main activity of an I/O or a network application. WattsApp on the other hand uses architecture agnostic parameters to model system power and its feasibility on multiple hardware platforms is demonstrated.

Lightweight power models, such as cWatts+ [17] and cWatts++ [18] are developed for containers. cWatts++ is a virtual power model that has two components: a client back-end and a server front-end. The client back-end is installed in the container and accesses the CPU event counters. cWatts++ uses two models, namely an event-based and RAPL-based models. The event-based model uses CPU performance counters and RAPL-based models uses only RAPL event counters. The evaluation shows that the two power models are useful on workloads obtained from the PARSEC and in-house benchmarks. However, cWatts only uses CPU related metrics to compute container power from server power. CPUs are a major power consuming component of a typical server (nearly one-third [6] and even up to 40% [16] of the total server power), but other components need to

be considered. Hence, WattsApp considers memory, IO and the network to account for container power. cWatts is also intrusive requiring container access for client installations.

There is research that accounts for the power consumption of individual threads and application containers [5]. The research relies on power estimation of each CPU core obtained from Intel's RAPL and hardware performance counters (related to CPU events) obtained from the OS. A power-aware consolidation technique based on a model built using CPU utilization is presented [12]. Both approaches are based on CPU-based metrics and do not account for the power consumed by other components [16]). WattsApp considers CPU, memory, disk and network related metrics.

Container Power Capping: Two power capping techniques are proposed in literature. The first is a power capping technique (DockerCap) for Docker containers [3]. The system power consumption is obtained from the hardware power meter and RAPL. The CPU quota of all the containers of different priority is reduced, thereby affecting the performance of all the containers. The WattsApp method however uses container migration and core reduction to achieve power capping. The merit is that the overall container performance is unaffected, but only the container that violates the power cap.

The second technique is proposed for Docker containers on the Kubernetes platform [2] by relying on DEEP-mon power monitoring [5]. This technique relies on RAPL and DVFS to manage power cap limits. It is demonstrated that RAPL affects the run-time performance of all containers on a server. RAPL enforces a power cap on the processor package and DRAM by reducing the CPU frequency and thus degrades the overall system performance. However, WattsApp uses architecture independent metrics to measure resource utilization (CPU, memory, disk, network) and is demonstrated to be effective for both power-aware scheduling and capping.

6 CONCLUSIONS

This paper proposes WattsApp that is underpinned by a six-step power-aware scheduling method for containers to minimize power cap violations on a server in real-time. The method relies on a neural network-based power estimation model. The trained model effectively predicts over 90% of data samples with less than 10% error. By testing on 10 representative benchmark workloads, the approach is able to achieve a MAPE error of less than 6%, and displays minimal overhead during run time scheduling. Unlike hardware-based power capping techniques, such as Intel's RAPL, which are indiscriminate to workloads and degrade the overall performance of all containers running on a server, this software-based approach is able to target individual containers running workloads, minimizing overall processing degradation while maintaining a node's power budget. The proposed technique considers multiple scenarios, including (i) single/multiple application, single container and single application, multiple containers. WattsApp has been shown to be feasible and outperforms existing techniques.

Future Work: WattsApp will be expanded for accelerator architectures. The current method prioritizes the power budget of an individual server, but not the performance of the container workloads. WattsApp will be explored for edge computing where power is a critical concern and containers are increasingly used. Alternate workloads (stream processing and sensor-based) will be considered.

REFERENCES

[1] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs Scale-out for Hadoop: Time to Rethink? In *Proc. of the 4th Annual Symposium on Cloud Comp.*, page 20, 2013.

[2] M. Arnaboldi, R. Brondolin, and M. D. Santambrogio. HyPPO: Hybrid Performance-aware Power-capping Orchestrator. In *Proc. of the IEEE International Conference on Autonomic Comp.*, pages 71–80, 2018.

[3] A. Asnaghi, M. Ferroni, and M. D. Santambrogio. DockerCap: A Software-level Power Capping Orchestrator for Docker Containers. In *Proc. of the IEEE International Conf. on Computational Science and Eng.*, pages 90–97, 2016.

[4] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé. Counter-based Power Modeling Methods: Top-down vs. Bottom-up. *The Computer Journal*, 56(2):198–213, 2012.

[5] R. Brondolin, T. Sardelli, and M. D. Santambrogio. Deep-mon: Dynamic and Energy Efficient Power Monitoring for Container-based Infrastructures. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 676–684, 2018.

[6] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. Process-level Power Estimation in VM-based Systems. In *Proc. of the 10th European Conference on Computer Systems*, 2015.

[7] G. Fieni, R. Rouvoy, and L. Seinturier. SmartWatts: Self-calibrating Software-defined Power Meter for Containers. In *Proc. of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Comp.*, 2020.

[8] C. Gu, H. Huang, and X. Jia. Power Metering for Virtual Machine in Cloud Computing - Challenges and Opportunities. *IEEE Access*, 2:1106–1116, 2014.

[9] J. M. Hirst, J. R. Miller, B. A. Kaplan, and D. D. Reed. Watts up? Pro AC Power Meter for Automated Energy Recording, 2013.

[10] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo. Characterizing Data Analysis Workloads in Data Centers. In *Proc. of the IEEE International Symposium on Workload Characterization*, pages 66–76, 2013.

[11] C. Kaewkasi and K. Chuenmuneewong. Improvement of Container Scheduling for Docker Using Ant Colony Optimization. In *Proc. of the 9th International Conference on Knowledge and Smart Technology*, pages 254–259, 2017.

[12] A. A. Khan, M. Zakarya, R. Buyya, R. Khan, M. Khan, and O. Rana. An Energy and Performance Aware Consolidation Technique for Containerized Datacenters. *IEEE Trans. on Cloud Comp.*, 2019.

[13] J. Krzywda, A. Ali-Eldin, T. E. Carlson, P.-O. Östberg, and E. Elmroth. Power-performance tradeoffs in data center servers: Dvfs, cpu pinning, horizontal, and vertical scaling. *Future Generation Computer Systems*, 81:114–128, 2018.

[14] E. Le Sueur and G. Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proc. of the International Conference on Power Aware Comp. and Systems*, pages 1–8, 2010.

[15] Z. Li, K. M. Greenan, A. W. Leung, and E. Zadok. Power Consumption in Enterprise-scale Backup Storage systems. *Power*, 2(2), 2012.

[16] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. Evaluating the Effectiveness of Model-based Power Characterization. In *Proc. of the USENIX Annual Technical Conference*, 2011.

[17] J. Phung, Y. C. Lee, and A. Y. Zomaya. Application-agnostic Power Monitoring in Virtualized Environments. In *Proc. of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Comp.*, pages 335–344, 2017.

[18] J. Phung, Y. C. Lee, and A. Y. Zomaya. Lightweight Power Monitoring Framework for Virtualized Computing Environments. *IEEE Trans. on Computers*, 69(1):14–25, 2019.

[19] B. Ruan, H. Huang, S. Wu, and H. Jin. A Performance Study of Containers in Cloud Environment. In *Proc. of the Asia-Pacific Services Comp. Conference*, pages 343–356, 2016.

[20] K. M. D. Sweeney and D. Thain. Efficient Integration of Containers Into Scientific Workflows. In *Proc. of the 9th Workshop on Scientific Cloud Comp.*, pages 7:1–7:6, 2018.

[21] S. S. Tadesse, F. Malandrino, and C.-F. Chiasserini. Energy Consumption Measurements in Docker. In *Proc. of the 41st Annual IEEE Computer Software and Applications Conference*, volume 2, pages 272–273, 2017.

[22] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. Nikolopoulos. Challenges and Opportunities in Edge Computing. In *Proc. of the IEEE International Conference on Smart Cloud*, pages 20–26, 2016.

[23] N. Wang, B. Varghese, M. Matthaiou, and D. Nikolopoulos. ENORM: A Framework for Edge Node Resource Management. *IEEE Trans. on Services Comp.*, 2017.

[24] O. Weidner, M. Atkinson, A. Barker, and R. Filgueira. Rethinking High Performance Computing Platforms: Challenges, Opportunities and Recommendations. In *Proc. of the ACM Int. Workshop on Data-Intensive Distributed Comp.*, pages 19–26, 2016.

[25] H. Zhang and H. Hoffman. A Quantitative Evaluation of the RAPL Power Control System. *Feedback Comp.*, 2015.

[26] R. Zhou, Z. Li, and C. Wu. Scheduling Frameworks for Cloud Container Services. *IEEE/ACM Trans. on Networking*, 26(1):436–450, 2018.