Energy-aware Provisioning of Microservices for Serverless Edge Computing

Energy-aware Provisioning of Microservices for Serverless Edge Computing

Madhura Adeppady
Politecnico di Torino
Torino, Italy

Alberto Conte
Nokia Bell Labs
Nozay, France

Holger Karl
Hasso Plattner Institute
Uni. of Potsdam, Germany

Paolo Giaccone
Politecnico di Torino
Torino, Italy

Carla Fabiana Chiasserini
Politecnico di Torino
Torino, Italy

***Abstract*—Serverless edge computing allows for highly efficient resource utilization, reducing the energy footprint of edge data centers. Indeed, the containers can be dynamically created and destroyed, allowing to adapt the workload to the available resources. Creating containers upon arrivals of service requests entails, however, a high start-up latency, which may be unsuitable for time-critical services. As alternative solution, pre-started containers ("warm containers") are used to decrease start-up latency, but incurring in higher resource costs.**

In this work, we minimize the energy consumption of the active servers in the data center by optimally managing the various container states while meeting the target delay of the requested services. Further, in light of the problem complexity, we investigate how a simple threshold-based algorithm performs and show that it can closely match the optimum.

I. Introduction

Edge computing has emerged as a solution to serve a large number of real-time computational tasks while reducing bandwidth usage and end-to-end latency [1]. Despite enjoying many benefits, the widespread deployment of edge computing is still challenging [2]. From the system perspective, provisioning computing resources at the granularity of virtual machines, as done in traditional cloud computing, brings in long provisional delays and resource wastage, which is unacceptable for resource-constrained edge servers and time-critical services. Also, application developers still bear the heavy burden of explicitly managing the resources, load balancing, and scalability. Serverless computing, with its Function-as-a-Service (FaaS) offering, redefines the way of deploying services [3], as it enables the decomposition of their logic into stateless microservices (MS). Such MSs are run on demand in an event-driven manner in lightweight containers with no need for resource pre-allocation. Using serverless edge computing allows services to use underlying resources on demand without the burden of load balancing, scalability, and runtime environments, thus improving resource utilization [4], [5].

Importantly, in serverless edge computing, MSs run inside the containers only when requested. Thus, serving a request involves creating a new container with appropriate runtime, which may involve downloading the necessary image from the remote repository, fetching and loading essential libraries and dependencies before executing the actual function. This process is known as *cold start* and the long delay involved in the initialization setup is known as start-up latency, which is one of the main performance issues faced by serverless computing platforms [6]–[8]. A *warm* container keeps instead the MS instance alive in the memory, with a negligible start-up latency when the warm containers is reused for serving a later request for the same MS. However, due to limited memory at the edge nodes, serving all the requests with warm containers is practically impossible [4]. Further, keeping warm containers in memory reduces resource utilization and violates the resource elasticity promises of serverless computing, in which resources are occupied when required.

Recently, many research efforts have been devoted to reducing the cold start frequencies MSs by proposing various strategies for managing the keep-alive time of the warm containers [6], [7]. However, these approaches allocate the resources to the container by largely overlooking MS-specific QoS requirements, e.g., target delay. Unlike prior work, in this paper, we face the above issue with the additional twofold aim to (i) ensure the level of QoS required by the MSs offered to the mobile users, and (ii) reduce the data centers energy footprint. Indeed, it is well known that edge data centers consume a significant amount of energy, which depends on their CPU load [9]. Towards these goals, we provide the following contributions:

1) Through a detailed, yet tractable, model of the system (Sec. II), we formulate an optimization problem that, looking at a finite time horizon, minimizes the servers energy consumption by leveraging cold, warm, and running containers (Sec. III). In particular, we note that, for the MSs with stringent delay constraints, serving the requests using a cold container requires a high CPU speed allocation. In contrast, using a warm container is more energy efficient because it requires low CPU speed allocation due to negligible start-up latency. However, it may be impossible to serve all requests with warm containers due to the limited memory of edge servers.

2) Since, in spite of the limited lookahead perspective of the proposed formulation, the problem turns out to be NP-hard, we investigate a simple threshold-based queueing solution (Sec. IV), which, surprisingly, closely matches the optimum (Sec. V). Specifically, we define a threshold on the number of requests waiting to be served, and start new containers from cold/warm state only when this number exceeds the threshold. By adopting a FIFO policy for scheduling the requests and restricting the number of waiting requests, we can reduce the

This work was supported by the EC through the SEMANTIC project (Grant No. 861165) and by the DFG through the SWAVES project.

number of running containers and cold starts, hence decreasing the overall energy consumption of the active servers.

II. System Model

Let us focus on a single data center and let $\mathcal{S}$ be the set of servers available therein, with $s \in \mathcal{S}$ having $\hat{\tau}_s$ bytes of memory and $\hat{\mu}_s$ CPU capacity (in cycles/s). A service orchestrator receive requests for any MS $k \in \mathcal{K}$ and serves them using the serverless computing paradigm. Any request for MS k demands certain amount of memory and workload (in CPU cycles), denoted by τ_k and w_k, respectively, and has a target maximum delay D_k in terms of time lapse from when the request arrives till the service execution is completed.

A container can be in any of the three states: running (R), warm (W), cold (C) or in any of these two transition states: transiting from C to R ($T_{C\to R}$), or transiting from R to C ($T_{R\to C}$). The transition to/from W involves negligible start-up latency, hence we consider only $T_{C\to R}$ and $T_{R\to C}$. For a container implementing service k, let $\delta_{k,C}$ denote the start-up latency of cold start. For simplicity, we assume that the transition time from state R to state C is same as $\delta_{k,C}$.

Let $c_{k,i,R}(t)$ be the container in state R serving the i-th request for MS k, $r_{k,i}$ at time t. When starting to serve request $r_{k,i}$, the orchestrator assigns a CPU speed in cycles/s, denoted by $\mu_{c_{k,i,R}}$, to container $c_{k,i,R}$ such that i) overall CPU capacity at the server is not exceeded and ii) $r_{k,i}$ is served within D_k. Additionally, the memory assigned to any container implementing MS k in state R is equal to the memory demand of an instance of MS k. We stress that a container in warm state at time t that implements MS k, $c_{k,W}(t)$, consumes only memory $\tau_{k,W}$. When in state C, a container consumes neither CPU nor memory. Further, let $c_{k,i,T_{C\to R}}(t)$ (or, $c_{k,i,T_{R\to C}}(t)$) be the container in transition from C to R (or, from R to C) to serve request $r_{k,i}$ at t. A container of MS k in any of the above mentioned transition states consumes both memory and CPU cycles/s, denoted by $\mu_{k,T}$ and $\tau_{k,T}$, respectively. For MS k, the memory and CPU consumption of all the container transition states remain the same.

We assume that service requests arrive randomly (e.g., according to a Poisson process) and are enqueued on a specific queue denoted by Q_k, for each service $k \in \mathcal{K}$. The orchestrator takes a decision regarding MS k only when either of these two events occur: i) a new request for MS k arrives, ii) container $c_{k,i,R}(t)$ finishes serving request $r_{k,i}$ (at any point in time, there could be multiple containers of service k in state R). Another possible event is a container finishing its transition. In this case, the orchestrator does not make any decision. If a container finishes a transition to the cold state, it is destroyed. Otherwise, if $c_{k,i,T_{C\to R}}(t)$ finishes its transition to become $c_{k,i,R}(t)$, request $r_{k,i}$ is removed from queue Q_k to be executed on it. Also, the orchestrator acts upon each queue according to a FIFO policy, by serving the requests on appropriate containers. Note however that the head-of-the-line (HoL) request and the first request to handle in the queue may not always be the same. If the HoL request has already been scheduled to run on a container that is currently transitioning, then the second request in the queue becomes the first request to handle. Instead, if the HoL request in the queue is not scheduled to run on any container, then it remains as the first request to handle in the queue.

For serving the first request to handle in the queue, we have two cases. In the first one, we consider that no warm container is available. Hence, the orchestrator creates a new container and, once the container reaches state R, the first request in the queue is served on it. In the second case, we consider that warm containers are available. In such a situation, the orchestrator can serve the request in a warm or cold container. In both cases, after being assigned to a warm or cold container, the request still remains in the queue and will be removed when the container enters state R. During this time, the subsequent request in the queue becomes the first request to be handled by the orchestrator. Let t be the time at which any of the previously mentioned events (arrival of a new request, container finishes serving a request, or container completes the transition) occurs. At time t, we denote by $\Omega^s_{k,W}(t)$, $\Omega^s_{k,R}(t)$, and $\Omega^s_{k,T}(t)$ the set of all containers of MS k on server s in state W, R, and $T_{C\to R}$ or $T_{R\to C}$, respectively. Let $\Omega_{k,R}(t) = \cup_{s\in\mathcal{S}}\Omega^s_{k,R}(t)$, and $\Omega_{k,T}(t) = \cup_{s\in\mathcal{S}}\Omega^s_{k,T}(t)$ be the set of running and transiting containers across all the servers, respectively.

Processing time $T_{k,i}$ of request $r_{k,i}$ running on container $c_{k,i,R}$ is computed by $T_{k,i} = w_k/\mu_{c_{k,i,R}}$ where we assume that $c_{k,i,R}$ runs at rate $\mu_{c_{k,i,R}}$ to process $r_{k,i}$, and thus the expression holds independently from the concurrent requests. The total service time of $r_{k,i}$ is the sum of the queueing delay before being served and the processing time. Note that the queueing delay experienced by a request also includes the start-up latency of the container on which the request is scheduled to run once the container enters state R.

The power consumption of a server is composed of (i) a constant P_{idle} representing the idle power consumption when a server is active, and (ii) a function P of the server's actual CPU load. Thus, the power consumption of server s with CPU load λ_s is given by $P_s = P_{\text{idle}} + P(\lambda_s)$.

III. Problem Formulation

We now describe the event-driven problem formulation whose objective is to reduce the data center's power consumption by minimizing the CPU load on the servers while ensuring that requests are served within the desired target delay.

Below, if the queue Q_k does not contain unhandled requests, then we denote this case by setting $\epsilon_k = 1$. Otherwise, $\epsilon_k = 0$. At t, the CPU load $\lambda_{s,t}$ of a server s is the sum of the CPU load by all the running and transiting containers on s, i.e.,

$$\lambda_{s,t} = \sum_{k\in\mathcal{K}} \Bigg(\sum_{c_{k,i,R}\in\Omega^s_{k,R}(t)} \mu_{c_{k,i,R}} + \sum_{c_{k,T}\in\Omega^s_{k,T}(t)} \mu_{k,T} \Bigg). \quad (1)$$

Similarly, at t, the memory consumption $\nu_{s,t}$ of server s is the sum of memory consumed by all the running, transiting,

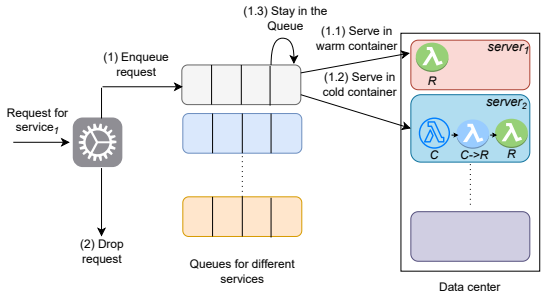

Fig. 1. Various decisions the orchestrator can take upon a new request arrival

and warm containers on s, i.e.,

$$\nu_{s,t}=\sum_{k\in\mathcal{K}}\Big(\sum_{c_{k,i,R}\in\Omega^s_{k,R}(t)}\tau_k+\sum_{c_{k,W}\in\Omega^s_{k,W}(t)}\tau_{k,W}+\sum_{c_{k,T}\in\Omega^s_{k,T}(t)}\tau_{k,T}\Big). \quad (2)$$

A. Problem formulation of new request arrival event

At t, upon the arrival of a new request, $r_{k,i}$, the orchestrator makes one of the decisions on MS k as listed below.

1) The orchestrator decides to queue $r_{k,i}$ into the queue Q_k. Let $t_{r_{k,i}}$ be the queue admission time of $r_{k,i}$, which is set to t. The decision of queueing $r_{k,i}$ is expressed by setting $q_{r_{k,i}}$=1. While queueing $r_{k,i}$, the orchestrator decides $\mu_{r_{k,i}}$, the CPU speed to allocate to it. After queueing $r_{k,i}$, the orchestrator has three options:

(a) To schedule the first request in the queue $r_{k,j}$ on an existing warm container $c_{k,W}(t)$ in server $s\in\mathcal{S}$. We denote this decision by setting $x_{r_{k,j},c_{k,W}}$=1 (Case 1.1 of Fig. 1). The decision variable $y_{r_{k,j},s}$=1 indicates that the request is served in server s. The container $c_{k,W}(t)$ moves to state R (denoted by $c_{k,j,R}(t)$) with negligible start-up latency and serves $r_{k,j}$. The CPU and the memory allocated to $c_{k,j,R}$ are set equal to, respectively, the CPU allocated to $r_{k,j}$ and to MS k's memory requirements. Also, the orchestrator revises the CPU speed allocated to all the requests in the queue. For the n-th request $r_{k,j+n}$ in Q_k, with j corresponding to the first request, let $\mu_{r_{k,j+n}}$ be the revised CPU speed allocated to it, where $n\in\{0,1,...,|Q_k|-1\}$.

(b) To schedule the first request to handle in the queue $r_{k,j}$ on a cold container in server s ($y_{r_{k,j},s}$=1), denoted by $z_{r_{k,j},c_{k,C}}$=1 (Case 1.2 of Fig. 1). That is, at t, a new container is created and starts the transition to R ($c_{k,j,T_{C\to R}}(t)$). After the start-up latency of $\delta_{k,C}$, $r_{k,j}$ is run on $c_{k,j,R}(t+\delta_{k,C})$. At t, the orchestrator sets CPU speed of $c_{k,j,R}(t+\delta_{k,C})$ to $\mu_{r_{k,j}}$, while the memory allocated to $c_{k,j,R}(t+\delta_{k,C})$ is same as the memory requirement of MS k. As before, the orchestrator revises the CPU speed allocated to all the queued requests.

(c) Not to start a cold or warm container to serve the first request in the queue, denoted by setting $z_{r_{k,j},c_{k,C}}=0$ and $x_{r_{k,j},c_{k,W}}=0$, i.e., the first request in Q_k, $r_{k,j}$, remains in the queue for the time being (Case 1.3 of Fig. 1), and the CPU speed assigned to queued requests is not revised.

2) The orchestrator drops the new request $r_{k,i}$ (Case 2 of Fig. 1). This decision is denoted by setting $q_{r_{k,i}}=0$.

Let $m_k(t)\in\mathcal{M}_k(t)$ denote the time at which currently running, transiting, and warm containers are available to serve the first request to handle in the queue, relative to current time t. For the first request to handle, $r_{k,j}$, let $\mu_{r_{k,j}}$ and Δt_j be its revised CPU speed allocation and the time at which an existing container will start the execution of $r_{k,j}$ relative to t. The orchestrator will run the first request $r_{k,j}$ waiting in the queue on the container that can start serving the request $r_{k,j}$ at the earliest, i.e., at $\Delta t_j=\min(\mathcal{M}_k(t))$. The queueing delay experienced by $r_{k,j}$ so far is $t-t_{r_{k,j}}$, where $t_{r_{k,j}}$ is the queue admission time of the request $r_{k,j}$. Additionally, $r_{k,j}$ will stay for Δt_j amount of time in the queue before being served. To get served within the target delay, the revised CPU speed of $r_{k,j}$ is calculated by the following equation, $\mu_{r_{k,j}}=w_k/[D_k-(t-t_{r_{k,j}}+\Delta t_j)]$. When $r_{k,j}$ runs on this container, the container's new residual time to finish the execution will be equal to $r_{k,j}$'s processing time. Let $\mathcal{M}_k(\Delta t_j)$ be the set of updated residual times of the running and transiting containers at Δt_j to finish their assigned tasks, which is given by

$$\mathcal{M}_k(\Delta t_1)=\Big\{m_k(t)-\Delta t_1,\ \forall m_k(t)\in\mathcal{M}_k(t)\setminus\min(\mathcal{M}_k(t))\Big\}\cup\Big\{\frac{w_k}{\mu_{r_{k,1}}}\Big\} \quad (3)$$

where $m_k(t)$ is the generic element of set $\mathcal{M}_k(t)$.

Similarly, with respect to the current time t, the second request to handle $r_{k,j+1}$ will be served at $\Delta t_{j+1}=\Delta t_j+\min(\mathcal{M}_k(\Delta t_j))$, The revised CPU speed of $r_{k,j+1}$ is given by: $\mu_{r_{k,j+1}}=w_k/[D_k-(t-t_{r_{k,j+1}}+\Delta t_{j+1})]$. Set $\mathcal{M}_k(\Delta t_{j+1})$ containing the time relative to Δt_{j+1} at which the running and transiting containers can handle the third request in Q_k is,

$$\mathcal{M}_k(\Delta t_{j+1})=\Big\{m_k(\Delta t_j)-(\Delta t_{j+1}-\Delta t_j),\ \forall m_k(\Delta t_j)\in\mathcal{M}_k(\Delta t_j)\setminus\min(\mathcal{M}_k(\Delta t_j))\Big\}\cup\Big\{\frac{w_k}{\mu_{r_{k,j+1}}}\Big\}.$$

Generalizing the above equation, the n-th request to handle in the queue $r_{k,j+n}$ will be served at $\Delta t_{j+n}=\Delta t_{j+n-1}+\min(\mathcal{M}_k(\Delta t_{j+n-1}))$. By revising the CPU speeds, the time at which the running and transiting containers can handle request n in Q_k relative to Δt_{j+n} is given by,

$$\mathcal{M}_k(\Delta t_{j+n})=\Big\{m_k(\Delta t_{j+n-1})-(\Delta t_{j+n}-\Delta t_{j+n-1}),\ \forall m_k(\Delta t_{j+n-1})\in\mathcal{M}_k(\Delta t_{j+n-1})\setminus\min(\mathcal{M}_k(\Delta t_{j+n-1}))\Big\}\cup\Big\{\frac{w_k}{\mu_{r_{k,j+n}}}\Big\}. \quad (4)$$

Thus, the residual queueing delay of the n-th queued request $r_{k,j+n}$ is given by Δt_{j+n}, where $n\in\{0,1,...,|Q_k|-1\}$.

The queueing delay of the newly arrived request $r_{k,i}$ will be the sum of the time at which the last request waiting in the queue will start executing, and the updated remaining time of the running and transiting containers at $\Delta t_{j+|Q_k|-1}$ is computed by, which is given by,

$$d_{r_{k,i}}(\mathcal{M}_k(t))=\Delta t_{j+|Q_k|-1}+\min(\mathcal{M}_k(\Delta t_{j+|Q_k|-1})). \quad (5)$$

We use the above estimated worst-case queueing delays in Sec. III-C for making decisions on starting a new container from cold/warm state, and in Secs. III-C and III-D to revise the allocated CPU speeds.

B. Pre-computation of power consumption

Upon the arrival of a new request $r_{k,i}$, the orchestrator needs to make a decision that minimizes the power consumption of the servers in the data center. The optimal policy for this problem is non-trivial. Keeping the requests longer in the queue may reduce the total number of running and transiting containers in the system. But this policy might result in a higher CPU speed allocation to these running containers and hence higher overall power consumption. On the other hand, if we keep the requests for a shorter duration in the queue, we may have higher number of transiting and running containers with lower CPU speed allocated, again resulting in a higher power consumption. Moreover, this policy will reduce the reusability of the same container to serve other requests. Thus, in the objective function, considering just the CPU cycles assigned to currently running and transiting containers is not enough; we also need to account for the impact of queued requests on the power consumption.

To account for the impact of queued requests on the power consumption, we pre-compute the possible power consumptions based on the various decisions the orchestrator can make for the first request in the queue and the newly arrived request. At t_1, the orchestrator knows the request execution end and transition complete events for the existing transiting and running containers. Let $\mathcal{L} = \{t_1, t_2, ...\}$ be the set of the time instants at which these known events will occur, including the current arrival at t_1. Based on the decisions made for the new request and the first request in the queue, the orchestrator updates $\mathcal{L}$ to include the time of occurrences of events of request execution end for the queued requests as well.

Let $\mathcal{L}_q$ be the auxilliary set containing the time at which each of the known events will occur if the orchestrator decides to queue the newly arrived request and not to serve the first request in the queue (i.e., $q_{r_{k,i}}$=1). Then the power consumption P_q across all the servers in the data center is $P_q = \sum_{\hat{s}\in\mathcal{S}} \sum_{i=1}^{|\mathcal{L}_q|-1} \int_{t_i}^{t_{i+1}} (P_{\text{idle}} + P(\lambda_{\hat{s},t_i,q}))dt$ where $\lambda_{\hat{s},t_i,q}$ is the load of server $\hat{s}$ at t_i if the decision is to queue the newly arrived request and not to serve the first request in the queue for the time being. Similarly, the orchestrator pre-computes the power consumption for the other decisions. If the orchestrator decides to queue the new request and start a warm container on server s to serve the first request in the queue, the power consumption $P_{x,s}$ across all the servers in the data center is $P_{x,s}$=$\sum_{\hat{s}\in\mathcal{S}} \sum_{i=1}^{|\mathcal{L}_{x,s}|-1} \int_{t_i}^{t_{i+1}} (P_{\text{idle}} + P(\lambda_{\hat{s},t_i,x,s}))dt$, where $\lambda_{\hat{s},t_i,x,s}$ is the load of server $\hat{s}$ at t_i and $\mathcal{L}_{x,s}$ is the auxilliary set containing the time of occurrence of known events. If the decision is to use a cold container on server s to serve the first request in the queue, then the pre-computed power consumption is represented by $P_{z,s}$=$\sum_{\hat{s}\in\mathcal{S}} \sum_{i=1}^{|\mathcal{L}_{z,s}|-1} \int_{t_i}^{t_{i+1}} (P_{\text{idle}} + P(\lambda_{\hat{s},t_i,z,s}))dt$. Finally P=$\sum_{\hat{s}\in\mathcal{S}} \sum_{i=1}^{|\mathcal{L}|-1} \int_{t_i}^{t_{i+1}} (P_{\text{idle}}+P(\lambda_{\hat{s},t_i}))dt$ represents the pre-computed power consumption if the new request is not queued.

C. Objective function for arrival event

The objective is to minimize the power consumption across all servers as well as the expected power consumption in the future, based on the currently known events (i.e., request end and transition complete) while minimizing the number of dropped requests. Notice that the number of requests served can be maximized by adding a high penalty F, if the orchestrator decides to drop the newly arrived request. Thus, upon the arrival of a new request, $r_{k,i}$, the orchestrator should solve the following problem:

$$\begin{aligned}\min_{\{y,x,q,z,\{\mu\}\}} \Big[& q_{r_{k,i}} \cdot (1-x_{r_{k,j},c_{k,W}}) \cdot (1-z_{r_{k,j},c_{k,C}}) \cdot P_q \\ & + q_{r_{k,i}} \cdot x_{r_{k,j},c_{k,W}} \cdot (1-z_{r_{k,j},c_{k,C}}) \cdot \sum_{s\in\mathcal{S}} y_{r_{k,j},s} \cdot P_{x,s} \\ & + q_{r_{k,i}} \cdot (1 - x_{r_{k,j},c_{k,W}}) \cdot z_{r_{k,j},c_{k,C}} \sum_{s\in\mathcal{S}} y_{r_{k,j},s} \cdot P_{z,s} \\ & + (1 - q_{r_{k,i}}) \cdot P \Big] + (1 - q_{r_{k,i}}) \cdot F \end{aligned} \quad (6)$$

subject to the constraints below (formal expressions can be found in [10], and are omitted for the sake of readability): (1) At any t, the CPU cycles used by the pre-existing running and transiting containers as well as by the newly scheduled container do not exceed the server capability. (2) The memory allocated to pre-existing containers in warm, running, and transiting states and newly scheduled container cannot exceed the server capability. (3) If the decision is to run the first request in the queue $r_{k,j}$ on a warm container, the total delay experienced by $r_{k,j}$ cannot exceed the target delay of MS k. (4) If the decision is to run the first request in the queue $r_{k,j}$ on a cold container, the total delay experienced by $r_{k,j}$ cannot exceed the target delay of MS k. (5) If the decision is to queue the new request, the total delay the request is going to experience in the worst case cannot exceed D_k. Note that the queueing delay the new request is going to experience depends on the decisions made on the first request in the queue. (6) The CPU speed allocated to all queued requests must be revised depending on the decisions made on the first request in the queue and the MS target delay. (7) If $x_{r_{k,j},c_{k,W}} = 1$ or $z_{r_{k,j},c_{k,C}} = 1$, we must specify on which server the warm or cold container has started its transition. Further, when $x_{r_{k,j},c_{k,W}} = 1$, $y_{r_{k,j},s}$ can be set only for that server which has a warm container; (8) $r_{k,j}$ is either scheduled in a warm container or in a cold container or later.

D. Problem formulation for request execution ends

At time t, when the running container $c_{k,i,R}$ finishes its current execution on server $\hat{s} \in \mathcal{S}$, the orchestrator makes one of the following decisions on MS k,

1) To schedule the first request to handle in the queue $r_{k,j}$ on the container $c_{k,i,R}(t)$, as shown in case 1 of Fig. 2. Note that this decision is possible since we assume that the

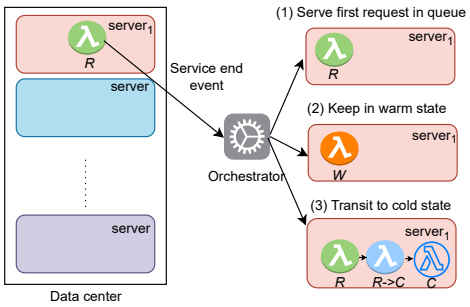

Fig. 2. Decisions the orchestrator can make upon request execution ends

transition time of a container from the state R to the state W is negligible. This decision is denoted by setting $x_{r_{k,j},c_{k,i,R}}$=1 and $e_{c_{k,i,R}}$=2. At this stage, $c_{k,i,R}(t)$ is referred to as $c_{k,j,R}(t)$, and CPU speed allocated to $c_{k,j,R}(t)$ is updated to $\mu_{r_{k,j}}$.

2) Not to schedule the first request to handle in the queue $r_{k,j}$ on the container $c_{k,i,R}(t)$ denoted by setting $x_{r_{k,j},c_{k,i,R}}$=0. Further, to set $x_{r_{k,j},c_{k,i,R}}$=0, the orchestrator needs to revise the CPU speed allocated to the queued requests such that they are served within the target delay of the service k. For n-th queued request, let $\mu_{r_{k,j+n}}$ be its revised CPU speed, where $n \in \{0, 1, ..., |Q_k| - 1\}$. Additionally, the orchestrator makes one of the following decisions regarding $c_{k,i,R}(t)$:

1) To keep $c_{k,i,R}(t)$ in state W, denoted by $e_{c_{k,i,R}}$=1 (case 2 in Fig. 2). This is useful to reduce the start-up latency of future requests and serve them with low CPU speed.

2) To delete the container $c_{k,i,R}(t)$, denoted by $e_{c_{k,i,R}}$=0, and the container $c_{k,i,R}(t)$ enters state $T_{R\to C}$, and it will be removed from the system once it reaches state C. This scenario is represented in case 3 of Fig. 2. Without knowing the future arrival pattern, deciding whether to keep the container in the state W or C is challenging. To make this decision, we considered probability p_0 of no arrivals within t and $t+\delta_{k,C}$, and added this as a penalty in the objective function.

The orchestrator decides to set $\hat{x}_{r_{k,j},c_{k,i,R}} = 0$, only if we can meet the target delay of the queued requests with existing running and transiting containers of MS k excluding $c_{k,i,R}(t)$.

For the request execution end event, let $P_{\hat{x}}$ be the pre-computed power consumption if the decision is to set $\hat{x}_{r_{k,j},c_{k,i,R}}$=1. Let P_{e_1} and P_{e_0} be the pre-computed power consumptions if the orchestrator decides to keep $c_{k,i,R}(t)$ in warm and cold states, respectively. We follow the same procedure described in Sec. III-B to pre-compute these power consumptions. Then the objective function is given by

$$\min_{\{x,e,\{\mu\}\}} \hat{x}_{r_{k,j},c_{k,i,R}}\cdot P_{\hat{x}} + (1-\hat{x}_{r_{k,j},c_{k,i,R}})\cdot e_{c_{k,i,R}}\cdot P_{e_1} + (1-\hat{x}_{r_{k,j},c_{k,i,R}})\cdot(1-e_{c_{k,i,R}})\cdot P_{e_0} - (1-\hat{x}_{r_{k,j},c_{k,i,R}})\cdot\Big[e_{c_{k,i,R}}\cdot(1-p_0) + (1-e_{c_{k,i,R}})\cdot p_0\Big] \quad (7)$$

subject to the following constraints (a formal expression of the constraints can be found in [10] and is omitted for the sake of readability). (1) The CPU cycles used by currently running, transiting, and CPU cycles consumed by newly scheduled container cannot exceed the server available CPU capacity. (2) The memory allocated to pre-existing containers in warm, running, and transiting states and newly scheduled container cannot exceed the server memory. (3) The deadline of queued requests must be met with revised CPU speeds, if the orchestrator sets $\hat{x}_{r_{k,j},c_{k,i,R}}$=0. (4) If $\hat{x}_{r_{k,j},c_{k,i,R}}$=0, either the container is kept in W state ($e_{c_{k,i,R}} = 1$) or it is destroyed ($e_{c_{k,i,R}} = 0$); if $\hat{x}_{r_{k,j},c_{k,i,R}} = 1$, the container starts serving request $r_{k,j}$ once it enters state R, i.e., $e_{c_{k,i,R}} = 2$.

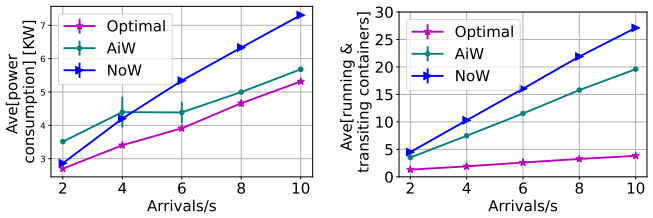

Fig. 3. NoW and AiW vs. optimal: average power consumption (left); average number running and transiting containers (right).

The optimization problem for new request arrival and request execution end events is in the form of Mixed Integer Non-Linear Program (MINLP) and, hence, NP-hard [11].

IV. Threshold-based Algorithm

The key idea is to set a threshold on the number of unhandled requests in the queue and create a running container from cold/warm state only when the number of unhandled requests in the queue exceeds the defined threshold. This idea reduces the number of running and transiting containers in the system, thereby minimizing overall energy consumption.

Upon the arrival of a new request $r_{k,i}$ for MS k, the orchestrator decides whether to start a new container based on the number of unhandled requests in the queue Q_k. Suppose the number of unhandled requests in Q_k is greater than or equal to the threshold. In that case, the orchestrator first verifies whether the first request to handle in the queue $r_{k,j}$ can be served on an existing warm container. If so, it determines the CPU speed to allocate to the warm container so as to meet the target delay using the expression $\mu_{r_{k,j}} = w_k/[D_k-(t-t_{r_{k,j}})]$. All the eligible servers having the warm containers of MS k and enough computing resources to serve $r_{k,j}$ are sorted in the increasing order of their remaining CPU speeds. The algorithm then selects the first server from the sorted set to serve $r_{k,j}$.

If instead there are no warm containers in any server, the orchestrator determines whether the request can be served in the cold container, as well as the CPU speed to allocate to the container using: $\mu_{r_{k,j}} = w_k/[D_k - (t - t_{r_{k,j}} + \delta_{k,C})]$. All the eligible servers having enough computing and memory resources to serve $r_{k,j}$ in a cold container are sorted in the increasing order of their remaining CPU speeds. The orchestrator selects the first server in the sorted set to serve $r_{k,j}$. The request is dropped if it cannot be served in either warm or cold containers. Finally, after making one of the above decisions, $r_{k,i}$ is enqueued.

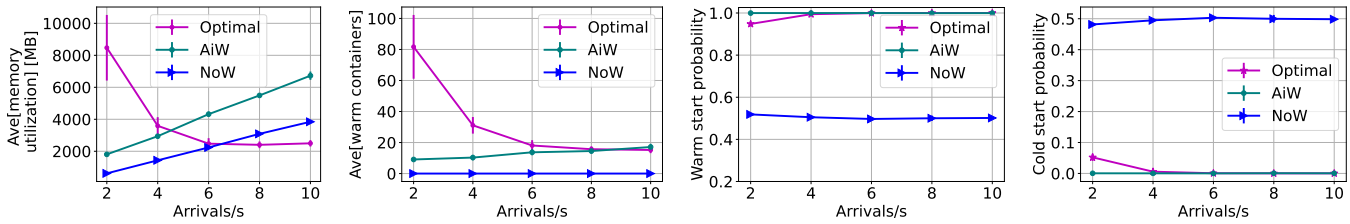

Fig. 4. Performance of NoW and AiW vs. optimal, as the arrival rate of service requests varies: average memory consumption (left most); average number of warm containers (left); warm (right) and cold (right most) start probabilities

At time t, when an existing container $c_{k,i,R}(t)$ finishes its current execution on server $\hat{s}$, we again follow a threshold-based approach to make decisions on the container and first request to handle in the queue $r_{k,j}$. If the number of unhandled requests waiting in the queue exceeds the threshold, the orchestrator decides whether $c_{k,i,R}(t)$ can be reused to serve the first request in Q_k, exploiting $\mu_{r_{k,j}} = w_k/[D_k - (t - t_{r_{k,j}})]$ to determine the CPU speed to allocate to $c_{k,i,R}(t)$. If the server has enough computing and memory resources, then $r_{k,j}$ is served on $c_{k,i,R}(t)$. Otherwise, the first request is dropped and we have two options: i) *NoW*: after serving a request, if the queue is empty, the container is always transitioned to state C, ii) *AiW*: after serving a request, if the queue is empty, the container is always kept in state W.

V. Performance Evaluation

Here, we evaluate the performance of the optimal solution against the simple threshold based queueing solution. Due to the complexity of the optimum, we focus on a small-scale scenario and derive the optimum using Gurobi. In particular, we consider only one type of MS, and that requests arrive according to a Poisson process with varying rate. Further, we consider a simple Python function as MS, and, using OpenWhisk, we measured its start-up latency, which resulted to be $\delta_{k,C} = 1.5\,\text{s}$. The MS workload requirement is fixed to $w_k = 3\,\text{G}$ clock cycles. In the simulation setup, we set the number of active servers in the data center to 4. Further, we pre-create three warm containers per server at the beginning of the simulation and the simulation is carried out for 1,000 s. We set the target delay for the MS requests to 2 s.

Surprisingly, Fig. 3(left-center) shows that the power consumption of AiW is comparable with that of the optimum. Optimum uses significantly fewer containers to serve the requests; however, the containers run at higher CPU speeds due to queueing delay. Conversely, AiW uses a higher number of containers to serve the requests, but, thanks to the threshold on the number of requests in the queue, these containers run at lower CPU speeds. As a result, the overall CPU load at the data center is comparable to the optimum. In the case of NoW, the power consumption is higher than both AiW and the optimum because of the higher number of cold starts. Further, Fig. 4 presents the measured average memory consumption and the number of warm containers across all the servers in the data center for various arrival rates. Note that warm, running, and transiting containers contribute to the server's memory utilization. For lower arrival rates ($< 6/s$), the memory consumption in the optimum case is higher because the containers are kept in the warm state after serving the requests, but, due to the low arrival rate, they are never used again. Consequently, the memory consumption of NoW is the lowest because it never keeps the container in the warm state.

Finally, Fig. 4(center-right) depicts the cold and warm start probabilities for optimum, AiW, and NoW. The warm start probabilities of AiW and optimum are close to one because the containers are either used again to serve the requests or kept warm to serve future requests. However, the NoW approach has a lower warm start probability because the container is always transitioned to the cold state if the queue is empty.

VI. Conclusions

We addressed the problem of reducing the data center's energy footprint in a serverless edge computing scenario. We formulated an optimization problem aimed at minimizing the energy consumption of active servers in the data center by utilizing cold, warm, and running containers over a finite time horizon. As the problem turns out to be NP-hard, we investigated the optimum against a simple threshold based queueing solution through a small scale simulation setup. Our results reveal that the performance of the threshold-based queueing solution closely matches the optimum in terms of overall energy and memory consumption of the data center.

References

[1] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing a key technology towards 5G," tech. rep., ETSI, 2015.

[2] Z. Tao *et al.*, "A survey of virtual machine management in edge computing," *Proc. of the IEEE*, vol. 107, no. 8, pp. 1482–1499, 2019.

[3] E. Jonas *et al.*, "Cloud programming simplified: A Berkeley view on serverless computing," *arXiv:1902.03383*, 2019.

[4] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *IEEE INFOCOM*, 2022.

[5] R. Xie, D. Gu, Q. Tang, T. Huang, and F. R. Yu, "Workflow scheduling in serverless edge computing for the industrial internet of things: A learning approach," *IEEE Trans. on Ind. Informatics*, pp. 1–10, 2022.

[6] I. E. Akkus *et al.*, "SAND: Towards High-Performance serverless computing," in *USENIX ATC*, 2018.

[7] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *ATC*, 2020.

[8] A. Mohan *et al.*, "Agile cold starts for scalable serverless," in *HotCloud*, 2019.

[9] P. Ruiu, C. Fiandrino, P. Giaccone, A. Bianco, D. Kliazovich, and P. Bouvry, "On the energy-proportionality of data center networks," *IEEE Trans. on Sustainable Comp.*, vol. 2, no. 2, pp. 197–210, 2017.

[10] "Appendix." https://www.dropbox.com/s/6mn1wp813eh3qpr/Appendix.pdf?dl=0.

[11] M. R. Gary and D. S. Johnson, “Computers and intractability: A guide to the theory of NP-completeness,” 1979.