

EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency

Jovan Stojkovic, Nikoleta Iliakopoulou, Tianyin Xu, Hubertus Franke*, Josep Torrellas

University of Illinois Urbana-Champaign *IBM Research

{jovans2,nmi4,tyxu,torrella}@illinois.edu, frankeh@ibm.com

Abstract—While serverless computing is increasingly popular, its energy and power consumption behavior is hardly explored. In this work, we perform a thorough characterization of the serverless environment and observe that it poses a set of challenges not effectively handled by existing energy-management schemes. Short serverless functions execute in opaque virtualized sandboxes, are idle for a large fraction of their invocation time, context switch frequently, and are co-located in a highly dynamic manner with many other functions of diverse properties. These features are a radical shift from more traditional application environments and require a new approach to manage energy and power.

Driven by these insights, we design *EcoFaaS*, the first energy management framework for serverless environments. *EcoFaaS* takes a user-provided end-to-end application Service Level Objective (SLO). It then splits the SLO into per-function deadlines that minimize the total energy consumption. Based on the computed deadlines, *EcoFaaS* sets the optimal per-invocation core frequency using a prediction algorithm. The algorithm performs a fine-grained analysis of the execution time of each invocation, while taking into account the specific invocation inputs. To maximize efficiency, *EcoFaaS* splits the cores in a server into multiple Core Pools, where all the cores in a pool run at the same frequency and are controlled by a single scheduler. *EcoFaaS* dynamically changes the sizes and frequencies of the pools based on the current system state. We implement *EcoFaaS* on two open-source serverless platforms (OpenWhisk and KNative) and evaluate it using diverse serverless applications. Compared to state-of-the-art energy-management systems, *EcoFaaS* reduces the total energy consumption of serverless clusters by 42% while simultaneously reducing the tail latency by 34.8%.

I. INTRODUCTION

Function-as-a-Service (FaaS) or serverless computing is emerging as a popular cloud computing paradigm that provides high developer productivity, fast on-demand scaling, and a pay-as-you-go fine-grain billing model. Users upload their code and the cloud provider secures all libraries, runtime environment, and system services needed to run it. The basic execution unit is a function, which runs in an ephemeral, stateless container or micro virtual machine created and scheduled on demand. To implement meaningful operations, serverless applications are composed of multiple functions orchestrated into an application workflow. Today, the serverless paradigm is offered by all major cloud providers (e.g., [13], [47], [60], [86]) and its popularity increases among cloud users [39].

However, serverless environments introduce a number of performance overheads. Prior work addressed some of the largest ones, such as lengthy startup latency [31], [43], [76], [99], [101], [104], [109], expensive storage accesses [70], [72], [75], [83], [98], [104], [106], [112], inter-function

communication [11], [64], [72], [82], [113], and inter-function orchestration [40], [75], [105].

One aspect that has barely been examined is the energy consumption of serverless environments. Substantial work has shown the unique performance issues of serverless computing, but how these issues translate into power/energy consumption is unknown. Hence, a framework for energy management in serverless systems is sorely needed. Advancing this area is critical, as serverless services are an increasing fraction of datacenter loads [29], [32], [95], and datacenters contribute substantially to the world energy consumption [20], [84] and carbon footprint [53], [54].

To address this shortcoming, this paper starts by performing a thorough characterization of the energy consumption in serverless environments. It shows that these environments pose a set of challenges not met by the existing energy management schemes designed for more traditional datacenter environments with long-lived applications (e.g., [33], [34], [55], [58], [67], [79], [91], [118]).

Specifically, serverless functions are highly sensitive to core frequency—much more, e.g., than to the amount of memory resources. However, because cloud providers have no visibility into the performance requirements of user functions, they always execute functions at the highest frequency, potentially wasting substantial energy. Also, as applications are comprised of multiple functions with different properties, it is hard for users to reason about how different frequency settings for individual functions affect total performance and energy.

In addition, many functions spend a considerable amount of time waiting on remote function calls or accesses to remote storage, both implemented as Remote Procedure Calls (RPCs). Hence, cores need to frequently context switch between invocations of different functions. Further, these invocations may need different optimal frequencies, but changing core frequency from virtualized sandboxes incurs significant software overheads. Finally, providers densely pack many functions with various properties on a shared server, and the mix of such co-located functions changes rapidly. Thus, the per-core frequency setting that is optimal for the performance-energy trade-off at one time may quickly become suboptimal.

Based on the insights of our characterization, we propose *EcoFaaS*, the first energy-management framework for serverless environments. *EcoFaaS* takes energy efficiency as a first-class design principle to achieve an energy-optimized serverless environment. At the same time, *EcoFaaS* ensures the end-to-

end performance target of serverless applications based on their Service Level Objectives (SLOs).

EcoFaaS achieves its goals through four main design principles. First, EcoFaaS is an SLO-driven serverless framework. In EcoFaaS, users specify the end-to-end SLO of the entire application—i.e., the maximum time that 99% of application invocations can take. EcoFaaS automatically splits the end-to-end application latency budget into per-function time budgets. Then, the functions, within their sandboxes, monitor the performance and comply with the assigned budget.

Second, EcoFaaS profiles and predicts the execution time and energy consumption of function invocations, taking into account their idle times and specific invocation inputs. Based on these predictions, EcoFaaS identifies the optimal core frequency to use for each function invocation. To take corrective actions on mispredictions, EcoFaaS continuously tracks the progress of invocations and updates the cores' frequencies as needed.

Third, to minimize the high overheads of changing core frequency, EcoFaaS dynamically splits the cores in the server into *Core Pools*. Within a pool, all cores run at the same frequency and are controlled by a single scheduler. Different pools have different core counts and use different frequencies. When EcoFaaS determines the best frequency to use for a given invocation, it assigns the invocation to the corresponding pool. In this way, EcoFaaS avoids changing frequency as much as possible while executing function invocations efficiently.

Finally, EcoFaaS makes the Core Pools *elastic*—it dynamically changes pool sizes and their frequencies. EcoFaaS supervises the system use and periodically recomputes the Core Pool set-up to adapt to the workload.

We implement EcoFaaS on top of the open-source serverless platforms OpenWhisk [1] and KNative [6], and evaluate it with a diverse set of serverless applications [22], [37], [69]. Compared to state-of-the-art energy management systems, and averaging across various system loads, EcoFaaS reduces the total energy consumption of serverless clusters by 42%, while reducing tail latency by 34.8% and increasing throughput by $1.8\times$.

This paper makes the following contributions:

- An analysis of the energy consumption of serverless systems.
- The design and implementation of EcoFaaS, the first energy-management framework for serverless environments.
- Evaluation of EcoFaaS on two open-source FaaS platforms.

II. BACKGROUND: SERVERLESS ENVIRONMENTS

Serverless platforms such as OpenWhisk, KNative, OpenFaaS, and OpenLambda [1]–[3], [6]–[8], [56], [109] are organized in a manner shown in Figure 1. The Frontend checks the integrity of function requests and forwards them to the Cluster Controller or Load Balancer. The Load Balancer monitors the state of the different nodes of the distributed machine and distributes the requests across the available nodes to ensure load balance. In each node, a Node Controller maintains a pool of available containers and is in charge of function scheduling. All requests forwarded to a node are handled by the Node Controller. When the resources needed

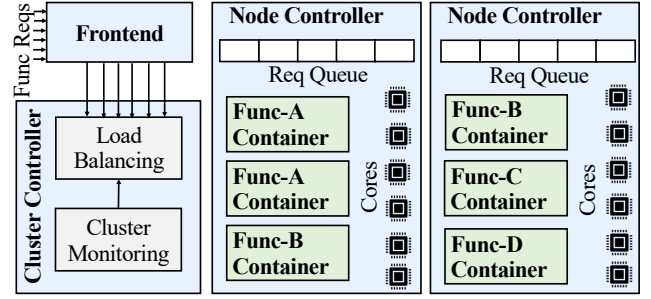


Fig. 1: Overview of existing serverless platforms.

for a function request (e.g., a number of cores) are available, the Node Controller encapsulates the function code with all dependencies inside a container, and then invokes the function's handler on a core.

One of the main attractions of serverless computing is its cost effectiveness [14], [89]. Users are charged only for the resources that their functions use, and once the function becomes inactive, it is unloaded from memory. This creates a highly dynamic environment, where providers densely pack functions on servers, and the mix of functions concurrently running on a server changes rapidly. Such functions can have very different properties. For example, functions that train a machine learning model and require many CPU cycles [18], [19] can run concurrently with I/O-bound web services [16].

These functions are only the building blocks for the end-to-end applications. Applications are composed of multiple functions connected with each other in a workflow constructed via composition frameworks such as AWS Step Functions [15], Azure Durable Functions [88], IBM Cloud Composer [59], or Google Cloud Workflows [50].

III. ENERGY-EFFICIENT SERVERLESS ENVIRONMENTS

To understand the unique energy-efficiency challenges in serverless environments, we characterize real-world serverless workloads [24], [116] and open-source applications [9], [69] on an Intel Haswell E5-2660 server. Section VII describes our methodology, including the datasets and applications in detail. Based on our observations, we propose the following recommendations to attain highly energy-efficient serverless computing environments.

1. Serverless environments should be SLO-driven. We characterize the execution time and energy consumption of serverless functions at different core frequencies, ranging from 1.2GHz to 3GHz. We show their response time (Figure 2a) and energy consumption (Figure 2b). Consider the execution at 3GHz, and set the SLO to a few times the function execution time, as it is commonly done [41], [90]. We can see that many functions can be executed at substantially lower frequencies without violating such SLO, while saving substantial energy. For example, running CNNServe at 2GHz rather than at 3GHz increases its response time by 23% while reducing its energy consumption by 40%. As another example, running WebServe at 1.2GHz rather than at 3GHz increases response time by only 12% while reducing energy consumption by 47%.

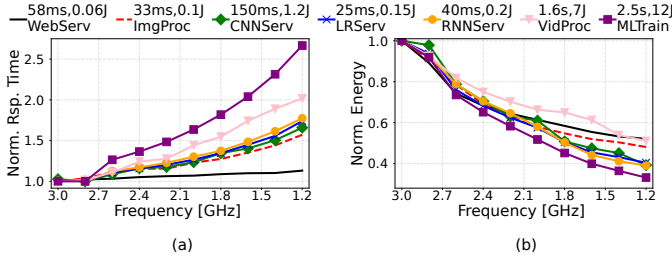


Fig. 2: Normalized (a) response time, and (b) energy consumption of serverless functions with different core frequencies. The numbers above the legend show the execution time and energy consumption per function execution at 3GHz core frequency.

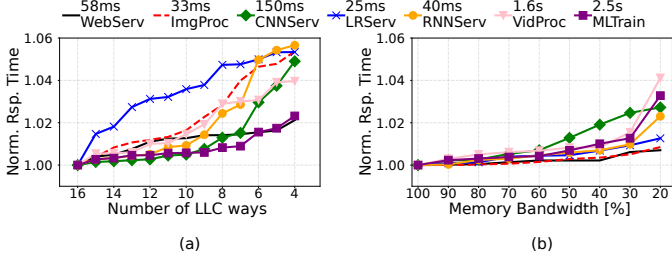


Fig. 3: Normalized response time of serverless functions at 3GHz with different (a) number of LLC ways, and (b) memory bandwidth. The numbers above the legend show the function execution time with 16 LLC ways and 100% memory bandwidth.

Furthermore, functions are typically single-threaded and have relatively small memory footprints. Hence, they do not benefit from extra cores and benefit little from more memory resources. For example, Figure 3 shows the normalized response time when executing a function at the highest frequency of 3GHz with different (a) number of LLC ways and (b) percentage of memory bandwidth, both controlled by the `pqos` tool [4]. The latencies are normalized to the setup with the maximum amount of resources (16 LLC ways or 100% memory bandwidth). We see that, when using 4 LLC ways or 20% memory bandwidth, the response time increases *at most* by 6% and 4%, respectively. Hence, the core frequency is the main controllable knob that affects a function’s performance and energy consumption.

Unfortunately, as cloud providers have no visibility into the performance requirements of user functions, they constantly operate at the highest frequencies, unnecessarily consuming energy. Furthermore, the energy-performance trade-offs are complex, as serverless applications are composed of multiple functions chained together into an application workflow. Different functions can have different latency and energy consumption profiles. Thus, it is hard for end users to reason about an execution plan that minimizes the energy consumption while satisfying the application’s performance requirements.

We believe that, to substantially increase the energy efficiency of serverless environments, users need to specify the overall performance expectations of their applications, commonly expressed as SLOs. Then, the platform should automatically and transparently optimize the execution of each function of the application for overall minimal energy consumption while operating within the performance constraints.

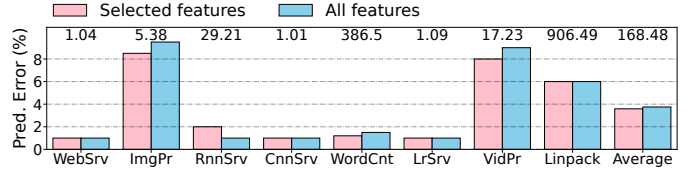


Fig. 4: Prediction error of function execution time when functions are called with inputs not seen during training. The numbers on top of the bars are the ratio of longest to shortest execution time.

2. Serverless environments should be input-aware. The execution time of serverless functions can depend on the functions’ inputs. To understand this effect, we execute our functions with various inputs from large open-source datasets [42], [65], [73], [81], [94]. After profiling over 100 open-source serverless functions [22], [37], [69], [108], [114], we observe that, typically, the execution time as a function of the inputs is either constant or can be approximated with simple polynomial functions. Consequently, like prior work [34], we extract high-level features from inputs, such as the size of the file, the duration of the video, or the resolution of the image, and train a simple three-layer neural network. After that, when a new input is received with certain values for the high-level features, the network estimates the execution time of the function with the new input.

In addition, in another set of experiments, we train the model with *all* the inputs of a function—not just those that appear to impact the execution time. With this approach, developers do not have to specify which are the important input features.

Figure 4 shows, for each function, the prediction error of our models, defined as $(|E - A|)/A$, where E is the estimated execution time and A the actual one. Each function has two bars, one for the model trained with selected features and one for the model trained with all the features. On top of the bars, we show the ratio of the longest to the shortest execution time for individual functions. We see that, although functions have a large range of execution times, their execution is highly predictable. On average, the error is only 3.6% when the model is trained with selected input features and only 3.8% when it is trained with all the input features. Therefore, we train our model with all the features and keep the prediction accuracy high while reducing the burden on developers.

3. Serverless environments should exploit the substantial idle time within function invocations. Function execution time is short, ranging from a millisecond to a few seconds [101], [111]. Even during such a short time, functions are mostly idle, waiting for RPC responses from remote functions or remote storage. In our applications, functions accessing data from remote storage commonly spend 70% of their execution time idling. Thus, to improve resource utilization and keep cores busy, advanced serverless systems perform a context switch when a core stalls. As a result, context switches are typically as frequent as one every few hundreds of μs [64], [100], [104].

However, state-of-the-art energy-management frameworks for traditional applications are designed with a run-to-completion model [34], [58], [67], [118]. As an example,

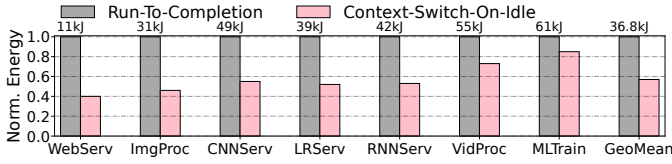


Fig. 5: Normalized total energy consumption of functions averaged across loads when executed under *Run-To-Completion* or *Context-Switch-On-Idle*. The numbers on top of the bars show the total energy consumption for 10-minute runs in kJ.

Gemini [118] maintains a FIFO request queue and detects the arrival of a request whose deadline will be missed if executing at the current frequency. Then, it executes all queued requests, one at a time, at a higher frequency to meet the critical request’s deadline. The system relies on the run-to-completion model to predict the queuing time for each incoming request. This approach works well for applications that are not I/O bound. However, it can be detrimental for the performance and energy efficiency of serverless environments with many I/O-bound functions. With the run-to-completion model, as the request queue builds up, the invocations execute at higher frequencies, while wasting the time that functions spend being blocked.

We measure the energy consumption needed to meet the SLOs in two environments: *Run-To-Completion* and *Context-Switch-on-Idle*. The latter uses the idle time of an invocation to run another ready-to-run invocation. We execute our functions at different loads, varying the request inter-arrival time with a Poisson distribution. We set the SLO of a function to be $5 \times$ its execution time on an unloaded system.

Figure 5 shows the total energy consumption averaged across all loads when running each function in either environment. Context-Switch-on-Idle allows for more invocations to execute at lower frequencies. Hence, as we see in the figure, it reduces the energy consumption by 42.3%. As the idle time within invocations increases and as the load gets higher, the savings become more substantial. It can be shown that Context-Switch-on-Idle also improves the performance of serverless functions, especially when the load is high. For example, on average across all functions in high load, the average and tail latencies are reduced by 48.2% and 67.4%, respectively.

4. Serverless environments should minimize the number of core frequency changes. To exploit the idle time within an invocation, when a core becomes idle, it should context switch to another function invocation. This new invocation may have a different optimal frequency. Thus, at every context switch, the system might need to change the core’s frequency.

Such changes are expensive. Recall that a server can be running serverless functions from different users, and they are isolated from each other in different VMs [10], [116] or containers [1], [6], [49]. To change the core’s frequency, sandboxed serverless functions need to communicate with the host and cross the OS kernel boundary (i.e., switch between user and kernel spaces). Consequently, even though the hardware overheads of changing the core frequency are only a few tens of microseconds, our results show that, in Linux-based systems with an ACPI frequency driver, the overhead of changing

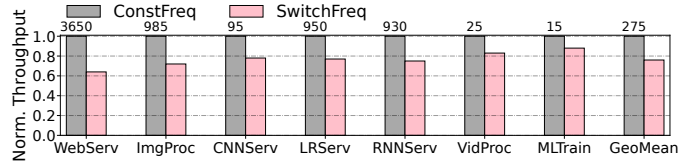


Fig. 6: Normalized function throughput (higher is better) when executed in *ConstFreq* and *SwitchFreq*. The numbers on top of the bars show the *ConstFreq* throughput in Requests-Per-Second (RPS).

core frequencies from userspace processes or containers is 10-20ms. This overhead is of the same order of magnitude as the execution time of a serverless function.

To see this effect, we execute our functions in two environments. First, in *ConstFreq*, we keep the frequency constant at 2.5GHz throughout the whole run. Second, in *SwitchFreq*, we keep setting the frequency to 2.5GHz at every context switch. Thus, the two environments execute under the same conditions; the only difference is the overhead of invoking the kernel to set the core’s frequency in the second case. Figure 6 shows the throughput of the functions in the two environments. We see that the overhead of changing the core’s frequency from the virtualized sandboxes can significantly reduce system throughput. For short functions like WebServ, the throughput losses are higher. On average across all functions, *SwitchFreq* reduces the throughput of *ConstFreq* by 24.1%. Hence, for energy efficiency, one should minimize frequency changes.

5. Serverless environments should dynamically adapt to workload changes. An intuitive way to minimize the number of core frequency changes is to dedicate some cores to each frequency level. Functions with similar frequency needs can be grouped together into classes and better utilize the shared cores while minimizing core frequency changes. The key challenge is to address serverless workload dynamics: the workload is ephemeral, functions are frequently loaded/unloaded from memory, and their loads continuously fluctuate [101], [111].

We analyze open-source production-level traces from Azure Functions [116] to understand the co-location of different functions. Figure 7 shows the CDF of the number of different functions executed in a small cluster within 1 second, 10 seconds, 1 minute, and 10 minutes. We see that, within a second, the system executes on average 3 different functions, but it may execute up to 36 different functions. Within 10 seconds, it may execute up to 52 different functions. This is in contrast to the traditional VM-based cloud environments, where only a few long-lived VMs share the cluster for long durations. This shows that the optimal allotment of cores to function classes at a given time can rapidly become suboptimal.

IV. LIMITATIONS OF CURRENT ART

Researchers have proposed energy-management frameworks for systems running long-lived applications. For example, Pegasus [79] achieves energy proportionality by setting the power limit of the entire server to be able to meet, but not exceed, the SLO of the application running on that server. EETL [55] places requests on slow cores, and reschedules them to fast cores when it predicts that the SLO will not be

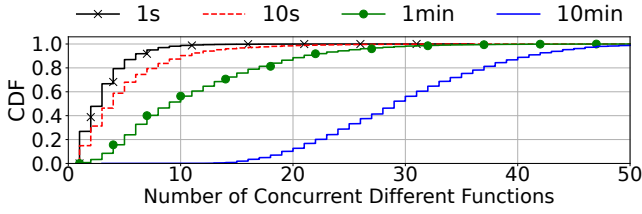


Fig. 7: CDF of the number of *different* functions executed in a small cluster within 1s, 10s, 1min, and 10min in Azure Functions.

met. Adrenaline [58] predicts long requests before scheduling them on cores and boosts their frequency from the beginning of their execution. ReTail [34], Rubik [67] and Gemini [118] take one step further and predict the optimal frequency for every request individually. NCAP [12] and NMAP [66] proactively transition a processor to an appropriate performance or sleep state based on the rate of received and transmitted network packets containing latency critical requests. These schemes are effective for latency-critical monolithic applications or backend services. Unfortunately, they are not a good fit for serverless environments.

First, these schemes target a single service or a monolithic application, while serverless applications are workflows of many functions glued together. As these functions may execute on different machines, have different performance and energy profiles, and execute in highly dynamic environments, it is non-trivial to specify individual function deadlines, while keeping end-to-end application target response time satisfied.

Second, these schemes rely on a *run-to-completion* model when determining the per-application or per-request optimal frequency. They assume that the running request has to finish before a new request can be scheduled for execution [34], [67], [118]. The run-to-completion model is a reasonable assumption for applications that keep their cores busy throughout most of the execution. However, it is energy inefficient for the typical serverless function, which is mostly idle waiting on I/O. Researchers have explored techniques to efficiently coordinate sleep states with frequency states during idle periods [35], [78]. However, these works assume server-wide idleness. In a serverless setup, while a single invocation experiences idle time, there are other invocations waiting in the queue ready to execute. Thus, if the core refuses to execute waiting invocations and goes to sleep, it will need to execute these invocations later at a higher frequency, to compensate for their longer queuing.

Third, these schemes assume a negligible cost for frequency changes. The assumption is true for long-lived applications with dedicated cores and infrequent context switches. However, in serverless environments: (i) cores frequently context switch between invocations of the same or different functions, and (ii) functions run in virtualized sandboxes, unable to directly change the core frequency without contacting the host.

Lastly, these schemes are mainly designed for server machines running a single long-lived application [34], [118], a latency-critical application co-located with a batch application [67], or a few co-located applications using exclusively partitioned resources [33], [91]. Serverless environments have

a very different resource model: many functions with various performance requirements can concurrently execute on the same machine, and the machine is typically overprovisioned with more functions than available cores [10].

On the other hand, state-of-the-art serverless systems are not optimized for energy efficiency. They improve performance by (i) minimizing cold start overheads [31], [43], [45], [92], [101], [102], [109], (ii) improving resource utilization [11], [27], [51], [64], [72], [104], (iii) reducing the cost of remote storage access [70], [82], [98], [104], and (iv) proactively scheduling or executing functions of an application's chain [30], [40], [105]. They do not consider energy consumption in their designs.

V. ECOFAAS OVERVIEW

Based on our recommendations in Section III, this section presents *EcoFaaS*, the first energy-management framework for serverless environments. *EcoFaaS* is based on four main ideas.

1. EcoFaaS is driven by SLO metrics. In existing serverless platforms, to meet the expected performance, end users specify the number of cores or the amount of memory devoted to the function [17], [48], [71], [87]. Unfortunately, it is not easy for users to reason about the exact impact of these resources on the performance and energy efficiency of functions [28]. Furthermore, applications are composed of multiple functions. Thus, specifying these resources for individual functions makes the end-to-end outcome even more opaque. As a result, cloud providers simply execute all functions with the requested cores or memory at the highest core frequency.

EcoFaaS argues for a different environment, where end users only specify the end-to-end SLO of the entire application—i.e., the maximum time that 99% of the application invocations can take. This approach enables the end user to provide an accurate specification of what they need, without increasing the user's specification burden. As importantly, it enables cloud providers to optimize execution for energy efficiency. Specifically, *EcoFaaS* automatically splits the SLO of an application into per-function time budgets, and then dynamically sets the frequency of cores executing individual functions based on the current system conditions. The goal is to execute functions at their most energy-efficient frequencies while satisfying the SLO of the end-to-end application.

2. EcoFaaS profiles and predicts the execution time and energy of function invocations. During execution, *EcoFaaS* profiles the execution time and energy consumption of functions at different core frequencies. It also takes into account variations due to different function inputs. The execution time is broken down into: 1) execution on a core (T_{Run}), 2) I/O blocking due to RPCs (T_{Block}), and 3) waiting in a queue to be processed (T_{Queue}). Note that T_{Block} and T_{Queue} are substantial and often comparable to T_{Run} . *EcoFaaS* uses values of profiled T_{Run} and T_{Block} , and values of predicted T_{Queue} based on current system conditions, to select the core frequency for each function that leads to the most energy-efficient application execution while satisfying the SLO of the entire application.

3. EcoFaaS splits cores into frequency classes. For highest energy efficiency, different functions and even different

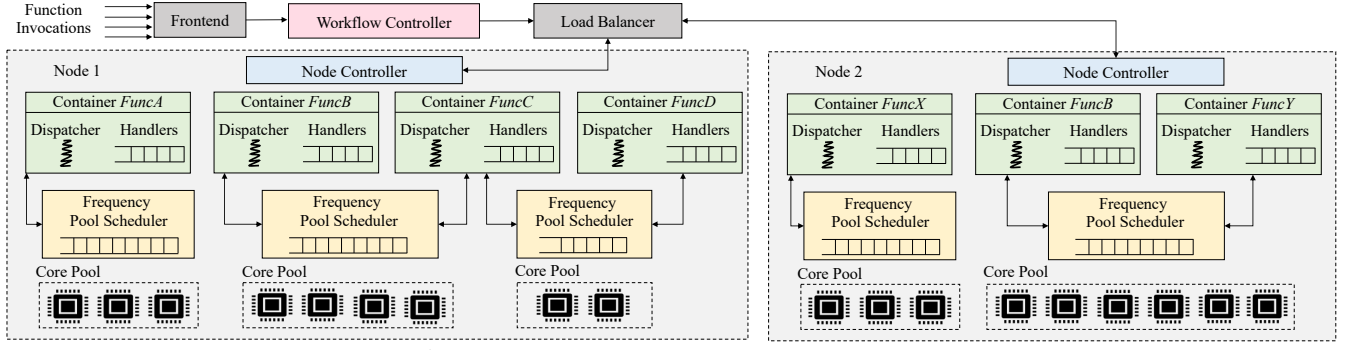


Fig. 8: Overview of an EcoFaaS serverless platform.

invocations of the same function may need to run at different frequencies. However, functions typically have millisecond-level execution times and include blocking events that cause context switches. Since changing core frequency at every context switch is inefficient, EcoFaaS avoids changing frequency as much as possible. It keeps *Core Pools* running at different frequency levels and tries to assign the execution of individual function invocations to the pool that has a frequency equal or slightly higher than the invocation's optimal frequency. Each pool is controlled by a *Frequency Pool Scheduler* (FPS).

4. EcoFaaS changes pools and pool frequencies dynamically.

In serverless environments, the functions being invoked and their popularity change over time. A partition of cores into frequency pools that was optimal at a given time may quickly become suboptimal. Hence, EcoFaaS supervises the system use and periodically recomputes the assignment of cores to pools and the frequency of each pool. The goal is to execute the functions in the most energy-efficient manner while capturing the dynamics of the workloads executing.

VI. ECOFAAS DESIGN

Figure 8 shows the organization of an EcoFaaS serverless platform. Like existing schemes depicted in Figure 1 [1], [6], [29], EcoFaaS has Frontend and Load Balancer modules, and per-node Node Controllers. On top of these, EcoFaaS adds a *Workflow Controller* per application and, inside each node, a *Function Dispatcher* per function container, *Core Pools*, and a *Frequency Pool Scheduler* per pool. Except for the core pools, all these structures are software structures.

The Workflow Controller maintains the SLO of the application and calculates the execution deadline for each function of the application to attain maximum energy efficiency. Then, it sends the computed deadlines for each function to the corresponding Function Dispatchers.

A Function Dispatcher manages a function container. It receives requests for its function, forks a *Handler* process to execute each request, and selects the optimal frequency to run the requests based on the function profile and the execution deadline. Additionally, each Function Dispatcher profiles the execution of the invocations of its function and, every T_{update} , sends the profile to the corresponding Workflow Controller to update the optimal per-function deadlines.

Based on the chosen frequency, the Function Dispatcher registers the function invocation to execute in a specific Core Pool. The cores in that pool run at a selected frequency and are managed by a *user-space* Frequency Pool Scheduler (FPS). When a core in that pool becomes available, the function invocation is scheduled for execution; when the execution blocks due to an RPC, another invocation registered with the same pool (from the same or a different function) is scheduled.

The Node Controller periodically collects runtime metrics from every FPS, including the number of queued and served invocations, and invocations that could have been executed at lower frequencies if an appropriate core pool had been available. Based on this information, the Node Controller adjusts the allocation of cores to pools and the frequency of each pool.

A. SLO-Aware Workflow Controller

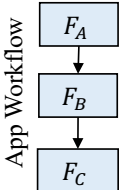
When a user submits an application invocation to EcoFaaS, they specify the end-to-end SLO for the application. Then, EcoFaaS automatically and transparently splits the application's SLO into the optimal per-function deadlines. The response time and energy consumption of different functions in the application may have different sensitivity to core frequency. The optimal execution is the one that minimizes the combined energy consumption of all the functions of the application while finishing the application within the SLO.

To achieve this, the Workflow Controller of an application uses profile data regularly provided by the Function Dispatchers of the constituting functions. Such information is kept in a per-application software structure called *Delay-Power Table* (DPT). The DPT has an entry for each function F_i of the application running at each of the possible frequencies f_j . The entry contains the predicted execution time $t_{f_j}^{F_i}$ of the function (equal to $T_{Run} + T_{Block} + T_{Queue}$) and the predicted energy consumption $E_{f_j}^{F_i}$. Then, the Workflow Controller uses Mixed-Integer Linear Programming (MILP) [85] to pick the frequency for each function that minimizes $\sum E_{f_j}^{F_i}$ under the constraint $\sum t_{f_j}^{F_i} \leq SLO$. MILP is a suitable technique as both objective function and constraint are linear relations and the calculations support FP numbers.

Figure 9 shows the DPT for an application composed of functions F_A , F_B , and F_C , and supporting frequencies f_1 , f_2 , and f_3 . The Workflow Controller determines the optimal frequency of each individual function and the resulting deadline

of each function. For example, Figure 9 shows with tick marks the optimal frequencies of each function. Based on the figure, F_B 's optimal frequency is f_1 , and F_B 's deadline is $t_{f_2}^{F_A} + t_{f_1}^{F_B}$.

App Workflow



Delay-Power Table

	F_A	F_B	F_C
f_1	$t_{f_1}^{F_A}, E_{f_1}^{F_A}$	✓ $t_{f_1}^{F_B}, E_{f_1}^{F_B}$	$t_{f_1}^{F_C}, E_{f_1}^{F_C}$
f_2	✓ $t_{f_2}^{F_A}, E_{f_2}^{F_A}$	$t_{f_2}^{F_B}, E_{f_2}^{F_B}$	$t_{f_2}^{F_C}, E_{f_2}^{F_C}$
f_3	$t_{f_3}^{F_A}, E_{f_3}^{F_A}$	$t_{f_3}^{F_B}, E_{f_3}^{F_B}$	✓ $t_{f_3}^{F_C}, E_{f_3}^{F_C}$

Fig. 9: Organization of the Delay-Power Table for one application. Check marks show the chosen frequencies for each function.

This approach handles cases where a function invokes multiple parallel children functions. The workflow controller assigns a deadline to the group of parallel functions based on the slowest function in the group.

B. Energy-Aware Function Dispatcher

When a container receives a function invocation, the dispatcher in the container creates a handler to execute the invocation. To execute the invocation in the most energy-efficient manner, the dispatcher uses: (i) information provided by the Workflow Controller that is included in the function invocation message and (ii) profiling information on the function that the dispatcher has gathered from past executions.

The function invocation message includes the deadline for executing the function in absolute time, as computed by the Workflow Controller. It does not include the recommended execution frequency. The reason is that the current execution environment may be different from the one used by the Workflow Controller to estimate the optimal computation.

To see why, consider our running example of the application with functions F_A , F_B , and F_C , and assume that we are at a point when we want to run F_B . Figure 10a shows the timeline of the optimal execution computed by the Workload Controller, where F_A 's deadline is t_A , F_B 's is t_B , and F_C 's is t_C . In practice, assume that the execution of F_A was faster (e.g., the load was low and there was little queuing) and took only t'_A (Figure 10b). Now, given F_B 's unchanged deadline, the dispatcher for F_B can pick a lower frequency.

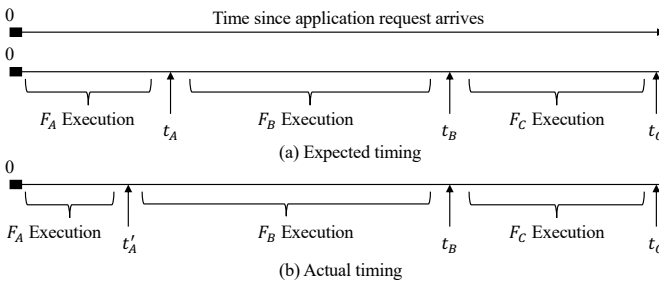


Fig. 10: Timeline of a three-function application execution.

To pick the best frequency at which to run F_B , the dispatcher examines the history of past executions of F_B . The dispatcher has been profiling prior executions of the function and stored

the profile in a software structure for that function called *History Table*. Figure 11 shows the design of the table. It contains T_{Run} , T_{Block} , and $Energy$ for the few most recent executions of the function. Recall that the execution time of the function is $T_{Run} + T_{Block} + T_{Queue}$, while $Energy$ is the energy consumed during T_{Run} . Since T_{Run} depends on the frequency, both T_{Run} and $Energy$ have entries for multiple frequencies. This is not the case for T_{Block} . The History Table is continuously updated: every time a handler executes the function, it measures and saves T_{Run} , T_{Block} , and $Energy$.

Frequency	T_{Run} History	Energy History	T_{Block} History
f_1	$T_{Run}^1, T_{Run}^1, T_{Run}^1$	E^1, E^1, E^1	$T_{Block}, T_{Block}, T_{Block}$
f_2	$T_{Run}^2, T_{Run}^2, T_{Run}^2$	E^2, E^2, E^2	

Fig. 11: The History Table in a dispatcher.

With this information, the dispatcher can estimate the expected T_{Run} , T_{Block} , and $Energy$ for different frequencies. We propose two different approaches. The first, simpler one, is for the dispatcher to use the exponentially weighted moving average (EWMA) [80] of these three parameters. EWMA assigns higher weights to more recent measurements, and uses adaptive smoothing with the Holt-Winters method [57] to dynamically adjust a parameter α based on the changes in the system state.

A better, more advanced approach is to also record in the History Table the inputs of the function for each invocation, and then use a machine learning model to estimate the expected T_{Run} , T_{Block} , and $Energy$ for different frequencies. We discuss this design in Section VI-E2.

After the dispatcher estimates the expected T_{Run} , T_{Block} , and $Energy$ for each frequency, it estimates T_{Queue} . For this, the dispatcher examines the number of waiting jobs in the queues of the Core Pools in the server. This is enough to estimate T_{Queue} , given that we use: 1) FIFO queueing, 2) pre-emption of a younger job when an older job is ready to run, and 3) a user-space scheduler whose scheduling overhead is largely negligible. Finally, with all this information, the dispatcher picks the frequency with the lowest $Energy$ that still satisfies $t'_A + T_{Run} + T_{Block} + T_{Queue} \leq t_B$, where t_B is the deadline to complete F_B according to the Workload Controller.

When a function is unloaded from the system, its History Table is saved as part of the function's context. It is then used as a starting point when a new instance of the function is created in the future, hence avoiding cold start effects. When there is no prior knowledge about a function's execution, the dispatcher picks the highest possible frequency.

C. Core Pools and Frequency Pool Schedulers (FPS)

As shown in Section III-3, cores in serverless environments experience context switches as frequently as one every few hundreds of μs . If, in EcoFaaS' quest to execute invocations at optimal frequencies, EcoFaaS had to change the frequency of a core at every context switch, the overhead would be intolerable. Indeed, changing the frequency takes about 10 μs

without software overhead and often 1000x more with software overhead (Section III-4). Therefore, EcoFaaS faces a tradeoff between fine-tuning the core frequency to attain higher energy efficiency and not doing it to reduce overhead.

In practice, serverless workloads are often bursty: the same function is invoked many times in a short period. When this happens, it offers the ability to reuse the core frequency across function invocations. Also, different functions co-located on the same server may have the same frequency requirements. For these reasons, and because cores typically offer a limited number of frequency levels, EcoFaaS organizes the cores in a server into dynamic *Core Pools*. The cores in a pool run at the same frequency, while different pools run at different frequencies. Core pools dynamically change the number of cores and the frequency based on application demands.

When a Function Dispatcher has estimated that a function should optimally run at a given frequency, the dispatcher searches for and picks a Core Pool that is running at the same or slightly higher frequency. Since different invocations of the same function may have different optimal frequencies, the Function Dispatcher can register different invocations of the function with different Core Pools.

A Core Pool is managed by a *Frequency Pool Scheduler* (FPS) that schedules function invocations on the cores. As indicated before, the FPS is user-level, uses FIFO, allows a ready-to-run older job to immediately preempt a younger job, and has negligible scheduling overhead.

Recall that EcoFaaS estimates T_{Queue} before deciding the optimal core frequency to use to run a function. To make it easy to do so, each FPS maintains *Estimated Wait Time (EWT)* counters that give the wait time for executions at the current and all higher frequency levels. An EWT counter contains the sum of the expected T_{Run} of all the queued and running jobs. When a new invocation is added to the queue, the EWT counter is incremented by the invocation's estimated T_{Run} ; when an invocation completes, the EWT counter is decremented by the invocation's T_{Run} . An EWT counter divided by the number of cores in the Core Pool estimates T_{Queue} .

With FPSs, user VMs or containers use a clean interface to register invocations for execution at different frequencies—while maintaining their black-box property. Cloud providers do not have, and do not need, visibility into the code of users' functions. Instead, Function Dispatchers perform all the profiling, monitoring, and optimal frequency calculations inside the sandboxed VMs or containers. Then, the Function Dispatchers register the invocations with the chosen Core Pool, and give to the pool's FPS only the estimated time that the invocation will spend running on a core.

D. Elastic Core Pools

Due to the dynamic nature of serverless environments, the number of cores in each pool and their frequency is unlikely to be optimal or even usable for more than a short time. Consequently, EcoFaaS dynamically changes the Core Pools.

Consider first the case when a function invocation I_0 may be unable to meet its deadline in any of the Core Pools. In

this case, the corresponding dispatcher checks if the deadline can be met in one of the pools by temporarily increasing the frequency. Specifically, the dispatcher first looks for a pool where I_0 's deadline is met if the existing frequency is kept for the currently-queued jobs, and the frequency is temporarily boosted only when it comes to I_0 's turn to execute. If such pool exists, this strategy is used. Otherwise, the dispatcher looks for a pool where I_0 's deadline is met if the frequency is temporarily raised for both the currently queued jobs and I_0 . If such pool exists, this second strategy is used. Otherwise, I_0 's deadline will likely be missed, but the dispatcher still picks the queue with the shortest T'_{Queue} (defined as the predicted queuing time at the highest possible frequency) and the system increases the frequency of all the queued jobs and I_0 to the maximum possible value.

Consider now the case when the workload changes substantially. For example, initially, there are many invocations choosing frequency f_1 and few choosing f_2 ; then, suddenly, frequency f_2 becomes popular while f_1 is rarely needed.

EcoFaaS tackles this challenge by periodically reassigning cores among pools and changing the frequency of pools. Specifically, during each $T_{refresh}$ interval, the FPS in each pool records the number of served invocations, the average waiting time in the queue, the number of invocations that could have been executed at a lower frequency (but found no appropriate pool), and the number of invocations that required temporary increases in frequency to meet deadlines. Then, at the end of the interval, all FPSs send their collected information to the Node Controller.

The Node Controller uses this information to apportion cores to pools, and to set the frequency of the pools for the next interval. The process is as follows. Each pool i is assigned a weight W_i . Pools that served more requests or had longer waiting times receive higher weights. After that, the Node Controller assigns a number of cores N_i out of the total N_{total} to pool i based on $N_i = \frac{W_i * N_{total}}{\sum W_i}$. Then, those pools that often had to temporarily increase their frequency to meet invocations' deadlines are assigned the next higher frequency level, while pools that often took invocations that could have executed at a lower frequency are assigned the next lower frequency level.

E. Improving the Robustness of EcoFaaS

We improve the robustness of EcoFaaS in three ways.

1) *Cold starts*: When an invocation experiences a cold start (i.e., there is no warm VM/container), the system starts-up the VM/container and initializes the function before executing the invocation. Despite recent advances that minimize this cost [5], [64], [101], [104], [109], the duration of a cold start is, at best, of the same order of magnitude as the function's execution. If the cold start is on the critical path, EcoFaaS executes both cold start and the function's handler at a high core frequency, substantially degrading energy efficiency. Otherwise, the Workflow Controller tries to execute the cold start early, off the critical path, and at lower frequencies. The process is as follows.

The controller checks if any function in the application has no available container in the cluster and, hence, will require a cold start. If such a function exists, the controller *prewarms* the container at a lower frequency, in the background, while the predecessor functions in the application’s chain are being executed. If there are multiple functions that will experience cold starts, their containers are prewarmed in parallel.

To determine the frequency to use in cold starts of the function, EcoFaaS prewarms the container at different frequencies in different cold starts of the function, and populates the Delay-Power Table of the function. After that, in subsequent cold starts of the function, EcoFaaS picks the minimal core frequency that can complete the cold start within the sum of the predecessor functions’ deadlines. For example, the cold start of F_c in Figure 9 has to be completed in $t_{f_2}^{FA} + t_{f_1}^{FB}$. In this way, when F_c executes, it will not suffer a cold start.

2) *Function Input Sensitivity*: As indicated in Section III-2, the execution time of some functions varies with the function inputs, but these variations are highly predictable. Hence, similar to prior work [34], EcoFaaS uses a machine learning (ML) model to estimate the expected T_{Run} , T_{Block} , and *Energy* of a function invocation. As indicated in Section VI-B, every time that a handler executes the function, it measures and saves T_{Run} , T_{Block} , *Energy*, and the function inputs in the corresponding History Table. Then, when a new invocation of the function is received (with potentially new inputs), the Function Dispatcher uses the ML model to estimate T_{Run} , T_{Block} , and *Energy* for different frequencies. The model we use is lightweight, has three fully connected (linear) layers and ReLU activations, and takes the features of all the inputs of the function, to eliminate any annotation burden on developers. The model is trained online using live traffic of function invocations. As indicated in Section III-2, its prediction error is less than 4%, while its overhead is only a few tens of μ s.

3) *Heterogeneous Servers*: While datacenters strongly favor machine homogeneity [25], [110], many of them have heterogeneous servers. In this case, the Delay-Power Table (DPT) generated for one server type (e.g., Intel Haswell [63]) cannot be directly reused by another server type (e.g., Intel Skylake [61]). Hence, EcoFaaS needs to profile the functions on all server types. Note that, on average, only a few tens of application invocations are needed to populate the DPT of the functions in the application for each type of server. During most of these invocations, the application maintains good performance, although its energy efficiency is suboptimal. Since applications are invoked thousands of times a day [24], [111], the impact of this profiling period is largely negligible.

However, to minimize this inefficiency, EcoFaaS uses transfer ML techniques. Given the profiles (execution time and energy consumption) of functions on machine *A* and a small subset of function profiles on machine *B*, transfer ML generates a model that predicts the profiles of the rest of the functions on *B*. In EcoFaaS, we train and test a simple Linear Regression model that performs transfer ML from Intel Haswell [63] servers to Broadwell [62] and Skylake [61] servers. By using only 1/4 of the samples from the last two machines, the model achieves

an accuracy of 93.1%.

VII. EVALUATION METHODOLOGY

Evaluation environment. We evaluate EcoFaaS in the state-of-the-art MXFaaS [104] serverless platform, on top of OpenWhisk [1] and KNative [6]. In this paper, we discuss only the results with OpenWhisk, as the results with KNative are similar. In our experiments, we vary the number of servers in a cluster from 5 to 20. Each server is an Intel Haswell E5-2660 v3 with 20 cores organized in two sockets, with 160GB of DRAM, a 50MB LLC, and running Ubuntu 22.04.2 LTS.

We use the ACPI frequency driver with the “userspace” governor to allow 7 user-defined frequency settings ranging from 1.2GHz to 3.0GHz in 0.3GHz increments. We measure the energy consumption of each server with CPU Energy Meter [26]. This includes the energy consumed by the package and DRAM for both sockets running the whole software stack. Similar to prior research [36], [44], [46], [68], [115], we use power modeling to apportion the overall socket power to individual cores. The model takes into account the core’s frequency, the number of active core cycles (stall cycles are excluded), and a few performance counters.

For the configurable parameters in EcoFaaS, we perform sensitivity studies and pick the following values: (i) keep the last 100 invocations in History Tables, (ii) update the Workflow Controller’s Delay-Power Table every 5s (T_{update}), and (iii) update the pools’ sizes and frequencies every 2s ($T_{refresh}$).

Evaluated functions and applications. We use functions from FunctionBench [69], a widely-used suite in serverless research [21], [45], [74], [109], [116], [117]. The functions include ML training and model serving, image/video processing, and web services. In addition, we use a set of real-world serverless applications from AWS Samples [22], Serverless-Bench [114] and vSwarm [108]. The applications include ML workflow, data analytics, online banking and booking, and video streaming. The evaluated benchmarks are summarized in the Table I. We use Azure Blob Storage [23] as the storage service for all the functions and invoke functions with inputs from open-source datasets [42], [65], [73], [81], [94]. Like prior work [41], [90], we set the SLO of an application to 5× the application’s warm latency on an unloaded system at the highest frequency.

TABLE I: Serverless benchmarks used in the evaluation.

Benchmark	Description
Standalone Functions (all from FunctionBench [69])	
WebServ	Processing JSON file fetched from the storage
ImgProc	Image processing: Resize image
CNNServ	ML model serving: CNN-based image classification
LRServ	ML model serving: Logistic regression
RNNServ	ML model serving: RNN-based word generation
VidProc	Video processing: Apply gray-scale effect
MLTrain	ML model training: Logistic regression
Serverless Applications	
MLTune [19]	Tuning an ML model (6 functions)
DataAn [114]	Wage-data analysis workload (8 functions)
eBank [22]	Withdraw money from an account (6 functions)
eBook [108]	A hotel reservation service (7 functions)
VidAn [108]	A video analysis system (3 functions)

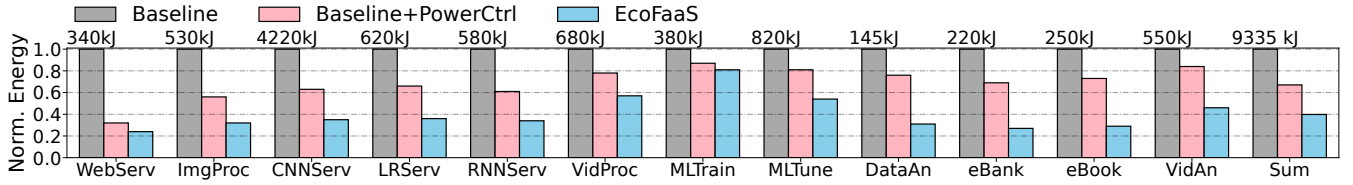


Fig. 12: Normalized energy consumption of *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* with real-world invocation traces. The numbers on top of the *Baseline* bars show the absolute energy consumption of *Baseline* for the 6-hour run in all 5 servers.

We first evaluate these benchmarks using the invocation patterns from the open-source production-level traces of Azure Functions [116]. Then, we vary the load using a Poisson distribution to model the request inter-arrival time [11], [27], [51], [100], [103], [107], [116]. We generate low, medium and high loads corresponding to CPU utilizations of about 25%, 50%, and 70%, which are representative [38], [52], [77], [97].

Baselines. We compare *EcoFaaS* to two advanced baseline frameworks: *Baseline* and *Baseline+PowerCtrl*. *Baseline* is the state-of-the-art MxFaaS [104] serverless platform. It assigns a set of cores to each function container, allows invocations of a function to be scheduled only on cores owned by the function, and runs all invocations at the highest frequency.

Baseline+PowerCtrl is *Baseline* plus a state-of-the-art energy-management framework for long-lived applications based on Gemini [118]. This framework saves energy by setting the frequency for a function invocation based on the function’s deadline. The framework assumes a run-to-completion model and, on a context switch, it changes the core’s frequency only if the predicted best frequency of the next request differs from the current core frequency. *Baseline+PowerCtrl* is an upper-bound of schemes such as Gemini because it predicts an invocation’s execution time at a given frequency with 100% accuracy. For applications with multiple functions, *Baseline+PowerCtrl* distributes the application’s SLO to functions in proportion to their execution time at the highest frequency, as it is done in other systems [27], [51].

VIII. EVALUATION RESULTS

A. Energy Savings with Real-World Invocation Patterns

We use traces from Azure Functions [116] to mimic real-world invocation patterns while executing the functions from our benchmark suite. The traces capture the typical bursty behavior of serverless workloads. During a 10-second window, 119 different functions are invoked. On average, a function executes for 50 ms and is invoked 14 times during the window. However, about 10% of the functions are invoked more than 113 times, and there is at least one time when 33 invocations of the same function are executing concurrently.

From this trace, we select the 12 most popular functions, which account for over 76% of all invocations. We assign 12 of our benchmarks to these popular functions. We run the traces for 6 hours on a cluster of 5 servers. On average, each server receives 50-100 requests per second.

Figure 12 shows the normalized and absolute energy consumption of *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* for each individual benchmark and for the sum of all the

benchmarks. The bars in the figure show the total energy consumed by all the invocations of a given function. From the figure, we see that *Baseline+PowerCtrl* and *EcoFaaS* reduce the total energy consumption of the benchmarks by 33% and 60%, respectively, over *Baseline*.

EcoFaaS curbs the energy consumption of all the evaluated benchmarks. Compared to *Baseline*, the benefits are higher for benchmarks that are sensitive to core frequency; compared to *Baseline+PowerCtrl*, the benefits are higher for benchmarks with significant idle time, such as *ImgProc* and *RNNServ*. In addition, in applications with many functions such as *eBank* and *eBook*, the Workflow Controller in *EcoFaaS* assigns optimized per-function deadlines. The result is higher energy reduction of *EcoFaaS* over *Baseline+PowerCtrl*. Finally, by prewarming the missing function containers, the Workflow Controller effectively minimizes the number of function cold starts on the application’s critical path and their energy cost by executing them at lower frequencies. It can be shown that, in total, the prewarming technique contributes to 10.2% of the energy savings of *EcoFaaS* over *Baseline+PowerCtrl*.

To get more insight into the sources of energy savings, Figure 14 shows the average frequency across all cores in a server over time during the peak load for *Baseline* and *EcoFaaS*. We can see that *EcoFaaS* always operates at lower frequencies than *Baseline*. Recall that the Core Pools in *EcoFaaS* are reconfigured every $T_{refresh} = 2s$ to adjust to load changes. This is why the average frequency fluctuates. Figure 15 shows the distribution of core frequencies used by *EcoFaaS* across different dynamic function invocations. More than half of the invocations need less than 2.0GHz. Most invocations (25%) run at 1.8GHz, while the least number of invocations run at the highest frequency (4%) and at the lowest frequency (7%).

B. Energy Savings with Varying System Load

We evaluate *EcoFaaS* with different system loads in a cluster with 20 servers. We generate *Low*, *Medium*, and *High* loads with a Poisson distribution for request inter-arrival times. Recall that these loads correspond to CPU utilizations of 25%, 50%, and 70%, respectively. Figure 13 shows the normalized and absolute energy consumption of *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* with the three load levels for each individual benchmark and for the sum of all the benchmarks. Every client request invokes one of the twelve evaluated benchmarks randomly. Therefore, the distribution of invocations to the different benchmarks is different than in Figure 12.

With low load, the overall CPU utilization is low. Hence, even *Baseline* consumes only a modest amount of energy.

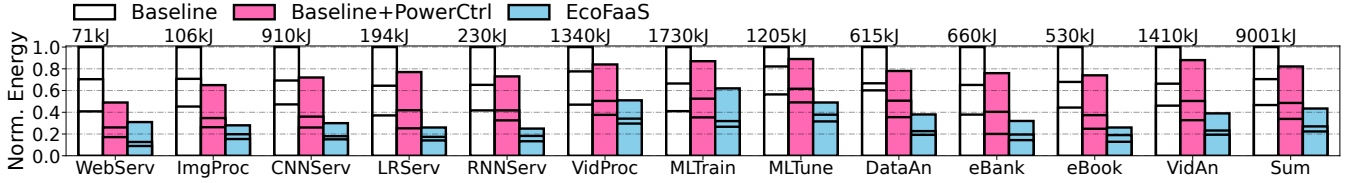


Fig. 13: Normalized energy consumption of *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* with Low, Medium, and High loads. In a given bar, the two horizontal lines show the values for Low and Medium loads, and the total bar corresponds to High load. All bars are normalized to Baseline-High. The numbers on top of the Baseline bars are the absolute energy consumption of Baseline-High for a 1-hour run on 20 servers.

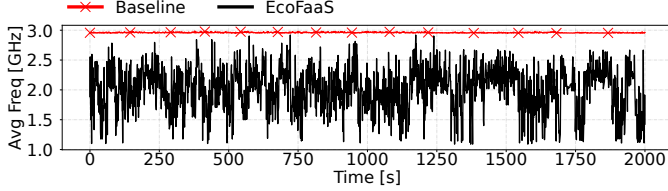


Fig. 14: Frequency over time during the peak load averaged across all cores in a server for *Baseline* and *EcoFaaS*.

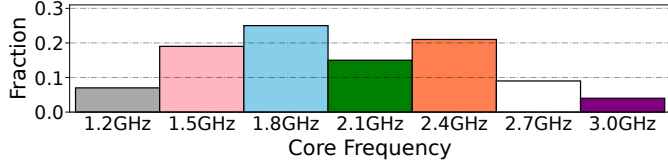


Fig. 15: Distribution of core frequencies used by *EcoFaaS* across different dynamic function invocations.

With medium load, *Baseline+PowerCtrl* reaches its sweet spot, mainly for two reasons: (i) the queuing effects are not substantial and, therefore, *Baseline+PowerCtrl* does not overestimate queuing time and does allow most of the requests to execute at lower frequencies, and (ii) there are few context switches across different containers and, therefore, containers keep their owned cores and the core frequencies do not need to be changed often. However, the benefits of *Baseline+PowerCtrl* diminish with high load. Request queues start building up and *Baseline+PowerCtrl* assigns higher frequencies than are actually needed to invocations. Further, cores frequently context switch between different containers, which results in crossing the boundary between user and kernel space many times.

Overall, while *Baseline+PowerCtrl* reduces the energy consumption of *Baseline* by 18%, 31%, and 27% with low, medium, and high load, respectively, *EcoFaaS* reduces it by 56%, 61%, and 52%, respectively.

C. Performance Improvements

To assess *EcoFaaS*' performance impact, we measure the reduction in end-to-end average and tail latency of requests, and the increase in their throughput. The end-to-end latency of a function or application invocation is the time from when the client sends a request until when it receives the result.

Tail latency. Figure 16 shows the normalized and absolute tail latency of the benchmarks when running with *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS*. The results are averaged across low, medium, and high load. *Baseline+PowerCtrl* substantially increase the tail latency over *Baseline* due to the

frequent and expensive core frequency changes. For example, under high load, *Baseline+PowerCtrl* runs *LRServ* at the highest frequency and *WebServ* at the lowest frequency. Thus, at every context switch between invocations of these two functions, *Baseline+PowerCtrl* changes the core frequency on the critical path. This overhead is higher than the time an *LRServ* invocation spends running on a core. Moreover, with increased load, more functions concurrently invoke the OS to change their core frequency, creating contention and further increasing tail latency.

On the other hand, *EcoFaaS* is able to keep the tail latency on-par with *Baseline* and even slightly reduce it. The reduction in tail latency comes from the ability to share cores between multiple functions, which reduces load imbalance. On average, *EcoFaaS* reduces the tail latency by 5.0% and 34.8% over *Baseline* and *Baseline+PowerCtrl*, respectively.

Average latency. We measure the average latency of *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* with different loads. As both *Baseline+PowerCtrl* and *EcoFaaS* allow invocations to execute at lower frequencies, their average response time is higher than the one with *Baseline*—which executes all requests at the highest frequency. As the load increases, more requests also need to execute at high frequencies with *Baseline+PowerCtrl* and *EcoFaaS* and, thus, their difference with *Baseline* shrinks. On average, it can shown that *EcoFaaS* increases the average response time over *Baseline* by $1.51\times$, $1.33\times$, and $1.17\times$ in low, medium, and high load, respectively. The reason for the higher average response time of *EcoFaaS* is the deliberate decision of *EcoFaaS* to slow-down function execution to the point of its SLO, in order to save energy. *EcoFaaS* finishes all requests within their deadline and reduces the average response time over *Baseline+PowerCtrl* by 13%, 19%, and 18% in low, medium, and high load.

Throughput. We measure a system's throughput as the highest sustained load that allows the benchmarks to still meet their SLO—which is defined as a tail latency below $5\times$ the execution time in an unloaded system. Figure 18 shows the tail latency of *CNNServ* as we increase its load with *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS*. The dashed line is *CNNServ*'s SLO. We see that *EcoFaaS* and *Baseline* keep the tail latency below the SLO until a load of 850 RPS, which is their throughput. On the other hand, *Baseline+PowerCtrl* reaches the function's SLO at 350 RPS, which is its throughput. The reasons for *EcoFaaS*'s high throughput are the improved CPU utilization and the reduced number of core frequency

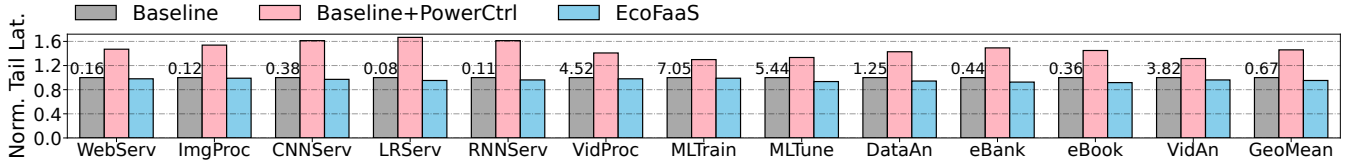


Fig. 16: Normalized tail latency with *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* averaged across different loads. The numbers on top of the *Baseline* bars are the absolute values of the benchmark tail latency measured in seconds.

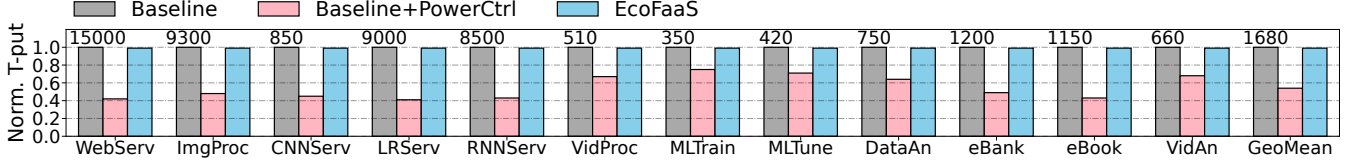


Fig. 17: Normalized throughput with *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS*. The numbers on top of the *Baseline* bars are the absolute values of the benchmark throughput measured in Requests-Per-Second (RPS).

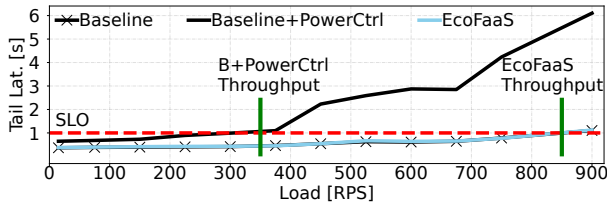


Fig. 18: Tail latency of CNNServ with the three systems while increasing the load. The dashed line indicates CNNServ’s SLO.

changes. For CNNServ, EcoFaaS improves the throughput over Baseline+PowerCtrl by 2.4 \times . For all the benchmarks, Figure 17 shows the normalized and absolute throughput with the three systems. On average, EcoFaaS improves the throughput over Baseline+PowerCtrl by 1.8 \times .

D. EcoFaaS Component Analysis

We measure the overheads introduced by different EcoFaaS components. First, for applications with multiple functions, the Workflow Controller uses the PuLP library [96] for its MILP solver to compute the optimal per-function deadlines. We test the time for the solver to produce its outputs as we vary the number of functions in an application (from 2 to 20) and the number of different frequency levels (from 2 to 10). The overhead of the MILP solver is around 10ms. Since EcoFaaS executes this operation only once every 5s and in the background, off the critical path, this overhead accounts for only 0.2% of the CPU cycles.

Second, Function Dispatchers communicate with the FPSs via shared memory to register invocations for execution. The communication overhead is only a few μ s. Finally, the Node Controller periodically reassigns cores across pools and sets the cores’ frequencies. The controller runs with root privileges. Hence, core frequency changes are triggered by writing to the MSR registers [93], and take effect in a few 10s of μ s.

We measure the accuracy of our prediction system. First, for functions whose response time does not change with different inputs (such as WebServe), we use a simple EWMA [80] approach. The Mean Absolute Percentage Error (MAPE) observed when predicting T_{run} , T_{block} , T_{queue} , and *Energy*

is 1.8%, 2.4%, 3.5%, and 1.9%, respectively. For functions whose response time and energy depend on the inputs (such as ImgProc), we predict the time and energy with a three layer neural network with ReLU activations. Its accuracy for both execution time and energy is 96.5% on average, while the prediction time is only 10-30 μ s.

E. Trade-offs and Sensitivity Analyses

Impact of prediction accuracy on system efficiency. We analyze how the accuracy of our prediction models affects the overall energy efficiency. We introduce bounded errors for the execution time overprediction and measure the energy consumption with various loads. Figure 19 shows the energy consumption of EcoFaaS during a 1-hour run on 20 servers under three loads with different average overprediction errors of the function execution time. The error is defined as $\frac{E-A}{A} \times 100\%$, where E is the estimated and A is the actual execution time. These overprediction errors force EcoFaaS to run at higher frequencies than necessary. From the figure, we see that the impact of the prediction error is modest across loads. For example, compared to an environment with no error, an environment with an 80% error increases the energy consumption by 22%, 16%, and 8% in low, medium, and high load, respectively. The impact diminishes at higher loads, as the system already runs at high frequency and there is less room to increase it.

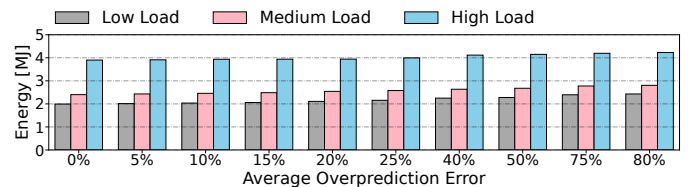


Fig. 19: Energy consumption of EcoFaaS during 1-hour runs on 20 servers under three system loads with different average overprediction errors of function execution time.

Sensitivity to configurations. We perform a sensitivity analysis of how the EcoFaaS efficiency is affected by the time between updates to the Delay-Power Table (DPT) and to the Core Pool

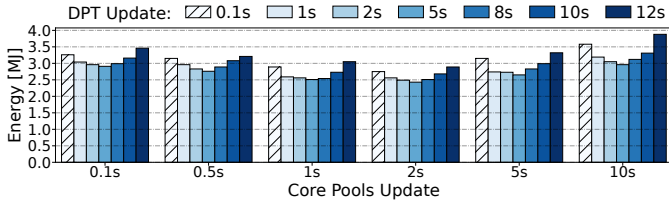


Fig. 20: Energy consumed for different times between updates to the DPT and Core Pools for *EcoFaaS* during a 1-hour run on 20 servers.

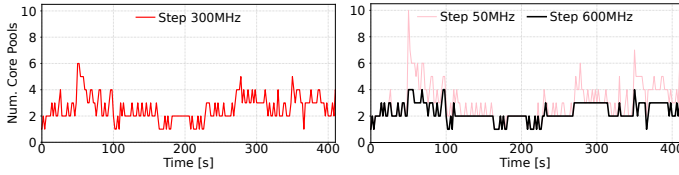


Fig. 21: Number of core pools with three core frequency granularities.

configuration. Figure 20 shows the total energy consumption during a 1-hour run on 20 servers with medium load, as we change the time between DPT updates from 0.1s to 12s, and the time between Core Pool updates from 0.1s to 10s. Updating DPT and Core Pools as frequently as every 0.1s introduces overheads that increase energy consumption. On the other hand, keeping the DPT and Core Pools unchanged for a long time results in *EcoFaaS* making sub-optimal decisions and increasing energy consumption. Thus, we set these parameters to their sweet-spot: 5s for DPT and 2s for Core Pools.

Frequency granularity. *EcoFaaS* picks frequencies for its Core Pools from a discrete set of values ranging from 1200MHz to 3000MHz. Too fine-grained core frequency steps leads to many Core Pools and high management overhead; too coarse-grain steps leads to too few Core Pools. Figure 21 shows the number of core pools over time in a 20-core node with core frequency granularity of 300MHz (left) and 50 and 600MHz (right) for the real-world invocation patterns of Figure 12. We see that the configuration with 300MHz steps is a good design point: it

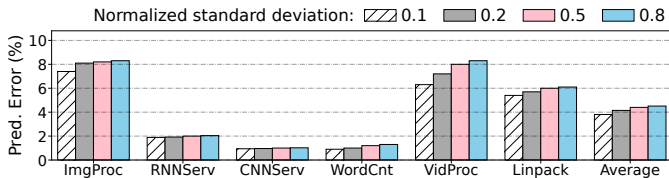


Fig. 22: Prediction error of function execution time for different levels of variability in function execution time.

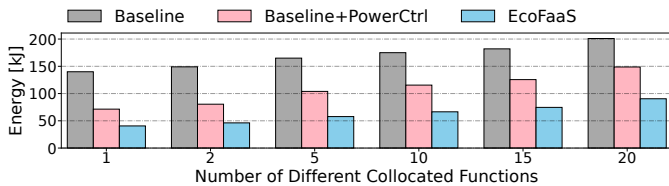


Fig. 23: Energy consumption of *CNNServ* during a 1-hour run on 1 server with *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* when collocated with different numbers of other (different) functions.

creates 1-6 pools. Steps of 50MHz result in many concurrent core pools (up to 10), fragmenting the server. It can be shown that this configuration increases the tail latency over the one with 300MHz steps by 6% and its energy consumption by 9%. On the other hand, steps of 600MHz reduce the number of concurrent core pools (to up to 4), inhibiting precise frequency tuning, and causing functions to run at higher than optimal frequencies. It can be shown that this configuration maintains the tail latency of the one with 300MHz steps but increases its energy consumption by 16%.

Impact of variation in function execution times on prediction error. Figure 22 shows how different levels of execution time variability affect the error in the prediction of function execution time. We generate different datasets and, for each one, we measure the execution time variability as the standard deviation of the dataset normalized to the maximum value in the dataset. We see that, for most functions, a higher variability does not affect the prediction accuracy much. However, for some functions such as *VidProc*, large variability in execution time increases the prediction error by an absolute 2%.

Impact of inter-function interference on system efficiency. Co-located functions may interfere, degrading system efficiency. Fortunately, *EcoFaaS* both profiles functions and trains the models online in a continuous manner. Consequently, it takes into account the interference between co-located functions. Figure 23 shows the energy consumption of the *CNNServ* function during one hour on a server with various numbers of co-located functions per core for the three systems. In all configurations, all functions are invoked with constant medium load. As the number of co-located functions increases, the function needs to run at higher frequencies for longer times. Thus, the energy consumption increases for all systems. However, we see that *EcoFaaS* consumes much less energy than the other two systems across all configurations. Other functions show a similar behavior.

IX. CONCLUSION

This paper proposed *EcoFaaS*, the first energy-management framework for serverless environments. *EcoFaaS* automatically splits the user-provided end-to-end application SLO into per-function deadlines that minimize the total energy consumption. It profiles functions online and, based on their deadlines, picks the optimal core frequencies. Further, *EcoFaaS* splits the cores into multiple pools, where all the cores in a pool run at the same frequency, and dynamically changes the size and frequency of the pools based on system conditions. Compared to state-of-the-art systems, *EcoFaaS* reduces the total energy consumption of serverless clusters by 42% while simultaneously reducing the tail latency by 34.8%.

ACKNOWLEDGMENTS

This work was supported in part by NSF under grants CNS 1956007 and CCF 2107470; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the IBM-Illinois Discovery Accelerator Institute.

REFERENCES

- [1] “Apache OpenWhisk,” <https://openwhisk.apache.org/>.
- [2] “Fission: Open Source Kubernetes-native Serverless Framework,” <https://fission.io/>.
- [3] “Fn Project,” <https://fnproject.io/>.
- [4] “Intel RDT Software Package,” <https://github.com/intel/intel-cmt-cat/tree/master>.
- [5] “Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues,” <https://www.serverless.com/blog/keep-your-lambdas-warm>.
- [6] “Knative,” <https://knative.dev/docs/>.
- [7] “Kubeless: The Kubernetes Native Serverless Framework,” <https://kubeless.io/>.
- [8] “OpenFaaS,” <https://docs.openfaas.com/>.
- [9] “Serverless Train Ticket,” <https://github.com/FudanSELab/serverless-trainticket>.
- [10] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight Virtualization for Serverless Applications,” in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, 2020.
- [11] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, “SAND: Towards High-Performance Serverless Computing,” in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.
- [12] M. Alian, A. H. M. O. Abulila, L. Jindal, D. Kim, and N. S. Kim, “NCAP: Network-Driven, Packet Context-Aware Power Management for Client-Server Architecture,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '17)*, 2017.
- [13] Amazon AWS, “AWS Lambda,” <https://aws.amazon.com/lambda/>.
- [14] Amazon AWS, “AWS Lambda Pricing,” <https://aws.amazon.com/lambda/pricing/>.
- [15] Amazon AWS, “AWS Step Functions,” <https://aws.amazon.com/step-functions/>.
- [16] Amazon AWS, “Call a microservice running on Fargate using API Gateway integration,” <https://docs.aws.amazon.com/step-functions/latest/dg/sample-apigateway-ecs-workflow.html>.
- [17] Amazon AWS, “Configuring Lambda function options,” <https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html>.
- [18] Amazon AWS, “Train a Machine Learning Model,” <https://docs.aws.amazon.com/step-functions/latest/dg/sample-train-model.html>.
- [19] Amazon AWS, “Tune a Machine Learning Model,” <https://docs.aws.amazon.com/step-functions/latest/dg/sample-hyper-tuning.html>.
- [20] A. Andrae and T. Edler, “On Global Electricity Usage of Communication Technology: Trends to 2030,” *Challenges*, vol. 6, 2015.
- [21] L. Ao, G. Porter, and G. M. Voelker, “FaaS Made Fast Using Snapshot-Based VMs,” in *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*, 2022.
- [22] AWS, “AWS Samples: AWS Serverless Workshops,” <https://github.com/aws-samples/aws-serverless-workshops/>.
- [23] Azure, “Azure Blob Storage,” <https://azure.microsoft.com/en-us/products/storage/blobs>.
- [24] Azure, “Azure Public Dataset,” <https://github.com/Azure/AzurePublicDataset>.
- [25] L. A. Barroso, J. Clidaras, and U. Hözlze, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, 2013. [Online]. Available: <http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024>
- [26] D. Beyer and P. Wendler, “CPU Energy Meter,” <https://github.com/sosy-lab/cpu-energy-meter>.
- [27] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, “Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.
- [28] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, “With great freedom comes great opportunity: Rethinking resource allocation for serverless functions,” in *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*, 2023.
- [29] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, “On-demand Container Loading in AWS Lambda,” in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '23)*, 2023.
- [30] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, “Serverless Workflows with Durable Functions and Netherite,” *CoRR*, vol. abs/2103.00033, 2021. [Online]. Available: <https://arxiv.org/abs/2103.00033>
- [31] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, “SEUSS: Skip Redundant Paths to Make Serverless Fast,” in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, 2020.
- [32] CB Insights, “Why Serverless Computing Is The Fastest-Growing Cloud Services Segment,” <https://www.cbinsights.com/research/serverless-cloud-computing/>.
- [33] S. Chen, C. Delimitrou, and J. F. Martínez, “PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.
- [34] S. Chen, A. Jin, C. Delimitrou, and J. F. Martínez, “ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*, 2022.
- [35] C.-H. Chou, L. N. Bhuyan, and D. Wong, “ μ DPM: Dynamic Power Management for the Microsecond Era,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA '19)*, 2019.
- [36] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe, “Process-Level Power Estimation in VM-Based Systems,” in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*, 2015.
- [37] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, “SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing,” in *Proceedings of the 22nd International Middleware Conference (Middleware '21)*, 2021.
- [38] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms,” in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Oct. 2017.
- [39] Datadog, “The state of serverless,” <https://www.datadoghq.com/state-of-serverless/>, 2023.
- [40] N. Daw, U. Bellur, and P. Kulkarni, “Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments,” in *Proceedings of the 21st International Middleware Conference (Middleware '20)*, 2020.
- [41] C. Delimitrou and C. Kozyrakis, “Amdahl’s Law for Tail Latency,” *Commun. ACM*, vol. 61, no. 8, jul 2018.
- [42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.
- [43] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, “Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.
- [44] G. Fieni, R. Rouvoy, and L. Seinturier, “SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers,” in *Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID '20)*, 2020.
- [45] A. Fuerst and P. Sharma, “FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.
- [46] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati, “Portable, scalable, per-core power estimation for intelligent resource management,” in *Proceedings of the International Conference on Green Computing*, 2010.
- [47] Google, “Google Cloud Functions,” <https://cloud.google.com/functions>.
- [48] Google Cloud, “Cloud Functions Pricing,” <https://cloud.google.com/functions/pricing>.
- [49] Google Cloud, “Serverless Computing,” <https://cloud.google.com/serverless/>.
- [50] Google Cloud, “Workflows,” <https://cloud.google.com/workflows>.
- [51] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, “Fifer: Tackling Resource Underutilization in the

- Serverless Era,” in *Proceedings of the 21st International Middleware Conference (Middleware '20)*, 2020.
- [52] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, “Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces,” in *Proceedings of the International Symposium on Quality of Service (IWQoS '19)*, 2019.
 - [53] U. Gupta, Y. Kim, S. Lee, J. Tse, H. S. Lee, G. Wei, D. Brooks, and C. Wu, “Chasing Carbon: The Elusive Environmental Footprint of Computing,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*, 2021.
 - [54] U. Gupta, M. Elgamel, G. Hills, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, “ACT: designing sustainable computer systems with an architectural carbon modeling tool,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*, 2022.
 - [55] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, “Exploiting Heterogeneity for Tail Latency and Energy Efficiency,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*, 2017.
 - [56] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Serverless Computation with OpenLambda,” in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud '16)*, 2016.
 - [57] C. Holt, “Forecasting seasonals and trends by exponentially weighted moving averages,” *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, 2004.
 - [58] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, “Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting,” in *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*, 2015.
 - [59] IBM, “IBM Cloud Composer,” https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer.
 - [60] IBM, “IBM Cloud Functions,” <https://cloud.ibm.com/functions/>.
 - [61] Intel, “Intel® Xeon® Gold 6142 Processor,” <https://ark.intel.com/content/www/us/en/ark/products/120487/intel-xeon-gold-6142-processor-22m-cache-2-60-ghz.html>, 2023.
 - [62] Intel, “Intel® Xeon® Processor E5-2640 v4,” <https://www.intel.com/content/www/us/en/products/sku/92984/intel-xeon-processor-e52640-v4-25m-cache-2-40-ghz/specifications.html>, 2023.
 - [63] Intel, “Intel® Xeon® Processor E5-2660 v3,” <https://intel.com/content/www/us/en/products/sku/81706/intel-xeon-processor-e52660-v3-25m-cache-2-60-ghz/specifications.html>, 2023.
 - [64] Z. Jia and E. Witchel, “Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.
 - [65] Y.-G. Jiang, J. Liu, A. Roshan Zamir, G. Toderici, I. Laptev, M. Shah, and R. Sukthankar, “THUMOS challenge: Action recognition with a large number of classes,” <http://csrc.ucf.edu/THUMOS14/>, 2014.
 - [66] K.-D. Kang, G. Park, H. Kim, M. Alian, N. S. Kim, and D. Kim, “NMAP: Power Management Based on Network Packet Processing Mode Transition for Latency-Critical Workloads,” in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021.
 - [67] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, “Rubik: Fast analytical power management for latency-critical systems,” in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '15)*, 2015.
 - [68] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, “RAPL in Action: Experiences in Using RAPL for Power Measurements,” *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, 2018.
 - [69] J. Kim and K. Lee, “FunctionBench: A Suite of Workloads for Serverless Cloud Function Service,” in *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD '19)*, 2019.
 - [70] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic Ephemeral Storage for Serverless Analytics,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, 2018.
 - [71] KNative, “Configure resource requests and limits,” <https://knative.dev/docs/serving/services/configure-requests-limits-services/>.
 - [72] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, “Faastlane: Accelerating Function-as-a-Service Workflows,” in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.
 - [73] S. Lahiri, “Complexity of Word Collocation Networks: A Preliminary Structural Analysis,” in *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*. Gothenburg, Sweden: Association for Computational Linguistics, April 2014, pp. 96–105. [Online]. Available: <http://www.aclweb.org/anthology/E14-3011>
 - [74] Z. Li, Q. Chen, S. Xue, T. Ma, Y. Yang, Z. Song, and M. Guo, “Amoeba: QoS-Awareness and Reduced Resource Usage of Microservices with Serverless Computing,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '20)*, 2020.
 - [75] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, “FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.
 - [76] P. Lin and A. Glikson, “Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach,” *CoRR*, vol. abs/1903.12221, 2019. [Online]. Available: <http://arxiv.org/abs/1903.12221>
 - [77] Q. Liu and Z. Yu, “The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View from Alibaba Trace,” in *Proceedings of the 2020 ACM Symposium on Cloud Computing (SoCC '18)*, 2018.
 - [78] Y. Liu, S. C. Draper, and N. S. Kim, “SleepScale: Runtime joint speed scaling and sleep states management for power efficient data centers,” in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14)*, 2014.
 - [79] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads,” in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14)*, 2014.
 - [80] J. M. Lucas and M. S. Saccucci, “Exponentially weighted moving average control schemes: Properties and enhancements,” *Technometrics*, vol. 32, no. 1, pp. 1–12, 1990.
 - [81] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, “Learning word vectors for sentiment analysis,” in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: <http://www.aclweb.org/anthology/P11-1015>
 - [82] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chatterji, and S. Bagchi, “SONIC: Application-aware Data Passing for Chained Serverless Applications,” in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.
 - [83] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chatterji, “WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows,” in *Proceedings of the 2022 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2022.
 - [84] E. R. Masanet, A. Shehabi, N. Lei, S. J. Smith, and J. G. Koomey, “Recalibrating global data center energy-use estimates,” *Science*, 2020.
 - [85] MathWorks, “Mixed-Integer Linear Programming (MILP) Algorithms,” April 2024. [Online]. Available: <https://www.mathworks.com/help/optim/ug/mixed-integer-linear-programming-algorithms.html>
 - [86] Microsoft, “Microsoft Azure Functions,” <https://azure.microsoft.com/en-gb/services/functions/>.
 - [87] Microsoft Azure, “Available Instance SKUs,” <https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal#available-instance-skus>.
 - [88] Microsoft Azure, “Azure Durable Functions,” <https://docs.microsoft.com/en-us/azure/azure-functions/durable/>.
 - [89] Microsoft Azure, “Azure Functions Pricing,” <https://azure.microsoft.com/en-us/pricing/details/functions/>.
 - [90] A. Mirhosseini and T. Wenisch, “μSteal: A Theory-Backed Framework for Preemptive Work and Resource Stealing in Mixed-Criticality Microservices,” in *Proceedings of the ACM International Conference on Supercomputing (ICS '21)*, 2021.
 - [91] R. Nishtala, V. Petrucci, P. Carpenter, and M. Själinder, “Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*, 2020.

- [92] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.
- [93] OSDev, "Model Specific Registers," https://wiki.osdev.org/Model_Specific_Registers.
- [94] J. Pont-Tuset, F. Perazzi, S. Caelles, P. Arbelaez, A. Sorkine-Hornung, and L. V. Gool, "The 2017 DAVIS challenge on video object segmentation," *CoRR*, vol. abs/1704.00675, 2017. [Online]. Available: <http://arxiv.org/abs/1704.00675>
- [95] Preeti Wadhvani, "Serverless architecture market size to cross \$90 Bn by 2032," <https://www.gminsights.com/pressrelease/serverless-architecture-market>.
- [96] Python PuLP, "Optimization with PuLP," <https://coin-or.github.io/pulp/>.
- [97] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC '12)*, 2012.
- [98] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "FaaS\$: A Transparent Auto-Scaling Cache for Serverless Applications," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.
- [99] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming Serverless Functions Better with Heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.
- [100] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*, 2019.
- [101] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.
- [102] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.
- [103] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A Scalable Low-Latency Serverless Platform," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.
- [104] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.
- [105] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '23)*, 2023.
- [106] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "UniCache: The Next 700 Caches for Serverless Computing," in *International Workshop on Cloud Intelligence / AIOps (AIOps '24)*, 2024.
- [107] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling Quality-of-Service in Serverless Computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*, 2020.
- [108] The vHive Ecosystem, "vSwarm - Serverless Benchmarking Suite," <https://github.com/vhive-serverless/vSwarm>.
- [109] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.
- [110] J. van Winkel, "The Production Environment at Google, from the Viewpoint of an SRE," <https://sre.google/sre-book/production-environment/>, 2023.
- [111] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.
- [112] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, 2020.
- [113] M. Wawrzoniak, I. Müller, G. Alonso, and R. Bruno, "Boxer: Data Analytics on Network-enabled Serverless Platforms," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR '21)*, 2021.
- [114] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '20)*, 2020. [Online]. Available: <https://doi.org/10.1145/3419111.3421280>
- [115] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars, "HaPPy: Hyperthread-aware Power Profiling Dynamically," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '14)*, 2014.
- [116] Y. Zhang, Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and Cheaper Serverless Computing on Harvested Resources," in *Proceedings of the International Symposium on Operating Systems Principles (SOSP '21)*, 2021.
- [117] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, 2021.
- [118] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan, "Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.