Fine-Grained Heterogeneous Execution Framework with Energy Aware Scheduling

Gourav Rattihalli
Hewlett Packard Labs
Milpitas, CA, United States
gourav.rattihalli@hpe.com

Ninad Hogade
Hewlett Packard Labs
Milpitas, CA, United States
ninad.hogade@hpe.com

Aditya Dhakal
Hewlett Packard Labs
Milpitas, CA, United States
aditya.dhakal@hpe.com

Eitan Frachtenberg
Hewlett Packard Labs
Portland, OR, United States
eitan.frachtenberg@hpe.com

Rolando Pablo Hong Enriquez
Hewlett Packard Labs
Oxford, United Kingdom
rhong@hpe.com

Pedro Bruel
Hewlett Packard Labs
Milpitas, CA, United States
bruel@hpe.com

Alok Mishra
Hewlett Packard Labs
New Jersey, NJ, United States
alok.mishra@hpe.com

Dejan Milojicic
Hewlett Packard Labs
Milpitas, CA, United States
dejan.milojicic@hpe.com

***Abstract*—The growing convergence of high-performance, data analytics, and machine-learning applications is increasingly pushing computing systems toward heterogeneous processors and specialized hardware accelerators. Hardware heterogeneity, in turn, leads to finer-grained workflows. State-of-the-art serverless computing resource managers do not currently provide efficient scheduling of such fine-grained tasks on systems with heterogeneous CPUs and specialized hardware accelerators (e.g., GPUs and FPGAs). Working with fine-grained tasks presents an opportunity for more efficient energy use via new scheduling models.**

Our proposed scheduler enables technologies like Nvidia's Multi-Process Service (MPS) to pack multiple fine-grained tasks on GPUs efficiently. Its advantages include better co-location of jobs and better sharing of hardware resources such as GPUs that were not previously possible on container orchestration systems. We propose a Kubernetes-native energy-aware scheduler that integrates with our heterogeneous framework. Combining fine-grained resource scheduling on heterogeneous hardware and energy-aware scheduling results in up to 17.6% improvement in makespan, up to 20.16% reduction in energy consumption for CPU workloads, and up to 58.15% improvement in makespan, and up to 28.92% reduction in energy consumption for GPU workloads.

***Index Terms*—Energy-awareness, Heterogeneity, Serverless, Framework, Scheduler, Fine-granularity**

I. Introduction

The widespread use of cloud services and cloud-based technologies such as containers and microservices in the IT industry has given rise to a new execution model known as serverless computing, or FaaS (Function as a Service). This model leverages container-based virtualization to allow software developers to deploy their applications with no infrastructure or maintenance costs and minimal operating expenses, making it an ideal solution for building and optimizing Internet of Things (IoT) operations. A key feature of serverless platforms is their high level of abstraction, which decomposes applications into fine-grained functions that are deployed and executed with no control from the developers. Unlike other cloud services, serverless function resources are only allocated and billed for when the function is invoked, eliminating the cost of preallocating resources. Thus, serverless computing has the potential to increase the efficient use of cloud resources, reduce the consumption of power, and lower the cost of using cloud services.

In serverless computing, the cloud provider manages and dynamically allocates the necessary computing resources to run the application code (or functions). The serverless model allows developers to focus on code development and reduces the burden of managing infrastructure. However, hardware heterogeneity in serverless frameworks can affect their functionality in several ways. Here are some of the ways in which heterogeneity in hardware affects serverless frameworks:

- **Performance:** The performance of an application can vary significantly based on the hardware it runs on. Different hardware resources have different computing architectures, processing speeds, memory capacities, and storage capabilities, which can impact the application's performance. For example, if the application requires high computational power, it may run slower on a serverless platform with low-powered CPUs.
- **Resource allocation:** Serverless frameworks dynamically allocate computing resources based on the demand for the application code. However, the hardware heterogeneity can make it difficult to allocate resources accurately. For example, if an application requires a lot of memory, but the serverless environment only has limited memory available, it may result in the application being throttled or not working correctly.
- **Compatibility:** Different hardware architectures require specific libraries to run correctly. If an application or function relies on specific software or libraries that are incompatible with the hardware environment, it may not run or may not get deployed at all.
- **Cost:** The cost of running an application or function on a serverless platform may vary depending on its hardware. For example, if an application requires high parallel computational power, it maybe more expensive to run

it on a serverless platform that uses high-performance hardware/accelerators.

To mitigate the impact of heterogeneity in hardware, serverless platforms usually use containerization technology such as Docker to create a consistent execution environment for application code. Additionally, some serverless platforms may offer multiple hardware options, allowing developers to select the hardware that best fits their application's requirements. But the current containerization technologies are not mature enough to work with various heterogeneous hardware resources. Developers need to understand the implications of hardware heterogeneity when designing and deploying applications in a serverless environment.

The rapid development of cloud services has led to the construction of large data centers, which face the challenge of rising power consumption due to increasing workloads. Data centers are estimated to account for about 2% of all global greenhouse gas emissions and consume roughly 3% of the world's electrical energy supply, equivalent to the entire airline industry [1]. This figure is expected to rise to 8% by 2030, which is alarming given the impact observed during the Covid-19 pandemic [2]. The workloads and data processed in data centers are continuously growing, leading to a 10-30% annual increase in energy consumption, a trend likely to remain unabated in the coming years [3].

It is crucial to understand these workloads and their energy consumption patterns to design more energy-efficient workload schedulers. Such schedulers must be aware of hardware performance per unit of energy and use appropriate constraints to schedule workloads on heterogeneous hardware. IT equipment, which includes compute, storage, and network resources, accounts for 50% of the power consumed in a data center, with servers consuming 40% of the IT equipment's share [4]. Therefore, improving cloud services' power consumption on such infrastructures can significantly reduce data center power consumption. This underscores the urgent need for energy-efficient resource management in cloud data centers.

Both heterogeneity and energy consumption can impact the functionality and sustainability of cloud data centers running serverless frameworks. Therefore, it is crucial to consider these issues when designing and deploying applications in a serverless environment. By optimizing resource allocation and energy consumption, developers can minimize the environmental impact of serverless computing while maximizing its benefits. To overcome the issue of Heterogeneity and energy-awareness with current frameworks, we present our Heterogeneous Execution Framework with a native Kubernetes scheduler that implements energy-aware techniques.

We make the following contributions in this paper:

- We propose a Kubernetes-native energy-aware scheduler for systems with heterogeneous CPUs and specialized hardware accelerators (e.g., GPUs and FPGAs).
- Our proposed framework enables technologies like Nvidia's Multi-Process Service (MPS) to efficiently pack multiple fine-grained tasks on GPUs.

We organize the rest of the paper as follows. In Section II, we provide background on heterogeneity, energy-awareness, and serverless frameworks. In Section III, we characterize our framework design, its components, and implementation. We include the experimental setup and the results of the experiments in Section IV. We discuss the outcomes of our experiments and their effects on the application runtime in Section V. Sections VI and VII describe future work and related work, respectively. Lastly, we conclude our work in Section VIII.

II. Background

Hardware resource requirements for High-Performance Computing, Data Analytics, and AI/ML applications are continuously changing. To improve the performance of such applications, heterogeneous hardware such as Graphics Processing Unit (GPU), Field Programmable Gate Array (FPGA), Tensor Processing Unit (TPU), Smart Network Interface Card (SmartNIC), and Quantum Processing Unit (QPU) are rapidly gaining popularity [5]. But such hardware is not fully integrated with today's state-of-the-art container-orchestration systems like Kubernetes [6] and Apache Mesos [7], or with serverless systems like Apache OpenWhisk [8], AWS Lambda [9], and Knative [10]. While some of these systems support specific hardware like GPU in the case of Kubernetes and its related serverless frameworks, they do not provide any optimizations for such hardware. For example, in the case of Nvidia GPU, Kubernetes can only schedule one application per GPU and, with time-sharing configurations, provide multi-application support. But, time-sharing is not the optimal way to share GPU resources. Nvidia provides Multi-Process Service (MPS) [11], and Multi-Instance GPU (MIG) [12], which are much better at sharing GPU resources. Kubernetes does support MIG, but MIG statically divides the GPU into up to seven equal parts at boot time. This, in our opinion, severely limits the possibilities for fine-grained scheduling. On the other hand, MPS provides the ability to allocate the required number of fine-grained GPU resources dynamically and at a finer grain, in this case, Streaming Multiprocessors (SMs).

Serverless computing has gained immense popularity in recent years because of its ability to abstract infrastructure management and provide scalability to applications without requiring complex configuration. There are several state-of-the-art serverless frameworks available in the market that offer unique features to developers, including:

FuncX [13], an open-source serverless framework that offers a simple and easy-to-use interface for deploying serverless functions. It supports Python functions and provides scalability, reliability, and fault tolerance to functions. It is particularly useful for scientific computing and data-driven workflows.

Apache Airavata [14] is another framework that supports building and executing complex scientific workflows. It offers a user-friendly interface for managing workflows and integrates with popular scientific tools and services.

Fission [15] is another serverless framework focusing on fast cold start times and supporting many languages. It can

automatically scale functions up and down based on demand and can be easily integrated with Kubernetes.

rFaaS [16], a serverless framework specifically designed for remote function-as-a-service workloads. It provides efficient and scalable execution of serverless functions across multiple data centers and cloud providers.

While each serverless framework has unique features and benefits, choosing the right framework depends on the application's specific use case and requirements. When selecting a serverless framework, it is essential to consider factors such as ease of use, scalability, fault tolerance, and cost-effectiveness. Many open-source serverless frameworks rely on the underlying platform **Kubernetes**—also known as K8s—an open-source system for automating containerized applications' deployment, scaling, and management. It provides various benefits, such as custom accelerator support, automatic scaling, multiple container runtimes, support for high-performance computing, open source, and product maturity. Because of its many advantages, we develop our energy-aware scheduler on top of Kubernetes.

III. Design

We have designed a framework to execute and manage tasks for heterogeneous severless computing (HSC) and to work with container-orchestration systems like Kubernetes (Figure 1). For applications that need to run in a Kubernetes-managed cluster, a Kubernetes control plane must be registered with the framework. The framework then submits applications to the Kubernetes control plane. If an application needs to be executed on other hardware, we use a device plugin to run that application either on a bare-metal node CPU or an accelerator if requested. Accelerators such as FPGAs do not have Kubernetes support; therefore, managing them via K8s master is challenging. We build a custom accelerator runtime, using Pylog [17], to allocate FPGA-intensive tasks to FPGA-based nodes bypassing the K8s Master. Pylog uses Python to program FPGAs, bridges the gap between software and hardware abstraction.

Figure 2 shows the design of the device plugin. The device plugin can be used for hardware connected to a bare-metal node, such as CPU, FPGA, and GPU. Jobs that are executed using the device plugin are received from the user by our framework's API server; the Job is sent to the HSC scheduler, which can be loaded with different algorithms (FIFO, energy-aware, etc.); the scheduler decides if the Job needs to be added to the queue or forwarded to an agent node for execution. The API server also sends any application data provided to a data store accessible by the agent nodes. On each agent node, an HSC executor daemon accepts the application, retrieves the application data from the data store if required, and executes the application on the appropriate device. Once completed, the job logs are added to the data store, and an API call is made to the API server to signal the job completion. The cluster resource metrics are monitored by the framework using Performance Co-Pilot (PCP) [18].

1) Kubernetes Energy Aware Scheduler: While our framework can be used with the default Kubernetes scheduler, we have also designed an energy-aware Kubernetes scheduler that uses energy consumption as the main scheduling parameter to optimize. The scheduler is designed with heterogeneous hardware and different energy requirements considerations in mind. For example, different CPUs consume different amounts of energy and scale energy usage differently as the core usage changes. To understand such characteristics, we have designed an energy profiler that captures the energy statistics of a CPU at different usage levels. We use *stress-ng* [19] to stress the CPU at each core level and monitor the energy consumption using RAPL [20] and Performance Co-Pilot [18]. The profiler then computes the average energy requirement at each core level and prepares a data file for the scheduler. Figure 3 shows how two different CPUs with identical core counts consume different amounts of energy at each core. This means that energy is not a static resource like CPU and RAM that can be provided during job submission. Our scheduler accepts CPU and memory requirements from the user just like any other system. It then converts these requirements into energy consumption estimates based on the current node usage and schedules it on the node that might consume the least energy. Kubernetes does not provide a direct API to get the current resource allocation of a node, so we compute the allocated resources by using the pods running on the node. We do this for all the nodes. From the profiled data of each node, we get the energy required to run the allocated resources plus the resources needed by the job. We then allocate the minimum difference of energy required to run the allocated resource and the allocated resources plus the required resources by the job, provided energy resources are available. The maximum amount of allocatable energy is the node's TDP (Thermal Design Power). The TDP value is added as an extended resource in Kubernetes.

2) Fine-Grained GPU scheduling: Kubernetes provides support for GPU by default, but this support is limited, and Kubernetes can only schedule one application per GPU. With time-sharing configurations, Kubernetes can support the scheduling of multiple applications on the GPU, but this mechanism is sub-optimal for resource sharing. NVIDIA GPUs offer spatial sharing of GPU compute resources using multi-process service (MPS). Unlike time-sharing, with MPS, the kernels from different applications can run concurrently. Furthermore, with MPS, users can also allocate the GPU percentage for each application. GPU% indicates the maximum number of GPU streaming multiprocessors (SM) each application can use; e.g., an application getting 50% of an A100 GPU means it can use 54 out of a total of 108 SMs. Running multiple applications together without oversubscribing the GPU (i.e., keeping the total used GPU% less than 100%) will effectively provide compute isolation for each application. Therefore, it prevents interference between concurrently running applications. Our framework supports execution with MPS, and each MPS job is managed by the framework. While Kubernetes manages the execution of the

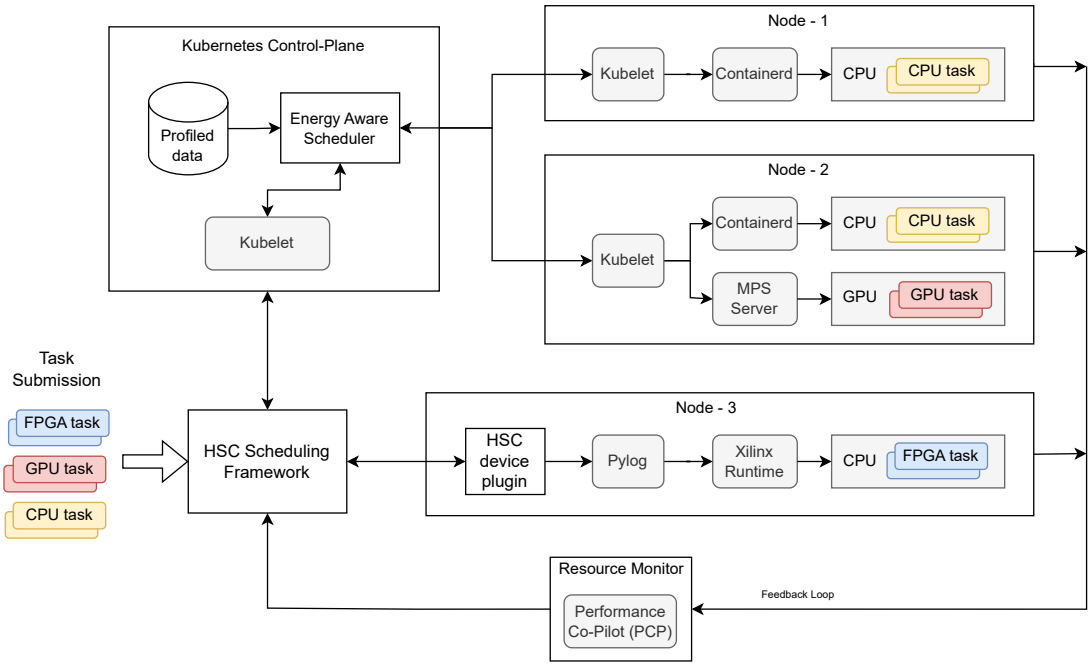

Fig. 1. Heterogeneous serverless computing (HSC) framework

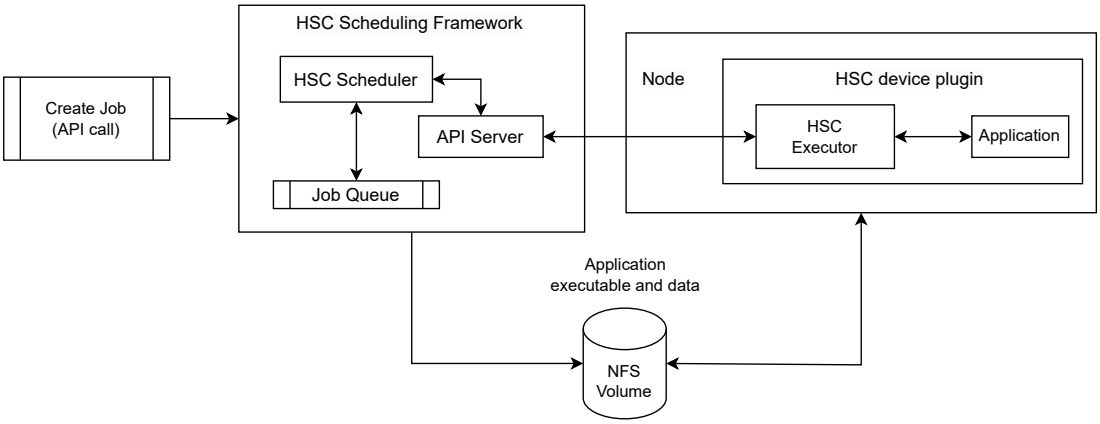

Fig. 2. HSC Device Plugin

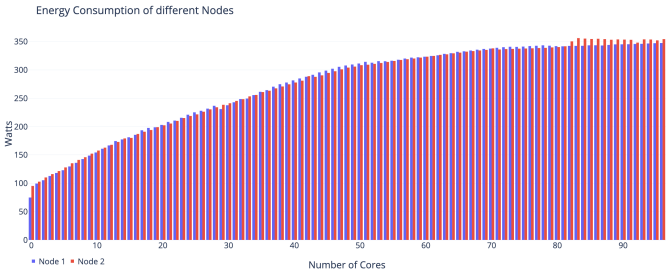

Fig. 3. Average Energy Consumption of nodes at different CPU core usage

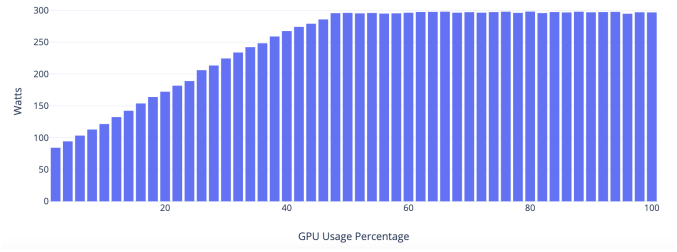

Fig. 4. Average GPU energy consumption at different GPU usage percentage

job on the GPU, the resource management is done by our framework.

Figure 4 shows a GPU's energy consumption at each GPU usage percentage level. We can see a stark difference compared to CPU energy usage at different core levels. On the CPU side, there is a noticeable difference in energy consumption

Algorithm 1 Pseudo Code - Energy-aware Algorithm

```
for each node do
    for each pod do
        check resource consumption (CPU and memory)
        add pod resource consumption to node resource consumption
    end for
end for
for each node with available compute and energy resources do
    if Compute resources are available then
        compute energy required to run the application (using profiled data)
    end if
    if compute resources are not available but energy resources are available then
        if 80% compute resources are available then
            Compute the energy required using the difference of the last two cores using profiled data
        end if
    end if
end for
assign the job to the node with the least computed energy requirement
```

when jumping from one CPU core of utilization to another. The same cannot be said for GPUs. In GPUs, we see a similar pattern at lower GPU utilization, but once we start consuming more than about 45% of the GPU, the energy consumption reaches its peak. It does not change much for any subsequent percentage level. When designing efficient cloud clusters, hardware utilization is an important factor, and any organization would want to maximize the utilization. In such cases, there are few opportunities to perform fine-grained resource management on GPUs using energy as the main parameter. A better alternative is to maximize the usage of the GPU by fine-grained resource allocation, which our framework achieves by utilizing the latest GPU technologies like MPS. MPS is a feature of NVIDIA GPUs; similar technologies (MxGPU) exist in AMD GPUs as well.

IV. Experimental Results

A. Experimental Setup

Our setup consists of four nodes, each with two AMD EPYC 7443 24-Core CPUs. Two nodes have Nvidia A100X 80GB GPU, while all three nodes have a network interface controller either integrated with the GPU or installed on a PCI slot. For our workload, we use the PARSEC benchmark suite [21], Mandlebrot [22], matrix multiplication kernel (MatMul), Resnet-50 and Resnet-18 [23].

B. Results and Analysis

1) CPU tasks: Figure 5 shows the energy consumption patterns of using the default Kubernetes and our energy-aware scheduler. For our experiment, we launched 210 PARSEC benchmarks, each compiled for sequential execution. Each job requests one CPU core and 1024MB of Memory. In the case of default Kubernetes, each job is configured with these resources and launched directly on Kubernetes. The total number of available cores in the cluster is 192, and one node serves as the control plane. This means that Kubernetes can only schedule a maximum of 192 1-core jobs on the agent nodes. The schedulable resources are fewer than 192 since Kubernetes reserves a few pods required for cluster management. Kubernetes can schedule 190 jobs, but about 20 jobs move into a pending status until the resources become available again. We see the effect of scheduling these 20 jobs later in Figure 5. These 20 jobs get scheduled on the cluster at the 320-second mark and create a spike in energy consumption. This causes the entire experiment to take much longer, averaging 480 seconds over five runs.

We executed the same experiment with our energy-aware scheduler. Our scheduler is run as a pod within the Kubernetes cluster and watches for pod-creation events. It runs alongside the Kubernetes default scheduler and schedules the job that needs to be scheduled with energy-aware considerations. The framework receives the jobs from the user, and the configuration file required by the scheduler is prepared and submitted to Kubernetes. This configuration includes the flag required by the scheduler. The scheduler watches for new pod creation events from Kubernetes, and the job that contains the scheduler flag is re-configured with energy requirements. At this point, the CPU and Memory requirements for the job are removed, and scheduling is purely done using energy requirements. This results in more jobs being scheduled on the node than the default method, as the jobs are not restricted by the number of available CPU cores. The experiment shows that all the jobs get scheduled on the nodes, and the makespan shrinks to 399 seconds. Similarly, we observe a reduction in peak power utilization using our scheduler.

Our energy-aware scheduler does not fully guarantee the user-requested resources; the jobs are allocated with a minimum guarantee of 80% resources. At the same time, the maximum limit is set at the user-requested resources. In the default Kubernetes experiment, we allocate the user-requested resources. Therefore, the benefits of the energy-aware scheduler are not necessarily captured by this experiment, as the difference in energy and makespan could be entirely attributed to the changes in resource allocation. To show this is not the case, we also execute the same experiment with default Kubernetes but without requesting any resources. This allows Kubernetes to schedule using its default spread algorithm, which spreads an equal number of jobs on each node. Note that not all nodes consume the same amount of energy. With Kubernetes spreading an equal number of jobs on all nodes, the node that consumes more energy is also scheduled from the start. This, in turn, results in more energy being consumed by the same set of applications. While, on average, the runtime is slightly higher than when using our energy-aware scheduler, the energy consumption peaks are also much higher.

Table I shows the average makespan and energy consump-

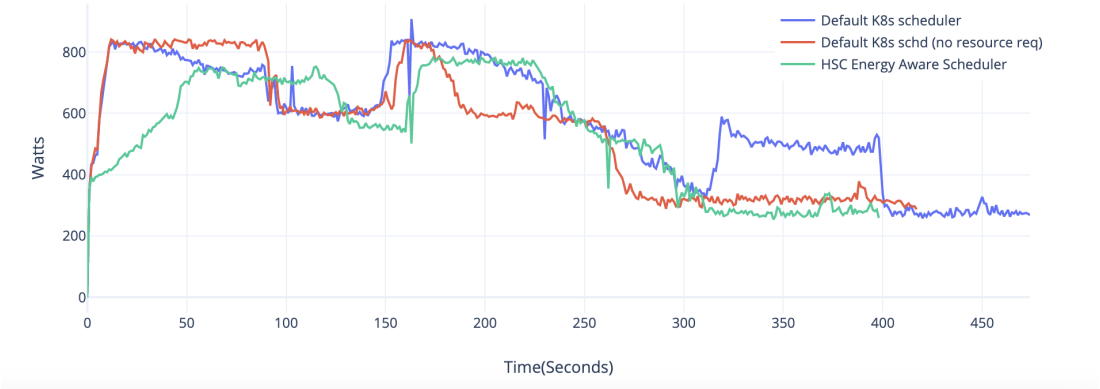

Fig. 5. Energy consumption of default K8s and our energy-aware K8s scheduler

TABLE I
AVERAGE MAKESPAN AND ENERGY CONSUMPTION

Scheduler	Makespan (s)	Energy (Wh)
Default K8s	476	73.47
Default K8s (no resource request)	430	66.53
Energy Aware	399	60.03

tion to run 210 applications on a Kubernetes cluster. With Default Kubernetes, we have an average makespan of 476 seconds, about 77 seconds or 17.6% longer than our energy-aware scheduler and about 46 seconds or 10.15% longer than scheduling without requesting any resources. We can also quantify the effects of using energy-aware scheduling; both energy-aware scheduler and default Kubernetes scheduler (no resource request) do not honor the user-requested CPU or Memory resources. This effectively shows that having the profiled data on the energy of all systems and using these profiles to deliberately schedule on the least power-consuming node at the time of execution can lead to a noticeable reduction in power consumption without impacting makespan. Compared to default Kubernetes (no resource request), our energy-aware scheduler has a makespan lower by about 31 seconds or 7.47%, and lower energy consumption by about 10.3%. Unlike default Kubernetes, our energy-aware scheduler consumes about 20.16% less energy.

TABLE II
GPU AVERAGE MAKESPAN AND ENERGY CONSUMPTION

Scheduler	Makespan(s)	Energy(Wh)
Default K8s	887	71.92
Default K8s(time-sharing)	364	36.93
HSC Framework	200	27.6

2) GPU tasks: Figure 6 shows the energy consumption patterns for default Kubernetes, default Kubernetes with time-sharing configurations, and our Heterogeneous Execution Framework (HSC). Table II shows each scheduling method's makespan and energy consumption. Default Kubernetes can only execute one application on a GPU at a time; this severely limits the capability of a GPU. In many situations, one application cannot utilize the GPU completely, and sharing the available resources is logical. This is partially solved by having time-sharing configurations in Kubernetes, where Kubernetes allows scheduling multiple applications on the GPU. The configurations explicitly specify the number of applications that can concurrently run on the GPU. In our experiments, this value was set at 10 per GPU. The configurations can be different for each GPU. Our experiments consisted of a mix of applications: Mandelbrot [24], matrix multiplication, Resnet-18 [23], and Resnet-50 [25]. We executed 40 jobs, 10 of each. In the default Kubernetes, each job consumes the entire GPU, which results in a low GPU utilization; this also means that at a given point during the experiment execution, the default Kubernetes consumes the least amount of energy. But for the entire experiment, it consumes the most at 71.92 Wh. Default Kubernetes also takes the most time at 887 seconds. With time-sharing configurations, default Kubernetes consumes 36.93 Wh. That is a reduction of about 64.29%. There is also a large reduction in makespan, from 887 seconds to 364 seconds. A reduction of 83.61%. But as already established, time-sharing is not the most optimal way to share GPU resources. With the inclusion of technologies such as MPS in the HSC framework, the same experiment consumes even less energy and greatly improves the makespan. Our framework successfully completes the same experiment in 200 seconds, an improvement of 58.15%. When compared to the default Kubernetes, the difference is about 126%. There are similar improvements in energy compared to default Kubernetes with time sharing; our framework consumes 28.92% less energy at 27.6 Wh. When compared to default Kubernetes, the difference is 89.07%.

V. DISCUSSION

1) CPU: Effects on individual application runtime: In our energy-aware scheduler, we do not fully honor the user-requested resources. Instead, we allocated 80% of the re-

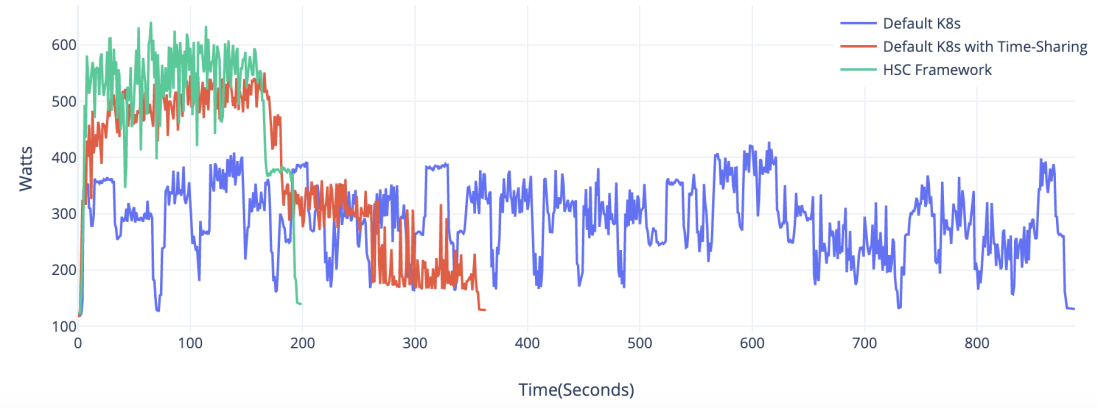

Fig. 6. Energy Consumption of default k8s, default K8s with Time Sharing and our HSC Framework

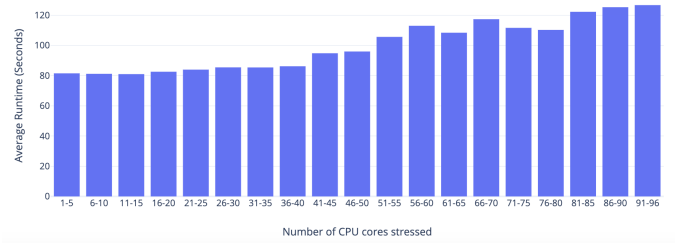

Fig. 7. Average Runtime at each CPU core usages

quested resources. This results in many tasks sharing the same CPU core. Which, in turn, results in each application taking longer to complete. In Figure 7, we show the effects on runtime while sharing the CPU. The experiment was conducted on a node with two AMD EPYC 7443 24-Core CPUs and multithreading enabled. This node has 48 physical cores and 96 multithreaded cores. We stressed the CPU with a specified number of cores using stress-ng. Once the cores were stressed, we executed the Blackscholes benchmark from the PARSEC benchmark suite. This benchmark, on average, takes about 82 seconds to complete when compiled serially. We see this in our experiment if the benchmark has full access to a CPU core. As the physical cores of the CPU start being shared, the runtime to complete the application also increase. The difference in runtime is about 1.5 times.

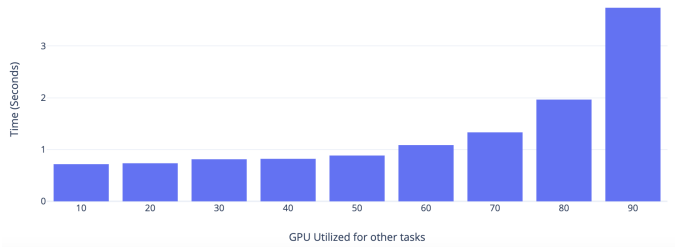

Fig. 8. Average Runtime at each GPU utilization percentage

2) GPU: Effects on individual application runtime: We conducted a similar experiment on GPUs. We used 1000 iterations of Resnet-50 Inference on Nvidia A100x GPU. The experiment shows identical characteristics to that of the CPU. GPUs can be divided into percentages, effectively dividing the number of SMs on the GPU. We ran high-dimension matrix multiplication to stress the GPU to the required GPU utilization percentage and then ran our Resnet-50 application. When there is little stress on the GPU, the runtime of the Resnet-50 application is not affected. But, as the stress increases, the runtime increases multifold.

The CPU and the GPU experiments show that while fine-grained scheduling significantly improves energy and overall makespan for a set of tasks, it comes at a cost to individual application runtime. Our goal in this paper was to design an energy-efficient framework that could handle fine-grained tasks normally seen in serverless systems. These tasks are short running, and the benefits seen in energy consumption outweigh the increase in runtime.

3) Long running tasks: Our framework is mostly catered to serverless tasks. Serverless tasks are usually short-running, and most current systems have a maximum execution time; for example, AWS Lambda has a limit of 900 seconds [26]. Orchestration systems like Kubernetes do not allow changing the resource allocation of scheduled jobs; this is not ideal in our situation. Our scheduler is designed to allocate energy for tasks based on the current energy allocations per node. If the tasks are long-running, then the energy allocation of the job will also need to change every time there is a change in the node's resource consumption. While this is not an issue for short-running tasks, for long-running tasks, the resource allocation made at the time of scheduling would be incorrect at a later point in time.

VI. Future Work

There are three areas of future work we will focus on: 1) integration with a tool for application performance prediction; 2) integration with FuncX for global scheduling, beyond a

single cluster; and 3) leveraging reinforcement learning for improved scheduling. We discuss these opportunities in the rest of this section.

1) Integration with Application Performance Prediction: One of the shortcomings of this work is that we rely on users' input on the compute resource requirement. In most systems, it is left to the user to provide the resources required to run their application, and often these requests are not accurate [27], which then leads to under-utilization of cluster [28]. So for an energy-efficient system, it is required to know the accurate resource requirement of an incoming application to schedule it properly. One such work is presented by Nassereldine at al. [29]. We plan to integrate such prediction systems to enable our scheduler to provide more accurate energy requirements using Runtime and configuration predictions.

2) Integration with FuncX: We plan to integrate this framework into FuncX [13] to allow scalable and flexible remote task execution. Our future approach will schedule tasks across distributed resources by selecting resources according to an energy budget. This way we plan to develop a next-generation energy-aware federated heterogeneous serverless computing framework. Our future scheduling framework will be hierarchical, using funcX to manage global scheduling policies across distributed endpoints and integrate the energy-aware scheduler (presented in this paper) to distribute tasks at the local/cluster level.

3) Add Deep Reinforcement Learning algorithms: We plan to use Deep Reinforcement Learning (DRL) based algorithm so that it makes accurate scheduling decisions at run-time using the feedback loop. The new algorithm will use the current node usage and energy consumption to predict future energy resource allocation so that future energy consumption is minimized. It will use Deep Neural Networks to make accurate energy consumption estimates instead of relying on profiled/historical data.

VII. Related Work

Brondolin and Santambrogio propose Power REgulator for Service Time Optimization (PRESTO), a latency-aware power-capping orchestrator to manage the power consumption of cloud-native applications properly [30]. This paper defines an Observe Decide Act (ODA) loop to manage power consumption and the average latency of microservice-based workloads by considering all the network interactions between microservices in the cluster.

Townend et al. propose an architecture for an energy management system (EMS) based on Kubernetes container cluster to resolve the issue of traditional EMSs' inability to achieve high reliability and resource utilization simultaneously [31]. Using container cluster technology, applications can be isolated and deployed, solving the problem of low system reliability due to interlocking failures. A dynamic Pod fault-tolerant EMS model is proposed using discrete Markov theory.

Li et al. discuss the intricate nature of data centers and the service models utilized in the industry [32]. The authors propose a comprehensive scheduling system that replaces the default scheduler in the Kubernetes container system, which considers both software and hardware models.

James and Schien outline the development and execution of a low-carbon scheduling policy for the open-source Kubernetes container orchestrator [33]. The scheduler is utilized in demand-side management, transferring the consumption of electric energy to countries with the least carbon intensity of electricity.

Ghafouri et al. introduce a deep reinforcement learning-based method called Mobile-Kube to reduce the latency of Kubernetes applications on mobile edge computing devices while maintaining energy consumption at a reasonable level [34]. The findings indicate that learning-based methods can replace conventional heuristic algorithms, such as bin-packing, to achieve comparable results for the targeted objective while enhancing performance in other aspects.

Kaur et al. discuss implementing a Kubernetes-based energy and interference-driven scheduler (KEIDS) for managing containers on edge-cloud nodes, considering carbon footprint emission, interference, and energy consumption [35]. KEIDS utilizes integer linear programming for task scheduling, optimizing the energy utilization of edge-cloud nodes to promote green energy utilization while ensuring optimal performance for end-users with minimum interference from other applications.

Douhara et al. introduce a customized Kubernetes scheduler, WAO-scheduler, which utilizes neural networks to minimize power consumption in computational resource allocation for the Kubernetes cluster [36]. The work also presents WAO-LB, a solution that accounts for both power-saving and response time necessities for edge computing.

Menouer introduces a new Kubernetes Container Scheduling Strategy (KCSS) that optimizes the scheduling of containers submitted online to improve performance for both the user and cloud provider in terms of makespan and power consumption [37]. KCSS utilizes a multi-criteria selection approach that considers cloud infrastructure and user needs to select the best node for each container.

Arnaboldi and Brondolin propose HyPPO, an orchestrator for Kubernetes and Docker environments that optimizes performance and power consumption of workloads by exploiting Dynamic Voltage and Frequency Scaling techniques [38]. It is based on a distributed ODA control loop that enforces a power cap while considering its impact on workload performance and aims to exploit the opportunity gap of OLDI workloads.

Khullar and Hossain propose an algorithm, Dynamic Voltage and Frequency Scaling (DVFS) enabled Efficient energy Workflow Task Scheduling (DEWTS), that uses DVFS for energy-efficient task scheduling in green cloud operation [39]. DEWTS optimizes slack time by merging inefficient processors and computes deadlines based on the finish time of the tasks heterogeneously while utilizing idle CPU slots without violating any constraints.

Gunasekaran proposes and evaluates Fifer, an adaptive resource management framework with energy and stage awareness [40]. It aims to run function chains on serverless platforms

efficiently while ensuring high container utilization and cluster efficiency without compromising on SLOs. Fifer utilizes a novel combination of stage-wise slack awareness and proactive container allocations that rely on an LSTM-based load prediction model. The proposed technique can smartly scale out and balance containers for every stage.

Jia and Zhao propose reducing energy consumption in workloads by managing energy consumption in the startup, runtime, and idle stages, considering the energy fungibility phenomenon [41]. The authors introduce RAEF, a resource allocator that proactively adjusts the functions' resources to minimize energy consumption while maintaining SLA guarantees.

Aslanpour proposes energy-aware resource scheduling algorithms to place functions on edge nodes powered with renewable energy sources, aiming to maximize the operational availability of edge nodes while minimizing their variation [42]. Techniques such as sticky offloading and warm scheduling were proposed to reduce recurrent function replacements.

Rocha et al. present a new orchestrator, Heats, designed to manage containerized applications on diverse clusters in a task-specific and energy-efficient way [43]. The system is capable of balancing performance and energy requirements and it first learns the performance and energy-related characteristics of the physical hosts. Then, it monitors task execution on these hosts and migrates them to different nodes in the cluster to optimize the deployment according to customer requirements.

Dhakal et al propose GSLICE [44], spatial sharing of the GPU to increase GPU utilization. GPU is partitioned using NVIDIA MPS, and applications are provided with GPU resources (GPU%) to meet the deadline for applications running in the GPU. Laius also utilizes spatial GPU sharing, with multiple applications running in GPU concurrently with predefined GPU% [45]. Whenever a request to run a GPU kernel arrives, Laius loads the kernel's binary (PTX) file using the application with the required GPU%. Both GSLICE and Laius show that spatially sharing the GPU increases the overall GPU throughput.

Most of the existing work has been around replacing the existing Kubernetes scheduler and providing a more efficient solution. While a part of our work in this paper also implements a similar solution, we do not replace the existing Kubernetes scheduler, and both can be run simultaneously. Based on user requirements, our framework can schedule tasks using either scheduler. Our solution is also unique in that it schedules tasks based on the current energy allocation and the estimate of how much the task would consume if launched on a node. Our framework is also unique in that it can schedule on various hardware types. It also provides ways to optimize hardware like MPS with GPU and Pylog for FPGA.

VIII. Conclusion

Our Heterogeneous Execution Framework with our Kubernetes native energy-aware scheduler provides an effective solution for energy-aware fine-grained task scheduling. One solution does not fit all, and we show that while fine-grained energy-aware scheduling works for CPU tasks, the same may not be the right solution for GPUs. So we have implemented our framework that understands these diverse requirements and schedules tasks through Kubernetes or directly on the bare-metal node using our device plugin. We compare with the current state-of-the-art Kubernetes scheduler as well as with the default implementation of the GPUs. Our framework has the capability to support a wide variety of hardware, unlike current solutions which are limited to either CPU, GPU, or a few FPGAs. With our energy-aware scheduler for Kubernetes, the experiments show a significant reduction in makespan by up to 17.6% and energy utilization by up to 20.16% for CPU workloads. The GPU tasks managed by the Framework and optimized to use spatial sharing of the GPU show a significant improvement in makespan and energy, too, 58.15% and 28.92%, respectively.

References

[1] climateneutralgroup.com, "Carbon emissions of data centers." [Online]. Available: https://www.climateneutralgroup.com/en/news/carbon-emissions-of-data-centers/

[2] A. S. G. Andrae and T. Edler, "On Global Electricity Usage of Communication Technology: Trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, 2015. [Online]. Available: https://www.mdpi.com/2078-1547/6/1/117

[3] IEA, "IEA (2022), Data Centres and Data Transmission Networks, IEA, Paris." [Online]. Available: https://www.iea.org/reports/data-centres-and-data-transmission-networks

[4] H. Cheng, B. Liu, W. Lin, Z. Ma, K. Li, and C.-H. Hsu, "A Survey of Energy-Saving Technologies in Cloud Data Centers," *J. Supercomput.*, vol. 77, no. 11, pp. 13 385–13 420, 11 2021. [Online]. Available: https://doi.org/10.1007/s11227-021-03805-5

[5] D. Milojicic, "Accelerators for Artificial Intelligence and High-Performance Computing," *Computer*, vol. 53, no. 2, pp. 14–22, 2 2020.

[6] "Kubernetes." [Online]. Available: https://kubernetes.io/

[7] M. Frampton, "Apache Mesos," in *Complete Guide to Open Source Big Data Stack*. Berkeley, CA: Apress, 2018, pp. 97–137.

[8] "Apache OpenWhisk: Open Source Serverless Cloud Platform." [Online]. Available: https://openwhisk.apache.org/

[9] "AWS Lambda." [Online]. Available: https://aws.amazon.com/lambda/

[10] "Knative." [Online]. Available: https://knative.dev/docs/

[11] "Multi-Process Service :: GPU Deployment and Management Documentation." [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[12] "Multi-Instance GPU (MIG) — NVIDIA." [Online]. Available: https://www.nvidia.com/en-us/technologies/multi-instance-gpu/

[13] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "FuncX: A Federated Function Serving Fabric for Science," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 65–76. [Online]. Available: https://doi.org/10.1145/3369583.3392683

[14] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann, R. Singh, T. Gunarathne, E. Chinthaka, R. Gardler, A. Slominski, A. Douma, S. Perera, and S. Weerawarana, "Apache Airavata: A Framework for Distributed Applications and Computational Workflows," in *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, ser. GCE '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 21–28. [Online]. Available: https://doi.org/10.1145/2110486.2110490

[15] "Fission: Serverless Functions for Kubernetes." [Online]. Available: https://fission.io/

[16] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rFaaS: Enabling High Performance Serverless with RDMA and Decentralization," 2022.

[17] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-M. Hwu, "PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2015–2028, 2021.

[18] "Performance Co-Pilot." [Online]. Available: https://pcp.io/
[19] "Ubuntu Manpage: stress-ng - a tool to load and stress a computer system." [Online]. Available: https://manpages.ubuntu.com/manpages/bionic/man1/stress-ng.1.html
[20] "Running Average Power Limit Energy Reporting / CVE-2020-8694 , CVE-2020-8695 / INTEL-SA-00389." [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html
[21] C. Bienia, S. Kumar, J. Pal Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," 2008.
[22] J. R. Vaughan and G. R. Brookes, "The Mandelbrot Set as a Parallel Processing Benchmark," *Univ. Comput.*, vol. 11, no. 4, pp. 193–197, 12 1989.
[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
[24] Numba, "Mandelbrot set computation using numba and cuda," https://numba.pydata.org/numba-doc/dev/user/examples.html#id1, 2023.
[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[26] "AWS Lambda enables functions that can run up to 15 minutes." [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/
[27] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2019-July, pp. 33–40, 7 2019.
[28] G. Rattihalli, M. Govindaraju, and D. Tiwari, "Towards enabling dynamic resource estimation and correction for improving utilization in an apache mesos cloud environment," *Proceedings - 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2019*, pp. 188–197, 5 2019.
[29] Amir Nassereldine, Safaa Diab, Mohammed Baydoun, Kenneth Leach, Maxim Alt, Dejan Milojicic, and Izzat El Hajj, "Predicting the Performance-Cost Trade-off of Applications Across Multiple Systems," *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2023.
[30] R. Brondolin and M. D. Santambrogio, "PRESTO: a latency-aware power-capping orchestrator for cloud-native microservices," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 11–20.
[31] P. Townend, S. Clement, D. Burdett, R. Yang, J. Shaw, B. Slater, and J. Xu, "Invited Paper: Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 156–15 610.
[32] Z. Li, H. Wei, Z. Lyu, and C. Lian, "Kubernetes-Container-Cluster-Based Architecture for an Energy Management System," *IEEE Access*, vol. 9, pp. 84 596–84 604, 2021.
[33] A. James and D. Schien, "A Low Carbon Kubernetes Scheduler," in *ICT for Sustainability*, 2019.
[34] S. Ghafouri, A. Karami, D. B. Bakhtiarvan, A. S. Bigdeli, S. S. Gill, and J. Doyle, "Mobile-Kube: Mobility-aware and Energy-efficient Service Orchestration on Kubernetes Edge Servers," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022, pp. 82–91.
[35] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4228–4237, 2020.
[36] R. Douhara, Y.-F. Hsu, T. Yoshihisa, K. Matsuda, and M. Matsuoka, "Kubernetes-based Workload Allocation Optimizer for Minimizing Power Consumption of Computing System with Neural Network," in *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2020, pp. 1269–1275.
[37] T. Menouer, "KCSS: Kubernetes container scheduling strategy," *The Journal of Supercomputing*, vol. 77, no. 5, pp. 4267–4293, 2021. [Online]. Available: https://doi.org/10.1007/s11227-020-03427-3
[38] M. Arnaboldi, R. Brondolin, and M. D. Santambrogio, "HyPPO: Hybrid Performance-Aware Power-Capping Orchestrator," *2018 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 71–80, 9 2018.
[39] R. Khullar and G. Hossain, "A New Algorithm for Energy Efficient Task Scheduling Towards Optimal Green Cloud Computing," in *2022 IEEE/ACS 19th International Conference on Computer Systems and Applications (AICCSA)*, 2022, pp. 1–6.
[40] 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*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 280–295. [Online]. Available: https://doi-org.ezproxy2.library.colostate.edu/10.1145/3423211.3425683
[41] X. Jia and L. Zhao, "RAEF: Energy-efficient Resource Allocation through Energy Fungibility in Serverless," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, 2021, pp. 434–441.
[42] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and R. Gaire, "Energy-Aware Resource Scheduling for Serverless Edge Computing," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 190–199.
[43] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Heats: Heterogeneity-and Energy-Aware Task-Based Scheduling," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 400–405.
[44] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "Gslice: controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 492–506.
[45] W. Zhang, W. Cui, K. Fu, Q. Chen, D. E. Mawhirter, B. Wu, C. Li, and M. Guo, "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *Proceedings of the ACM international conference on supercomputing*, 2019, pp. 58–68.