Energy Consumption of Heuristic Kubernetes Schedulers

Alfred Daimari, Matthijs Jansen, Daniele Bonetta
Vrije Universiteit, Amsterdam, the Netherlands

***Abstract*—Optimizing and understanding energy consumption is essential for cost reduction in modern cloud native environments. This paper evaluates the energy consumption of the Kubernetes Default Scheduler and two heuristics based open source schedulers, Descheduler and Poseidon under realistic workloads. This study contributes by providing actionable insights by evaluating the energy efficiency and consumption of schedulers that use predefined static constraints, like Descheduler, against Poseidon's dynamic scheduling approach. Furthermore, we explore the potential of integrating Service Level Objectives (SLOs) into our chosen schedulers to further improve energy efficiency.**

I. Introduction

As cloud-native environments continue to scale, optimizing resource usage, particularly energy consumption, has become a key priority for reducing operational costs [17]. Kubernetes is the leading platform for container orchestration in the cloud, and the scheduler used within a Kubernetes cluster plays a central role in managing resources. The scheduling strategies that various schedulers use significantly impact the energy consumption of the cluster. One such example is the Heats scheduler, which was able to create energy savings of upto 8.5% [18]. Hence, evaluating the energy consumption of available Kubernetes schedulers is key in helping make informed decisions and gaining insights into operational costs of Kubernetes clusters.

Existing surveys [4, 8] on Kubernetes scheduling algorithms primarily focus on performance metrics like throughput, latency, and task completion time. Energy consumption often receives less attention in Kubernetes scheduler surveys. The publication [1] that introduces the Firmament scheduler (part of Poseidon scheduler) only analyzes task response times and runtimes, and lacks an evaluation of energy consumption. This report contributes by evaluating the energy consumption of two open-source heuristic schedulers, which are the Descheduler [5]—a well-known scheduler in the Kubernetes community with 4600 Github stars—and Poseidon [6]—the only open-source Kubernetes scheduler based on the novel Firmament scheduler [1]. The goal of this report is to contribute to the growing body of research on energy-efficient Kubernetes scheduling. In doing so, it seeks to further the understanding of how different scheduling strategies can impact energy consumption. To our knowledge, there is no research on energy consumption for the schedulers we have chosen.

Integrating SLOs into scheduling strategies can significantly contribute to energy efficiency [3]. By dynamically adjusting resource allocations based on real-time workload demands and pre-defined SLOs, systems can reduce overprovisioning and idle resource usage. Additionally, our report explores whether integrating Service Level Objectives (SLOs) into our chosen schedulers can further increase energy efficiency without significantly decreasing performance.

The report is structured as follows:

- Section II describes why heuristic schedulers are important in the context of energy efficiency. It also contains a description of how the Poseidon, Default and Descheduler schedulers function and how their scheduling decisions may affect energy efficiency.
- Section III contains a description, specifics and differences of the benchmarks used to evaluate the schedulers.
- Section IV outlines the experimental design used to evaluate the schedulers, ensuring reproducibility of results.
- Section V presents an in-depth analysis of the results, discussing the performance, energy efficiency, and energy consumption of the schedulers.
- Section VI summarizes the findings and offers insights into the implications of the study for Kubernetes-based resource management and energy optimization.

II. Kubernetes Schedulers

Kubernetes schedulers can be classified into four types based on the optimization techniques they use to find the best pod placement, which are mathematical modeling, heuristics, metaheuristics, and machine learning [4]. Mathematical modeling schedulers model the scheduling problem as ILP problems (Integer Linear Programming) and are not scalable [4]. ML-based schedulers also struggle with pod placement latency in large Kubernetes clusters which run huge numbers of microservices due to their computing overhead [8] thereby directly affecting quality of service. For large-scale scheduling of microservices, in theory, metaheuristic schedulers—schedulers that use population-based optimization algorithms inspired by the intelligent processes and behaviors arising in nature [19]—should have comparatively lower

computation requirements and be scalable. However, in a container scheduling survey [4] conducted by Ahmad et al., it was observed that none of the ML, ILP, and meta-heuristic schedulers had scalability as an objective, whereas 7 of 20 heuristic schedulers had it.

Heuristic-based schedulers are the most scalable solutions for managing large-scale clusters due to their low scheduling overhead. They are widely used in large clusters, where the primary goal is rapid decision making. Their large scale makes them ideal candidates for energy-efficient enhancements because even small reductions in energy consumption can lead to substantial savings in operating costs. Their scale presents opportunities for substantial energy savings. This positions heuristic-based Kubernetes schedulers as an important focus for research, evaluation, and innovation regarding energy consumption. This is why we chose to evaluate the energy consumption of two heuristic-based schedulers.

2.1 Default Scheduler

The Default scheduler, also called Kube-scheduler, is the baseline scheduler in our evaluation. It uses resource requests, node availability, and affinity rules to schedule a pod on a node. The goal of the scheduler is to filter out nodes that do not meet the pod's resource requests. The scheduler selects a node in a two-step operation, filtering and scoring. Filtering and scoring are implemented by two types of components: Predicates and Priorities. Predicates indicate whether a node can run a pod or not, whereas Priorities score a node between 0 and 10. Kubernetes has a default set of Predicates and Priorities that cover a fair amount of common use cases. The Default scheduler uses Predicates to create a list of available nodes. It then scores (0 - 10) the nodes using Priorities. The node with the highest score is given preference for running the pod.

However, some issues may arise in the Default scheduler regarding efficient use of resources. The scheduling policy used by the Default scheduler only influences pod placement when a pod is created. This initial snapshot of the cluster when a new pod arises may no longer hold true as the cluster continues to run. Some of the reasons why pods need to be moved around are the following.

1. Nodes are over or underutilized
2. Taints and labels are added or removed from the nodes, pod/node affinity requirements are not satisfied anymore
3. Some nodes failed and pods are moved to other nodes
4. New nodes are added to the cluster
5. Several pods may be moved to less desirable nodes

In the Default scheduler, the Kubelet—the "node agent" that manages Kubernetes pods—in each node is solely responsible for pod eviction. The pod is evicted when it uses more resources than the amount it initially requests. Post-eviction, the pods enter a pending state and are then reassigned to a different node by the scheduler. Each Kubelet is unable to perform optimal evictions because it lacks visibility of the overall cluster state and

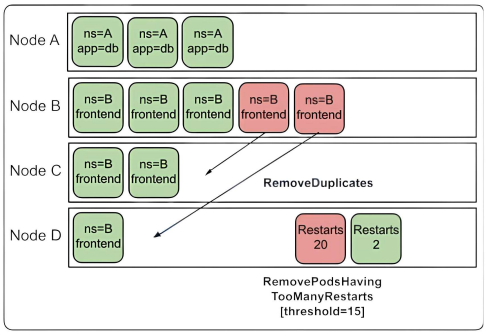


Figure 1: Descheduler using *RemoveDuplicates* strategy

makes decisions based on its own local state. This can lead to inefficient use of resources and a decrease in energy efficiency.

2.2 Descheduler

The issues mentioned in the previous section 2.1 hint at a need for a global pod eviction strategy for the cluster. This is what the Descheduler [5] tries to improve upon. The Descheduler's main goal is to maintain "a balanced and optimal cluster" by evicting running pods from nodes even if they do not violate resource requests. The definition of what a balanced and optimal cluster is depends on the strategy used. An example of *RemoveDuplicates* strategy is shown in Figure 1.

For the benchmarks in this report, two strategies were used, *lowNodeUtilization* and *highNodeUtilization*. Both strategies come standard with the scheduler and are the only strategies which our benchmark can support.

2.2.1 Low Node Utilization

This strategy finds nodes that are underutilized and tries to schedule pods onto them. The goal of this strategy is to maintain a uniform utilization across all nodes in the cluster. It first finds nodes that are overutilized and evicts pods from them, hoping that these evicted pods would get scheduled onto underutilized nodes by the Default scheduler. There are two configurable parameters in this strategy; they are called thresholds and *targetThresholds*.

If a node's usage is below the set threshold, for all (CPU, memory, number of pods, etc.), the node is considered underutilized. The requested resource requirements of a pod, which are defined in its manifest file, are considered to calculate the resource utilization of a node.

targetThresholds is used to calculate potential nodes from where pods could be evicted. If the usage of a node is above the set *targetThresholds*, the node is considered over-utilized. Any node whose resource utilization is between thresholds and *targetThresholds* is considered appropriately utilized or balanced and is not considered for eviction. The thresholds and *targetThresholds* can be configured for CPU, memory, and the number of pods.

2.2.2 High Node Utilization

This strategy finds underused nodes and evicts pods from these nodes. The goal of this strategy is to compactly schedule pods onto a few nodes and prevent a sparse pod

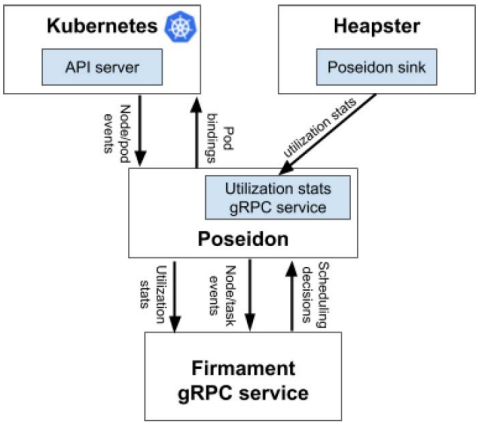


Figure 2: Poseidon scheduler architecture

distribution across the cluster. It has only one configurable parameter, which is thresholds. The underutilization of nodes is determined by the set thresholds, which can be configured for CPU, memory, and the number of pods. Resource utilization is calculated as the current resources requested on the node, which are defined in each pod's manifest file.

2.2.3 Limitations

In real-world scenarios, workloads are often bursty, characterized by sudden and unpredictable traffic spikes. These abrupt changes are challenging to anticipate using basic metrics and typically require advanced forecasting techniques, such as machine learning and deep learning models [20]. When workloads exhibit significant variability, the Descheduler's strategies may prove inefficient, as their parameters are often tuned based on assumed workload patterns that may not align with such erratic behavior. This could lead to a decrease in energy efficiency.

2.3 Poseidon Scheduler

The limitations of a scheduler that uses preset scheduling parameters, as described in 2.2.3 hint at a need for an adaptive scheduler that optimizes its actions based on real-time data. The Poseidon scheduler [6] solves the issue of optimizing bursty workloads by using a minimum-cost flow network. As shown in Figure 2, the Poseidon Kubernetes scheduler comprises two components. The Poseidon component is responsible for managing the pods in the cluster, and the Firmament component implements a flow network.

The Poseidon component also keeps track of real-time changes in the cluster and maintains a global view of the entire cluster and updates the flow network in the Firmament component. After the Firmament component optimizes the flow network, the Poseidon component starts placing or removing pods from different nodes in the cluster.

2.3.1 Firmament Scheduler

The Firmament Scheduler was introduced in the paper Firmament: Fast and centralized cluster scheduling at scale [1]. It is a centralized scheduler that can scale to 10,000 machines at subsecond latency. The Firmament scheduler assigns multiple jobs to the nodes using a batch approach. Once it has a list of jobs, it maps all available

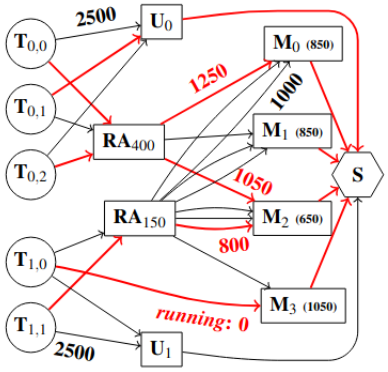


Figure 3: Firmament: Resource-aware Flow Network

jobs to nodes using a flow network. Each job-to-node assignment is listed as an edge in the flow network with a defined cost. Using a min-cost flow network, it tries to find the most cost-effective strategy to place the jobs onto the cluster.

The firmament scheduler has three policies: load-spreading policy, quincy policy, and network-aware policy, but the Poseidon scheduler has extended the network-aware policy to handle CPU and memory. For the benchmarks in this report, a multidimensional CPU-memory cost model was used.

2.3.2 Firmament Resource-Aware policy

Overcommitting a node's network bandwidth, CPU, or memory of a node leads to slower response times. Figure 3 shows a diagram of how a flow network is set up for Poseidon's resource-aware policy for the CPU, network, or memory. Every task or work, represented by T_{ij}, is directed to an RA or resource aggregator. The RAs have edges connected to nodes that are represented by M_{ij}. For every job assigned to the machine, an edge is drawn from the RA to the node. The weight of the edge is the amount of resource used by the job. This weight is updated dynamically as the use of resources changes over time. The jobs are only assigned by the RA when there are enough resources on the node. Unscheduled jobs are stored in vertices tagged with U_i.

III. Benchmark Design

Many publications on Kubernetes scheduling mechanisms [10, 9] use the Sock Shop benchmark [7], but it only includes ten microservices and does not reflect real-world workloads [21]. While other benchmark suites that have been released by academia and industry [12, 13, 14] exist, they focus on simpler single-tier or small-scale applications, which differ significantly from modern cloud deployments [2]. To effectively evaluate the energy consumption of Kubernetes schedulers and to gain concrete actionable insights, a benchmark with complex and tightly coupled microservice dependencies is needed. DeathStarBench [2] which is being developed by the SAIL group at Cornell University, addresses this by providing diverse, end-to-end systems like Hotel Reservation, Media Microservices, and Social Network, which reflect the complexity of real-world cloud environments. The sub-

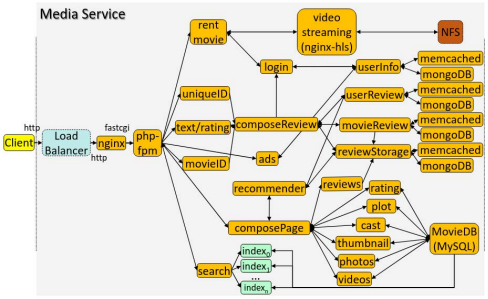


Figure 4: Dependency graph: Media Service

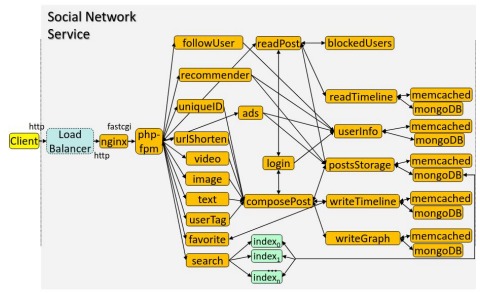


Figure 5: Dependency graph: Social Network Service

sections below(3.1-3.3) describe how each of these end-to-end systems differ from each other.

3.1 Microservice count

The various microservices Hotel Reservation includes are user management, user profile management, a recommendation system, ratings, reviews, hotel search, and a front-end service. Most of these services have their own respective MongoDB and memcached services to handle their storage and caching requirements. It consists of 12 microservices.

As shown in Figure 4, Media Service consists of microservices that allow users to search and browse information about movies, including their plot, photos, videos, cast, and review information. Users can also insert new reviews for a specific movie. It also consists of an authentication module. It also includes a recommendation system and auxiliary services that are not shown in the figure. As of now, the benchmark hasn't yet implemented the streaming service. It consists of about 38 unique microservices.

As shown in Figure 5, Social Network consists of microservices that for allow composing and displaying posts, for advertisements, search engines, etc. Users can create posts embedded with text, media, and links. The system also includes machine learning plugins, such as ads and recommendation engines, a search service using Xapian, and microservices to record and display user statistics. It consists of 36 unique microservices.

3.2 Communication

Hotel Reservation consists mainly of synchronous communication for tasks like booking management and payment processing, with some asynchronous messaging for background tasks.

Social Network consists of a mix of synchronous and asynchronous communication, with real-time messaging, notifications, and background processes handling interactions, updates, and notifications.

Media Service consists of only synchronous calls that are made for user interactions such as media browsing, reading reviews, and purchasing.

Table 1: Hotel Reservation - Route Request Probability

Service Name	Request probability
search	0.6
recommend	0.39
user	0.005
reserve	0.005

3.3 Throughput and Traffic

Hotel Reservation generates relatively low data throughput, as user interactions are mostly transactional and involve fewer data points per user session.

Social Network generates high data throughput due to dynamic user interactions and content updates, with frequent changes in user-generated data (posts, comments, likes, etc.).

Media Service generates a medium-level of data throughput as it consists mainly of reading movie reviews, recommendations, and ads.

3.4 Workload

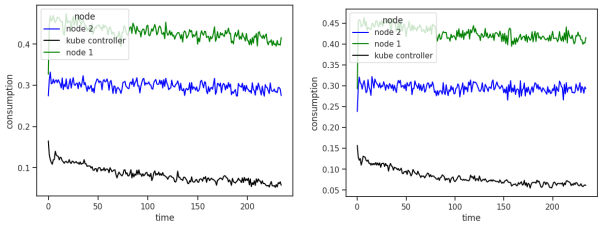
Each end-to-end service in the benchmark generates its workload in a fairly similar fashion. Every available HTTP route in an end-to-end service is given a probability of being requested. For each benchmark, the set probability for each HTTP route mimics real-world API calls [2]. The workload generator then allows us to set the number of requests per second. One example of the set probabilities for service HTTP routes for the hotel reservation benchmark is given in table 1.

IV. Experimental Setup

The experiments were run on a Kubernetes cluster of 3 virtual machines that was created using the continuum framework [11]. The host system is running Ubuntu 22.04.5 LTS, powered by an Intel Xeon Silver 4210R processor, with 20 cores running at 3.2 GHz and 256 GB of memory. Each VM is running Ubuntu 20.04, with 4 cores and 16 GB of memory. The VM which is named 'kube-controller' in our graphs is the control plane node and only houses all the components necessary for cluster orchestration and management.

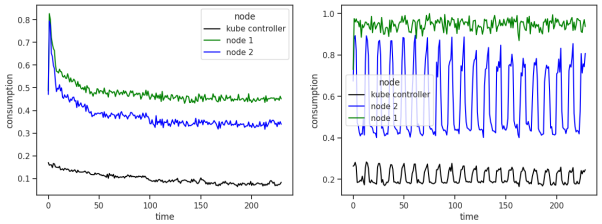
The energy data from the VMs was collected using *Scaphandre* [16]. The key metrics collected were:

- Energy consumption for each node in microwatts.
- CPU Usage upto pod level.
- Memory Usage upto pod level.
- Total requests completed in 20 minutes.



(a) Default scheduler (b) Descheduler

Figure 6: Energy Consumption (Hotel Reservation Benchmark): x-axis represents time(s), y-axis represents energy consumption in microwatts (normalized) at time t.



(a) Hotel Reservation (b) Media Microservices

Figure 7: Energy Consumption (Poseidon Scheduler): x-axis represents time(s), y-axis represents energy consumption in microwatts (normalized) at time t.

Each workload was run for a duration of 20 minutes and executed three times. The requests per second were set at 100, 500, and 1000. The number of threads used to create these requests in the host system was set to 50. The HTTP requests in each workload were directed to the kube-controller VM where, for each benchmark, the Nginx service or the front-end service was exposed using NodePort. For each benchmark, for every service deployment, the number of replicas was set to one. To check whether service level objectives could be integrated into our selected schedulers, workloads were run when the VM's *vcpu_quota* was set at 50%, 80% and 90% using virsh.

To replicate the experiments described in this paper, we have made our implementation and data publicly available on GitHub [15]. The repository [1] contains the source code, configuration files, and detailed instructions for reproducing the results.

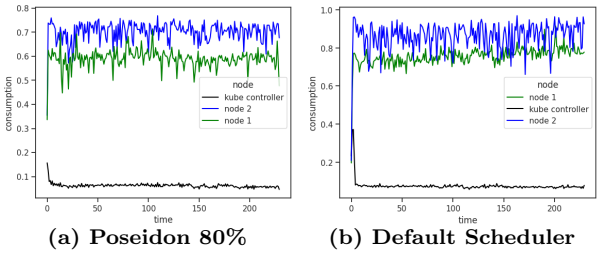
V. Evaluation[2]

Observation 1: Performance and energy consumption of the Default scheduler and Descheduler are identical.

This is caused by the technique Descheduler uses to remove the pods. As mentioned above, the Descheduler uses the resource requests defined in the manifest file to schedule pods. Since the defined limits do not exceed the thresholds defined for the lowNodeUtilization strategy, scheduling does not occur during the entire workload. In addition, Descheduler uses the default scheduler for the initial placement of the pod, which is why its energy consumption metrics are identical. For the Descheduler to function correctly, the resource requests in the pods need to be aligned with the expected load during runtime. Figure 6 shows the similar energy consumption of the Default and Descheduler schedulers, respectively. Table 2 shows the performance of both schedulers.

Table 2: Total requests completed (20mins) and tail latency (99%) for different schedulers and benchmarks

Scheduler	Benchmark	Requests	Latency (ms)
Default	Hotel	697134	8.3
Default	Media	458754	3.16
Default	Social	71517	17.53
Poseidon	Hotel	740117	7.62
Poseidon	Media	521431	11.25
Poseidon	Social	71108	17.81
Poseidon	Media 80%	421964	1.82
Descheduler low	Hotel	689512	8.48
Descheduler high	Hotel	691177	8.41



(a) Poseidon 80% (b) Default Scheduler

Figure 8: Energy Consumption (Media Microservices Benchmark): x-axis represents time(s), y-axis represents energy consumption in microwatts (normalized) at time t.

For lowNodeUtilization, the thresholds were set between 35 and 50. Even though the CPU usage goes beyond the thresholds set when workloads are run with around 1000 requests per second, the Descheduler does not move the pods around.

Similar behaviour is seen in the *highNodeUtilization* strategy. This is because DeathStarBench's resource requests defined in the manifest file are above the high-node utilization's CPU threshold, which is set at 20. This behavior of the Descheduler is seen across all benchmarks.

Observation 2: The Poseidon scheduler is not energy efficient when microservices are not heavily coupled.

As shown in Table 2, Poseidon outperforms the Default scheduler with 6% more requests handled in the hotel benchmark. The hotel benchmark only has about 12 unique microservices and consists of simple user interactions. Here, Poseidon handles 6% more requests but uses 13% more energy than the default scheduler. Refer to Table 3 for efficiency metrics. When microservices are not heavily coupled, the Default scheduler is able to achieve high throughput and utilization without any optimizations. When Poseidon tries to further optimize for more throughput, it leads to inefficient energy usage.

[1] github link: https://github.com/AlfredDaimari/kubernetes-energy

[2] The y-axis of the energy consumption graphs have been normalized to a common scale, allowing for direct comparison across all energy consumption graphs.

Observation 3: Under low traffic (upto 500 requests per second), energy consumption and performance is fairly similar across all schedulers.

Up to 500 requests per second for each benchmark, energy consumption is fairly identical, and there does not exist a substantial difference in energy consumption among schedulers. When CPU and RAM resources are not heavily utilized, Poseidon maintains a balanced distribution of pods (similar to the Default scheduler) and does not move pods around because worker nodes are still underutilized.

Observation 4: Poseidon is best suited for services number of unique microservices is large and just a few services receive bulk of the traffic.

Poseidon outperforms the Default scheduler and is more energy efficient in cases where the number of unique microservices is fairly large like the Media benchmark which has 38 microservices. Poseidon is able to move related pods to the same node, maximizing CPU utilization, leading to better throughput and energy efficiency. The Poseidon scheduler handles 13% more requests but uses only 2% more energy. Refer to Figures 7(b) and 8(b) for the respective energy consumption of the Poseidon and the default schedulers for the media microservices benchmark. As shown in Table 3, Poseidon is 11% more energy efficient than the Default scheduler.

Observation 5: When requests transition from multiple microservices to a cache, Poseidon consumes similar energy to the Default scheduler, as the dependency on managing distributed microservices diminishes.

As shown in Table 2, the performance of Poseidon and the Default scheduler is comparable for the Social Network benchmark, despite it comprising 36 unique microservices. This similarity arises because, after a few minutes of runtime, all traffic is directed to the Redis cache service.

Observation 6: Poseidon scheduler cannot schedule all pods consistently.

In the hotel reservation benchmark, it was not able to schedule 1-3 pods in some runs. Poseidon was constantly marking these pods as unscheduled. The firmament scheduler constantly put these pods in its U vertices of its min-cost max-flow graph. These remaining unscheduled pods had to be scheduled using the Default scheduler.

Observation 7: Service Level Objectives can best be integrated with the Poseidon scheduler for those benchmarks where there are numerous microservices but only a few receive most of the traffic.

While the Default scheduler and Descheduler cannot schedule around deliberate throttling down of the CPU's frequency or CPU governor getting changed, the Poseidon scheduler can. Figure 8(a) shows how Poseidon can work around deliberate throttling down of a VM to just 80% CPU capacity and kube-controller to 50% CPU capacity. Compared to the Default scheduler, the throttled Poseidon cluster handles 8% fewer HTTP requests, but uses around 23% less energy, which is substantial. As shown in Table 3, the throttled down Poseidon scheduler is the most energy efficient. It is 13% more energy efficient than the Default Scheduler.

Table 3: Default and Poseidon Scheduler Efficiency Table [3]

Benchmark	Scheduler	Efficiency
Hotel	Default	477.65
Hotel	Poseidon	446.33
Media	Default	161.68
Media	Poseidon	180.11
Media 80%	Poseidon	183.81

VI. Conclusion

This study highlights the varying effectiveness of Kubernetes schedulers—Default, Descheduler, and Poseidon—under different workload characteristics. Dynamic schedulers like Poseidon demonstrate significant energy efficiency improvements when deployed in environments which are tightly coupled and consist of a large number of unique microservices, especially when traffic is unevenly distributed across services. However, in decoupled systems or scenarios where caching reduces interdependencies, like the Social Network benchmark, the benefits of such schedulers are limited. Notably, using Service Level Objectives (SLOs) enhances the optimization potential for schedulers; the Poseidon scheduler ended achieving up to 23% savings and 13% greater energy efficiency while keeping similar performance to the Default scheduler. These insights underline the importance of selecting schedulers based on workload properties to maximize energy savings and operational efficiency in Kubernetes-managed systems.

References

[1] I. Gog (2016). "*Firmament: fast, centralized cluster scheduling at scale*", M. Schwarzkopf, A. Gleave, R. Watson, S. Hand, Usenix 2016.

[2] Y. Gan (2019). "*An Open-Source Benchmark Suite for Microservices & Their Hardware-Software Implications for Cloud & Edge Systems*", Y. Zhang, D. Cheng, Cornell 2019.

[3] M. Savasci (2024)."*SLO-Power: SLO and Power-aware Elastic Scaling for Web Services*". UMass 2024.

[4] I. Ahmad (2024). "*Container scheduling techniques: A Survey and assessment*", A. AlMutawa, L. Alsalman, Science Direct 2020.

[5] Kubernetes Sigs (2020). "*Descheduler*".

[6] Kubernetes Sigs (2020). "*Poseidon-Firmament*".

[7] P. Bastide (2020). "*Sock Shop*", C. Bade, V. Adhirwadkar.

[8] Yi Sun (2023). "*A Review of Kubernetes Scheduling and Load Balancing Methods*", H. Xiang, Q. Ye, J.

[3] Efficiency formula = Total Requests Completed/Total Energy Consumed (watt)

Yang, M. Xian, H. Wang, IEEE 2023.

[9] Q. Chen (2024).“*Optimal Resource Allocation Using Genetic Algorithm in Container-Based Heterogeneous Cloud*”, C. Wen, IEEE 2024.

[10] C. Guerrero (2024). “*Genetic algorithm for multiobjective optimization of container allocation in cloud architectures*”, I. Lera, C. Juiz, arXiv 2024.

[11] M. Jansen (2023), “*Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum*”, L. Wagner, A. Trivedi, A. Iosup, ICPE 2023.

[12] M. Ferdman (2012). “*Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern hardware.* ” A. Adileh, O. Kocberber, ASPLOS 2012.

[13] J. Hauswald (2015). “*Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers.*”, M. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R.G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, J. Mars, ASPLOS 2015.

[14] H. Kasture (2016). “*TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications.*”, D. Sanchez, IISWC 2016.

[15] A. Daimari (2024), “*Kubernetes Energy*”

[16] Hubblo (2022). “*Scaphandre: Energy consumption metrology agent*”

[17] A. Uchechukwu (2014). “*Energy Consumption in Cloud Computing Data Centers*”, K. Li, Y. Shen, IJ-CLOSER 2014.

[18] I. Rocha (2019). “*HEATS: Heterogeneity- and Energy-Aware Task-based Scheduling*”, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, V. Schiavoni.

[19] K. Hussain (2018). “*Metaheuristic research: a comprehensive survey*”, M. Salleh, S. Cheng, Y. Shi, Springer 2018.

[20] A. Rossi (2023). “*Uncertainty-Aware Workload Prediction in Cloud Computing*, A. Visentin, S. Prestwich, K. N. Brown, arXiv 2023.

[21] M. Haytham (2021). “*End-to-End Latency Prediction of Microservices Workflow on Kubernetes: A Comparative Evaluation of Machine Learning Models and Resource Metrics* ”, O. El-Gayar, Research & Publications 2021.