Carbon Aware Serverless System - Group 6

Jaykumar Patel
patel.jay4802@utexas.edu
jnp2369

Afnan Mir
afnanmir@utexas.edu
amm23523

Nidhi Dubagunta
nidhi.dubagunta@utexas.edu
nsd632

Abstract

Serverless computing is a method that enables developers to build and deploy applications without having to provision resources or manage infrastructure. However, existing serverless systems do not consider the carbon footprint of function invocations when allocating or configuring resources, such as memory and vCPU. In this paper, we propose a carbon-aware serverless system that focuses on minimizing the carbon emissions of function invocations while continuing to meet service level objectives (SLOs) by finding an optimal configuration for vCPU count and operational frequency of the vCPU cores. We first perform a measurement study by running a variety of workloads with different resource configurations to understand the relationship between resource usage, energy consumption, and performance. Leveraging these insights, we developed a system that utilizes online regression techniques as well as Dynamic Frequency Scaling (DFS) that attempts to optimize the vCPU count and frequency of each serverless function invocation to minimize carbon emissions while meeting SLOs. This system is integrated into OpenWhisk, an open-source serverless cloud platform, with a custom scheduler that routes an invocation to an invoker running at the optimal frequency with the corresponding vCPU count, given the availability of a warm container. When compared to baselines, our system is able to reduce the amount of energy expended by almost 50% while retaining comparable performance in terms of SLO violations for functions with strong correlations between energy and input features (function input characteristics, vCPU count, and frequency)

1 Introduction

The rise of cloud computing has transformed the space of digital infrastructure, offering higher amounts of scalability, flexibility, and cost-effectiveness compared to on-premise solutions. Within cloud computing, the concept of serverless computing emerged, which further increased efficiency in application development by abstracting the complexities of infrastructure management and resource provisioning.

In a serverless system, when a function is invoked, the system checks for a pre-existing warmed container to host the function. If no warmed container is available, a cold start is initiated, during which the runtime environment is set up, necessary dependencies are installed, and the function code is loaded. After the function is executed, the container can remain idle for a period of time for future invocations before it is killed.

Several existing serverless systems, such as AWS and GCP, couple memory and CPU allocation, assigning a CPU share proportional to the memory requested by the user, or using a preset resource allocation option [2]. Serverless systems today also do not incorporate dynamic frequency scaling (DFS), in which the operating frequency of the vCPU cores can be dynamically modified. These policies used for resource allocation are carbon-agnostic, meaning they do not consider the carbon emissions of the function execution and idle containers.

In this paper, we propose a serverless framework that provisions resources based on workload characteristics to minimize the carbon emissions associated with the function invocation, while continuing to meet SLOs, such as total runtime. We introduce the frequency at which vCPUs run as a new resource to allocate per invocation. Our system will optimize the number of vCPUs allocated, as well as the frequency at which the vCPU cores operate. The end-to-end framework consists of an online model that predicts the optimal resource configuration for a given function invocation and a custom scheduler, which attempts to route the invocation to an invoker operating at the optimal frequency, given a warm container.

2 Motivation

According to a study by Microsoft and the Green Software Foundation, data centers and data transmission networks account for almost 1% of energy-related greenhouse

gas emissions, which contribute to rising global temperatures and climate change [3]. By adopting carbon-aware computing, these data centers can play a significant role in decarbonization efforts.

The boxplot in Figure 1, shows the variability in energy consumption (in joules (J)) for different functions with a fixed input, while changing the amount of vCPUs allocated and the frequency at which the cores run. The large range of energy consumption for the same function due to changes in vCPU count and frequency underscores the importance of understanding the relationship between these factors and energy usage. This motivates further study into optimizing vCPU allocation and frequency at a function-level to minimize energy consumption and therefore carbon footprint.

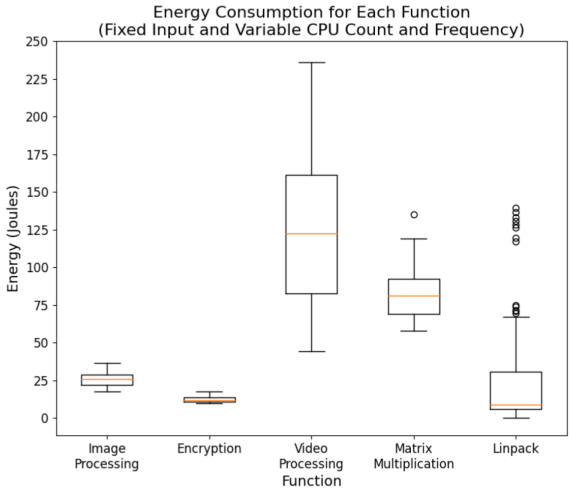


Figure 1: Boxplot showing energy variability across functions with fixed input, while changing vCPU count and frequency.

3 Related Work

CherryPick [1] is a framework that was developed to find optimal or near-optimal cloud configurations (vCPU and memory) for repetitive big data analytics jobs, addressing challenges of system cost, performance, and adaptivity. CherryPick leverages Bayesian Optimization (BO) to build performance models that are able to determine optimal or near-optimal configurations that satisfy performance constraints with a relatively low number of samples.

Aquatope [9] also utilizes Bayesian models to optimize for resource allocation for serverless function invocations. This framework also utilizes BO to learn the mapping from resource configuration to performance and cost, while considering noise and uncertainty in the cloud environment. Aquatope also uses batch sampling to reduce the cost of exploration when finding a resource configuration.

Our work aims to draw from the ML techniques utilized in CherryPick and Aquatope. We plan to use BO to find the near-optimal configuration of vCPU and operational frequency with the objective of minimizing carbon footprint for serverless function invocations. We will also set performance constraints to ensure user-specified SLOs are met.

In the space of carbon-aware computing, GreenCourier [4] is a scheduling framework that enables runtime scheduling of serverless functions across geographically distributed regions based on their carbon efficiencies. This work focuses on minimizing carbon footprint based on the location at which a function is run, whereas our work aims to optimize the host environment through proper resource allocation and configuration to minimize carbon emissions using function-level characteristics.

4 Measuring Carbon Emission

Before conducting the measurement study, we need to define how to measure carbon emissions. According to Gupta et al. [5], the carbon footprint of a server can be calculated using the following formula where OP_{CF} is the operational carbon footprint, CI_{use} is the carbon intensity of the energy source, and $Energy$ is the operational energy:

$$OP_{CF} = CI_{use} \times Energy \quad (1)$$

The carbon intensity refers to how *clean* the energy source is. For example, coal has a higher carbon intensity than solar. We assume that the energy source of the serverless system is constant, and therefore, the carbon intensity is constant. We can then use the operational energy as a proxy for the carbon footprint.

4.1 Measuring Energy

We first need a way to measure the energy consumption of a serverless function invocation. EnergAt [6] is a tool that measures fine-grained energy consumption at the thread-level while taking into account NUMA effects. An EnergAt process continuously samples a target container, where each sample returns energy readings over an interval of at least 50ms. Therefore, EnergAt will get continuous back-to-back samples of energy readings over time. The energy consumption of a target container is the sum of the energy readings over the duration of the function invocation. One limitation of EnergAt is that every target container for which we want to measure energy consumption must be instrumented with its own EnergAt process. This is not feasible as the number of EnergAt processes and resource contention scales with the number of target containers.

To mitigate this issue, we created EnergyLiteDaemon. EnergyLiteDaemon is a daemon that samples multiple target containers in a round-robin fashion, where each sample

Variable	Values
Function Type	Float Matrix Multiplication, Image Processing, Video Processing, Encryption, Linpack
vCPU Count	1, 2, 3, 5, 7, 10, 13, 16, 19, 22, 25, 28, 31
vCPU Frequency (GHz)	1.0 to 2.4 in 0.2 increments

Table 1: Measurement Study Configurations – Energy Variation with vCPU and Frequency Allocation

Variable	Values
Image	main-python, video-ow, mobilenet-ow, sentiment-ow, audio-ow
vCPU Count	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32
Memory (MB)	512 to 10240 in 512 increments

Table 2: Measurement Study Configurations – Energy Variation with vCPU and Frequency Allocation

returns energy readings over an interval of at least 50ms. This allows us to measure the energy consumption of multiple target containers with a single, lightweight, process. Due to the round-robin fashion, however, the sampling frequency per target container decreases as a function of the number of target containers. If there are multiple containers, the samples are no longer back-to-back – there may be some time delay between two samples. To solve this, we complete the following steps to interpolate the total energy consumption of a target container:

Power Calculation: We compute the average power for every sample by dividing the energy consumed over the interval of the sample by the duration of that interval. This allows us to get a power-time scatter plot that contains n points, where n is the number of samples.

Trapezoidal Integration: Given a power-time scatterplot, we compute the total energy consumption of a target container using trapezoidal integration. All of the points in the power-time scatter plot are connected by straight lines, and the area under the curve is calculated. This area is the total energy consumption of the target container.

A toy example of energy measurement using EnergAt and EnergyLiteDaemon is shown in Appendix A.1.

Due to EnergyLiteDaemon's round-robin sampling approach, which reduces its overhead, it may not capture all energy readings for a container. Even with trapezoidal integration, the energy consumption will not be exact. To determine the accuracy of EnergyLiteDaemon, we ran 20 stress docker containers, with varying CPU usage and duration. Figure 2 shows the energy consumption of the containers as measured by EnergAt and EnergyLiteDaemon. The absolute percentage error of EnergyLiteDaemon is less than 15%, which is acceptable for our purposes.

5 Measurement Study

We conduct two measurement studies to answer the following questions: **1. How does energy vary with changes in vCPU and frequency allocation? 2. What is the energy consumption over the lifecycle of a serverless function invocation?**

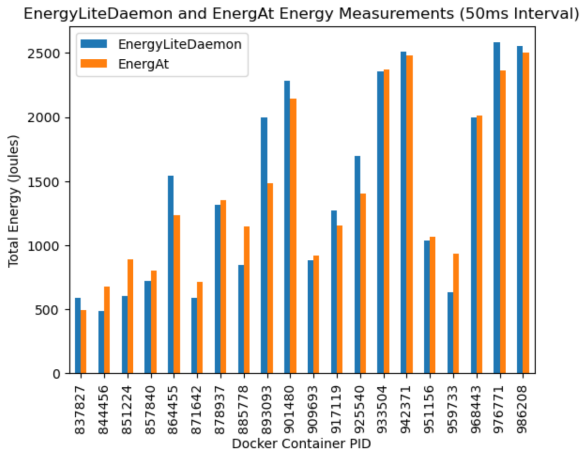


Figure 2: Energy measurements of EnergAt and EnergyLiteDaemon for 20 stress docker containers.

To perform the measurement study, we use OpenWhisk, which is an open-source serverless platform, to run function invocations. We use EnergyLiteDaemon to measure the energy consumption of the function invocations.

5.1 Energy Variation with vCPU and Frequency Allocation

A variety of workloads were run on different vCPU and frequency configurations. We collected the vCPU utilization, energy consumption, and performance in terms of duration. The resource configurations are shown in Table 2.

We then analyzed the data to understand how energy consumption varies with changes in vCPU and frequency allocation. For float matrix multiplication, energy usage initially decreases and then tends to plateau as the vCPU

allocation increases. This is because matrix multiplication is highly parallelizable and has high vCPU utilization, leading to a decrease in duration and energy usage. For image processing, energy usage has no apparent correlation with vCPU allocation, likely because most of the image processing tasks are non-parallelizable. For encryption, energy usage, duration, and vCPU utilization do not change with vCPU allocation because encryption is also non-parallelizable and does not benefit from additional vCPUs.

For both types of functions, energy consumption as a function of frequency displays a convex relationship, where energy consumption decreases with frequency until a certain point (which varies between different functions and function inputs), and then begins to increase again. This is because the energy consumption per time unit of a vCPU is proportional to the frequency at which it runs, but the overall duration of the function is inversely proportional to the frequency at which the vCPU runs.

For functions that are non-paralellizable, we wanted to assess whether the optimal frequency to run an invocation is the same across all function inputs (since vCPU count is not as big of a factor as in parallelizable functions). However, across various inputs. we found that the frequency that results in minimum energy expended varies.

All of these findings support the idea that we should use machine learning to learn the relationship between vCPU, frequency, and energy consumption, and use this model to minimize energy consumption while meeting SLOs.

More details on the results can be found in Appendix A.2.

5.2 Energy Consumption Over the Lifecycle of a Serverless Function Invocation

Additionally, we ran a variety of docker images, each of a different size, with a variety of vCPU and memory configurations to investigate carbon emissions of different stages of the container lifecycle. For each (image, vCPU, memory) configuration, we measured the energy consumption at different stages of the function invocation lifecycle. More specifically, we measured the energy consumed during the container spin-up, the idle time, and the container spin-down.

After collecting and analyzing the data, we first found that for each stage of the lifecycle we measured, when changing the vCPU and memory configurations, the energy consumption did not vary significantly. This was true for all image sizes. This is likely due to the fact that these resources will not actually be allocated to the container until the compute inside the container requires them. We did notice that if we varied the image size, the energy consumption of the container spin-up process significantly increased as the image size increased. However, it is worth noting that most of the energy consumption of the spin-up is accredited to the 'docker pull' process, which would only be a one-time operation. The 'docker run' does not consume a significant amount of energy and does not show any statistical correlation. For the idle time, we noticed a strange negative correlation trend with the image size, and for the spin-down, we noticed a somewhat positive correlation with a few outlier points. Since we only used a few image sizes, we cannot definitively conclude that these trends will extrapolate.

Due to our findings of little to no correlations, we decided to not pursue a carbon-aware keep-alive policy further. More details on the study can be found in Appendix A.3.

6 Final System

Our final system, Carbon Aware Serverless System (CASS), consists of two main components: a machine learning model that predicts the optimal vCPU count and frequency for a given function invocation, and a custom scheduler that attempts to route the invocation to a warm container with or greater than the predicted vCPU count and frequency.

6.1 Machine Learning Model

To implement a machine learning model to minimize carbon emissions, we utilize a Gaussian Process Regressor (GPR), similar to Aquatope[9]. GPR uses the same mathematical foundation as Bayesian Optimization (BO), resulting in it having the same advantages as BO, including not needing a lot of data points to train and having online learning capabilities. In addition, by using the regression model, we can use the vectorized function input as an additional feature for the model and test different resource configurations while fixing the function input to find the minimum energy for a specific invocation. In our final implementation, we utilize the Matern kernel, which offers more flexibility in modeling smoothness and can help better capture relationships between different inputs when the function is not smooth everywhere, and the white kernel, to model noise.

Using our model, we optimize for vCPU allocation and operational frequency to try and minimize carbon emissions, while setting SLOs as our constraint. Since we have collected data on 5 different functions and each function has different usage patterns, we have a separate model for each function. Each function's input is encoded into an input vector for the regression model, and the vCPU count and frequency are concatenated to this input vector.

When a function is invoked, after the input feature vector is constructed, we perform an iterative process in which we go through all possible resource configurations (in terms of vCPU count and frequency) and predict the energy consumption and duration for that specific function and function input given the configuration. Given these predictions, we filter the results that were predicted to violate SLOs, and choose the vCPU count and frequency configuration that minimizes the energy consumption.

Since our model is trained from scratch, early inferences by the model will have lower confidence. To encourage exploration of the feature space in early invocations of a function and allow the model to see a more diverse set of data points for training, we set the value for a confidence threshold that the model prediction should meet. In the case that the prediction is not confident, we sample a random configuration of resources for the invocation to run at.

Simultaneously, we have a background thread that is periodically polling a database that receives observed energy and latency readings from our invokers and uses this as training data to train our models, to utilize online learning. This, however, is not on the critical path.

A diagram of the machine learning model is shown in Figure 3.

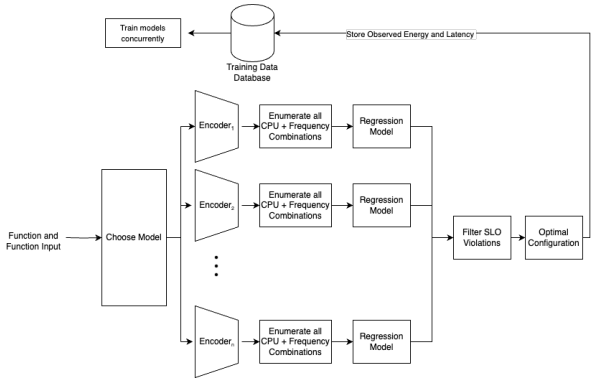


Figure 3: Gaussian Process Regression Model for Carbon-Aware Serverless System

6.1.1 Evaluating Model Performance on Measurement Study Data

In order to analyze the effectiveness of our model and tune hyperparameters, such as the confidence threshold and the type of kernel used, we simulate a trace of function invocations using the measurement study data and compare our model predictions to the ground truth values obtained from the study.

Every ten seconds, we "invoke" a function with a certain function input and send this information to the model, which vectorizes the function input features and finds the optimal resource configuration that meets SLOs. Our model is trained simultaneously in a background thread every 30 seconds.

For Float Matrix Multiplication, Linpack, and Encrypt, our model is able to eventually converge to a minimum energy value after training on a certain number of invocations and learning from the patterns that it sees overtime. We analyze the energy expended over time for each function invocation (for a few function inputs) and see a clear convergence for these three functions. When comparing the energy consumption of the predicted configuration to the energy of the known minimum configuration (from our ground truth values), we also see the difference between these two values eventually converge to a value close to 0. (meaning our prediction of the optimal configuration is very close to the actual minimum configuration).

We also assess the model predictions themselves by calculating the difference in the energy that is predicted for a certain configuration vs the actual energy expended for that configuration. For these three functions, we see the model can eventually predict accurate energy values for a function given the function input and configuration of resources.

However, for image processing and video processing, our results are far more unstable and don't show clear signs of convergence in any of these aspects (energy over time, energy of predicted configuration vs minimum configuration, energy predicted by model vs ground truth energy). As we saw in the measurement study, the correlation between energy, function input, and vCPU count and frequency was more unpredictable/unclear compared to the previously discussed functions. Therefore, the model proved to have a more difficult time finding optimal resource configurations and had lower confidence in its predictions, leading to a higher amount of random configurations being chosen.

More details on this analysis can be found in Appendix A.4.

6.2 Adapting OpenWhisk

We adapted OpenWhisk to incorporate Gaussian Process Regression to drive a carbon-aware DFS and vCPU allocation model. Figure 4 shows the system overview.

The Gaussian Process Regression model is contained in a shim layer, called the EnergyModel. Whenever a user invokes a function, the EnergyModel determines the optimal vCPU and frequency allocation. The EnergyModel is trained online periodically at 30 second intervals.

We created a custom scheduler, called EnergyScheduler, that replaces OpenWhisk's current scheduler. There are multiple invoker nodes, each running at a different frequency, as shown in Figure 4. The EnergyScheduler uses the output of EnergyModel to determine which invoker to send the function invocation to. We base the EnergySched-

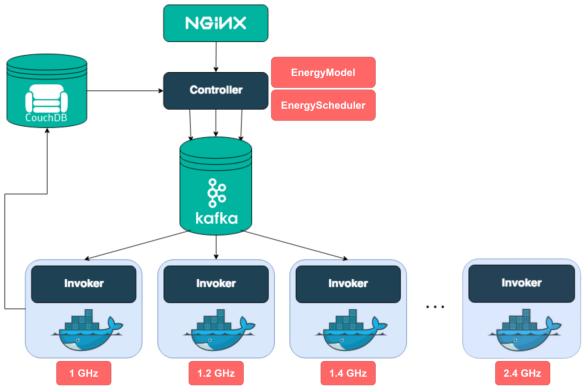


Figure 4: Adapted OpenWhisk System Overview

uler's algorithm off of Shabri's scheduler [8]. EnergyScheduler is designed to mitigate cold starts by:

1. Prioritizing routing the invocation to warm containers: The scheduler will receive the invocation and the predicted vCPU and frequency allocation from the EnergyModel. The scheduler will then iterate through the invokers running at at least the predicted frequency and attempt to find a warm container that is closest to the predicted vCPU allocation. If a warm container is found, the invocation is sent to that container. If no warm container is found, the invocation is send to the invoker running at the predicted frequency.

2. Preemptively spinning up containers: In the case that the invocation is sent to an invoker running at higher than the predicted frequency or with more vCPUs than predicted, the scheduler will preemptively spin up a container on the invoker running at the predicted frequency with the predicted vCPU allocation. This is done to mitigate future cold starts and underutilization of resources. The underlying assumption is that function invocations exhibit temporal locality, meaning that in a short time frame, the same function is likely to be invoked again. Thus, preemptively spinning up a container will reduce the likelihood of cold starts for future invocations and underutilization of resources.

7 Evaluation

7.1 Methodology

Workloads. We evaluate our system using functions shown in Table 3. These are the same functions used in the measurement study. These functions vary in their parallelizability and resource requirements, making them suitable for evaluating the effectiveness of our system. We utilize the same Azure trace used in Shabri [8] to simulate function invocations, which is a scaled down sample of the Azure trace. We utilize a requests per second (RPS) of 1 for the evaluation.

Metrics. For a high-level, comprehensive evaluation of our system compared to baseline models, we aggregate all of the energy readings for each function invocation for a specific function type (as the overall energy expended for that function) and compare these energy totals across each model. We also count the total number of SLO violations for each function type and compare this across all models as well.

Baselines. We compare our system to two baselines Parrotfish and Aquatope. Parrotfish [7] utilizes parametric regression to predict resource configuration, while Aquatope [9] uses Bayesian Optimization to predict resource configuration. Neither of the aforementioned baselines consider optimizing for energy efficiency. Furthermore, the original OpenWhisk's scheduler was used when running Parrotfish and Aquatope. Parrotfish only optimizes memory allocation; thus, we allocate CPU proportional to memory.

Our System. Since our system, CASS, only optimizes CPU and frequency, we run our system with two memory configurations: 1. using Parrotfish's memory predictions (CASS_Parrotfish), 2. using Aquatope's memory predictions (CASS_Aquatope). We compare our system to the aforementioned baselines in terms of energy consumption and SLO violations.

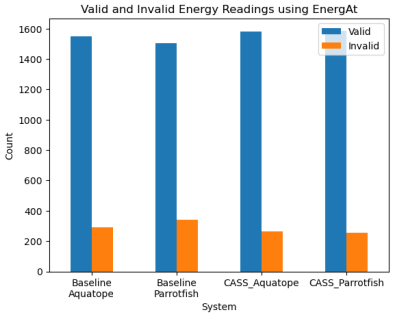
7.2 Assessing the Efficacy of EnergyLiteDaemon for Collecting Energy Readings

Before running the full evaluation of our regression model and scheduler, we ran a preliminary evaluation of our energy measurement tool, EnergyLiteDaemon, on the Azure trace, in terms of how effectively and accurately it can collect energy readings for each function invocation. We compared its results to the original implementation of EnergAt.

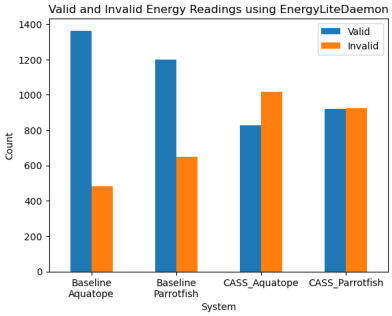
As shown in Figure 5, EnergyLiteDaemon has far more invalid energy readings compared to EnergAt. Invalid is defined as a negative reading, which signifies that no energy measurements were collected during the lifecycle of that function invocation, or the energy measured was 0. The reason EnergyLiteDaemon has far more invalid readings is that unlike EnergAt, which has a dedicated process energy measurement process per active container, EnergyLiteDaemon only consists of a single process that loops through all the active containers and iteratively samples for energy for each container for a specified period of time. When there are hundreds of containers that are running simultaneously, there is a high likelihood that the EnergyLiteDaemon is not able to collect energy samples for functions with shorter durations before they complete.

Function	Input Type	# Runs	Sizes	Size Range
floatmatmult	square matrix	383	25	1000 - 8000
linpack	square matrix	381	22	250 - 10000
encrypt	string	348	36	500 - 50000
imageprocess	image	374	64	916K - 30M
videoprocess	video	358	17	284K - 11M

Table 3: Summary of serverless functions used in experiment



(a) EnergAt



(b) EnergyLiteDaemon

Figure 5: Comparing the number of invalid vs valid energy readings for EnergyDaemonLite and EnergAt

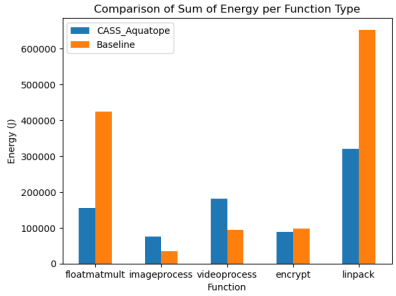
Though EnergAt is more computationally expensive and could lead to resource contention, for collecting accurate energy values, it was the most practical tool to incorporate into our system. To alleviate the issues associated with EnergyLiteDaemon, possible future steps that could be taken to improve the tool include reducing the energy sampling period, refining the interpolation technique used, and settling on a middle ground of having a separate process for each set of 20 or so containers, rather than having only 1 process for all containers or having 1 process per container. Furthermore, the current implementation of EnergyLiteDaemon is in Python; rewriting it in a lower-level language such as C++ could also help speed up the process of collecting energy readings.

7.3 Results using EnergAt

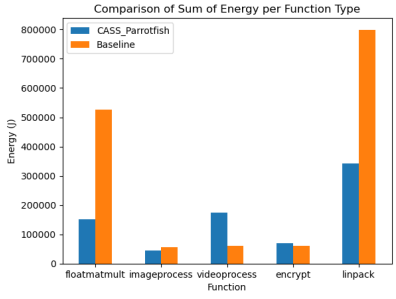
Due to EnergyLiteDaemon's inability to collect accurate energy readings with many containers, for the rest of the evaluation, we used EnergAt as our energy measurement tool and evaluated the efficacy of our end-to-end system in terms of conserving energy while meeting SLOs.

7.3.1 Energy Comparison

For a given specific function type, we add up the energy readings for each invocation and use that as the metric for the total amount of energy consumed for that function type. Figures 6a and 6b show the comparison between the Carbon Aware Serveless System (CASS_Aquatope) and Aquatope, as well as CASS_Parrotfish and Parrotfish, respectively.



(a) Overall energy expended for CASS_Aquatope vs Aquatope



(b) Overall energy expended for CASS_Parrotfish vs Parrotfish

As shown in the figures, the functions that are highly parallelizable and have a clear underlying correlation between the relevant input features (such as vCPU count) and energy consumption, such as Float Matrix Multiplication and

Linpack, see a reduction in energy consumption by almost 50% compared to both baseline models. Encrypt sees comparable energy between the CASS and the two baselines, whereas Image Processing and Video Processing see worse energy consumptions compared to both baselines.

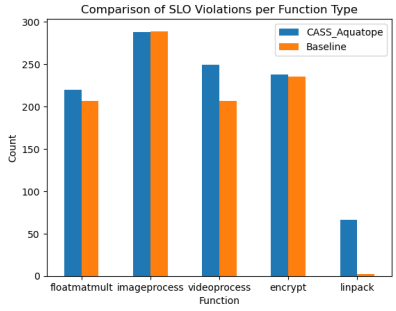
For Encrypt, as shown in our analysis of our model on the measurement study data, the model was able to converge to an optimal resource configuration (in terms of energy consumption) after several hundred invocations per function given a specific input. Because the Azure trace only contained a few hundred invocations per function type, and only a handful for a specific function input, we were not able to see the model learn to its full capacity. Given more function invocations to train on, we believe we could see a reduction in energy consumption comparable to Float Matrix Multiplication and Linpack.

For Image Processing and Video Processing, we can likely attribute this decrease in performance in terms of energy consumption to the highly complex relationships between energy consumption and the function input and resources allocated. As discussed in the measurement study section, there are no clear trends or correlations that could be seen, and for the model to predict configurations with high confidence (if at all possible), it would require far more function invocations for it to learn from.

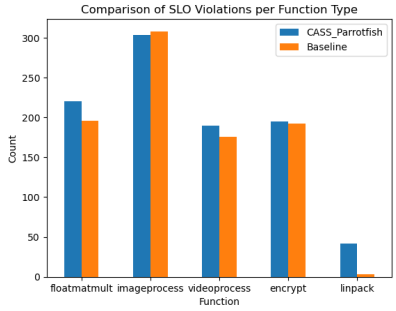
After doing further analysis, we concluded that the model provides the scheduler random configurations due to the low confidence in its energy predictions, which is most likely what leads to large fluctuations in energy consumption. In our current implementation, if the configuration with the minimum energy has low confidence, we immediately choose a random configuration. For future improvements, we can instead check the top k configurations (where k is a hyperparameter) and choose the best configuration that meets the confidence threshold (and choose a random configuration if none of them meet the threshold). This optimization may help improve the energy expenditure of the overall system.

7.3.2 SLO Violations Comparison

For each function type, we also calculate the number of invocations that violate the specified SLO. Figures 7a and 7b show the comparison between CASS_Aquatope and Aquatope, as well as CASS_Parrotfish and Parrotfish, respectively.



(a) Number of SLO violations for CASS_Aquatope vs Aquatope



(b) Number of SLO violations for CASS_Parrotfish vs Parrotfish

Overall, CASS has slightly more SLO violations than the baseline models for the different function types. However, overall, the number is comparable across CASS and the baseline models, except for Linpack.

We can likely attribute the increase in SLO violations to the random configurations provided to the scheduler in the case of low confidence or generally poor predictions in early stages of training. For future works, we could try to incorporate a mechanism that can optimize the performance in a different aspect, such as increased CPU and memory utilization or lower latencies (similar to the baselines), if the energy model is not able to confidently make predictions to minimize energy. This could help reduce the total number of SLO violations.

8 Discussion

9 Conclusion

References

[1] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, Mar. 2017. USENIX Association.

[2] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues. With great freedom comes great opportunity: Rethinking resource allo-

cation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pages 381–397, New York, NY, USA, 2023. Association for Computing Machinery.

[3] W. Buchanan, J. Foxon, D. Cooke, S. Iyer, E. Graham, B. DeRusha, C. Binder, K. Chiu, L. Corso, H. Richardson, V. Knight, A. Hussain, A. Allison, and N. Mathews. Carbon-aware computing: Meaasuring and reducing the carbon intensity associated with software in execution. White Paper, 2023.

[4] M. Chadha, T. Subramanian, E. Arima, M. Gerndt, M. Schulz, and O. Abboud. Greencourier: Carbon-aware scheduling for serverless functions. In *WoSC '23: Proceedings of the 9th International Workshop on Serverless Computing*, pages 18–23. ACM Press, Dec. 2023.

[5] U. Gupta, M. Elgamal, 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, pages 784–799, New York, NY, USA, 2022. Association for Computing Machinery.

[6] H. Hè, M. Friedman, and T. Rekatsinas. Energat: Fine-grained energy attribution for multi-tenancy. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23. ACM, July 2023.

[7] A. Moghimi, J. Hattori, A. Li, M. Ben Chikha, and M. Shahrad. Parrotfish: Parametric regression for optimizing serverless functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, pages 177–192, New York, NY, USA, 2023. Association for Computing Machinery.

[8] P. Sinha, K. Kaffes, and N. J. Yadwadkar. Shabari: Delayed decision-making for faster and efficient serverless functions, 2024.

[9] Z. Zhou, Y. Zhang, and C. Delimitrou. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, pages 1–14, New York, NY, USA, 2022. Association for Computing Machinery.

A Appendix

A.1 Energy Measurement

A toy example of the energy measurement of a container using EnergAt and EnergyLiteDaemon is shown in Figure 8. Figure 8a shows the energy readings of EnergAt. Figure 8b shows the energy readings of EnergyLiteDaemon. EnergyLiteDaemon doesn't capture all energy readings for the container due to the round-robin sampling approach. Figure 8c shows EnergyLiteDaemon's power calculation of each energy sample. Figure 8d shows the energy interpolation of EnergyLiteDaemon using trapezoidal integration.

A.2 Energy Variation with vCPU and Frequency Allocation

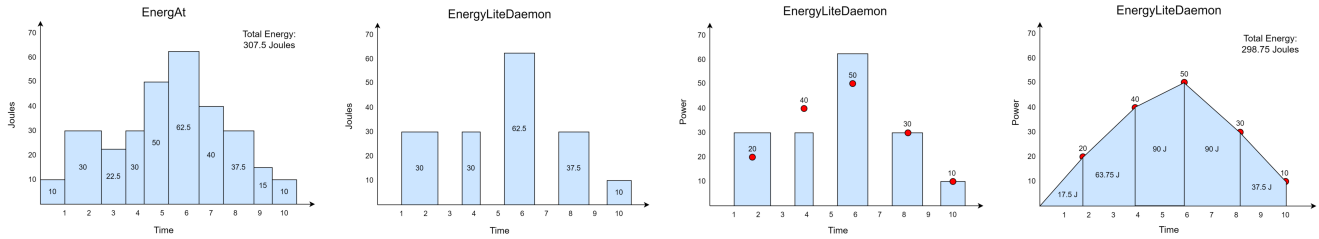
The energy usage, duration, and CPU usage for Float Matrix Multiplication, Image Processing, and Encryption as a function of vCPU allocation are shown in Figure 9.

The energy usage and duration for Float Matrix Multiplication, Image Processing, and Encryption as a function of vCPU frequency are shown in Figure 10.

A.3 Energy Variation for Different Stages of a Serverless Function Invocation

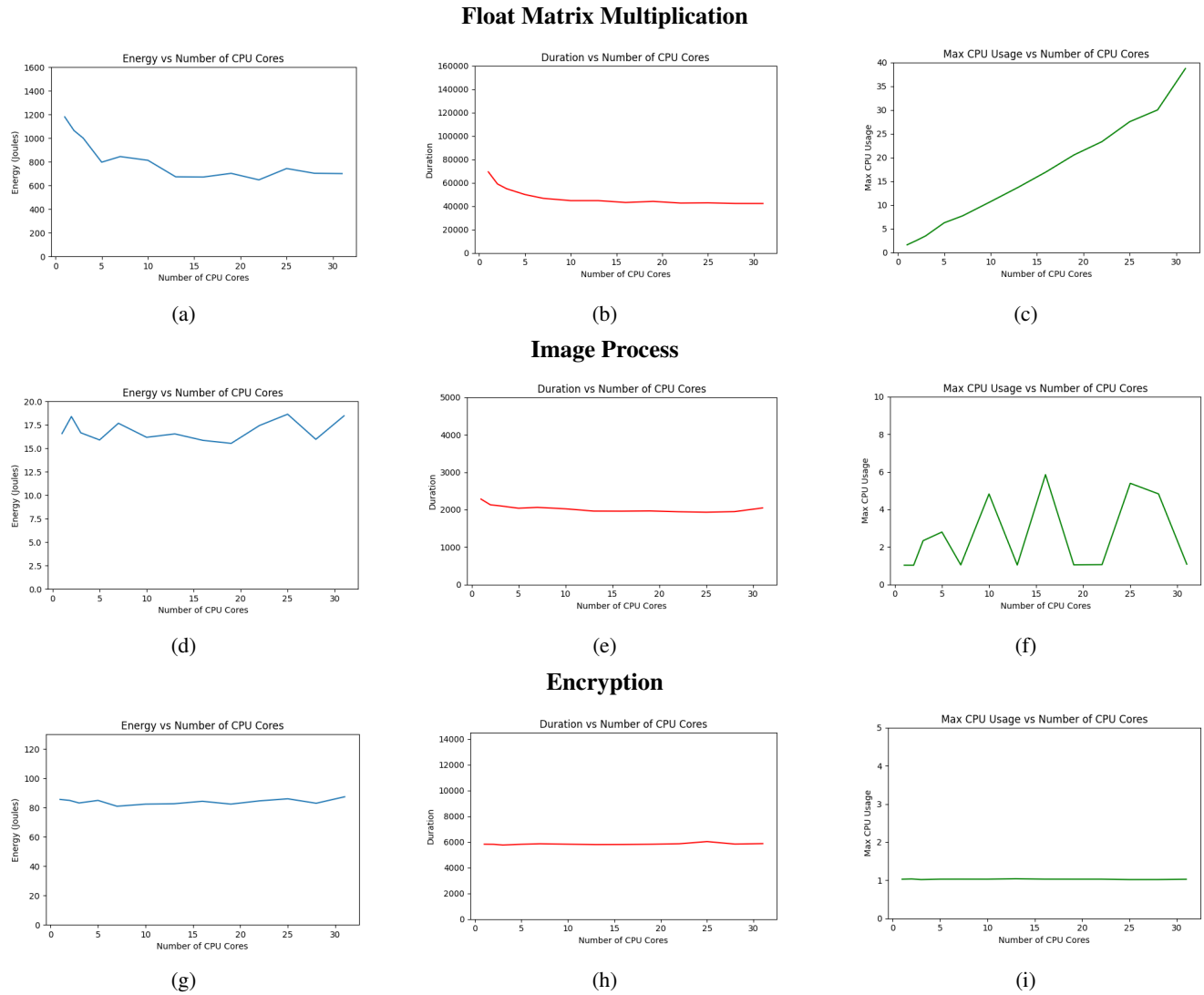
We can see here the energy consumption of the container spin-up, idle time, and container spin-down for different image sizes in Figure 11.

A.4 Evaluation of the Performance of Gaussian Process Regression Model on Measurement Study Data



(a) (b) (c) (d)

Figure 8: Toy example of energy measurement of a container using EnergAt and EnergyLiteDaemon. (a) EnergAt energy readings. (b) EnergyLiteDaemon energy readings. (c) EnergyLiteDaemon power calculation (red dots). (d) EnergyLiteDaemon energy interpolation using trapezoidal integration.



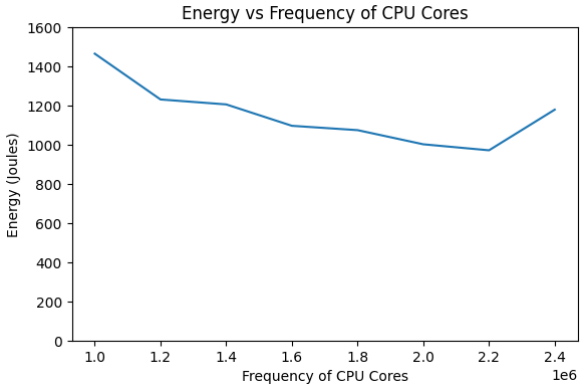
(a) (b) (c)

(d) (e) (f)

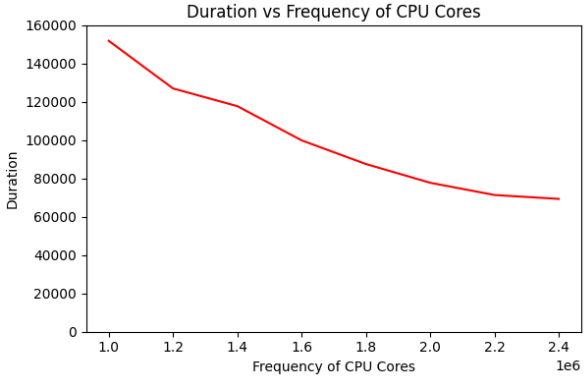
(g) (h) (i)

Figure 9: Energy Usage (a), Duration (b), and CPU Usage (c) for Float Matrix Multiplication, Image Processing, and Encryption as a function of vCPU allocation. Frequency was fixed at 2.4 GHz.

Float Matrix Multiplication

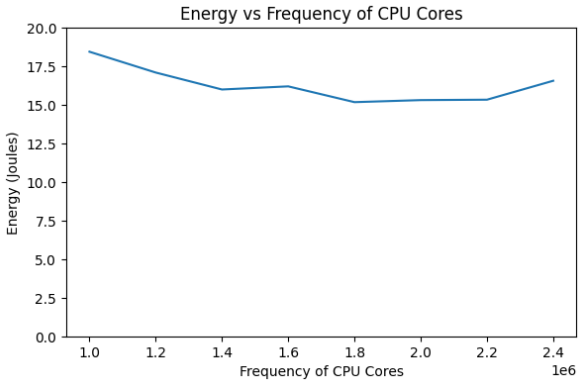


(a)

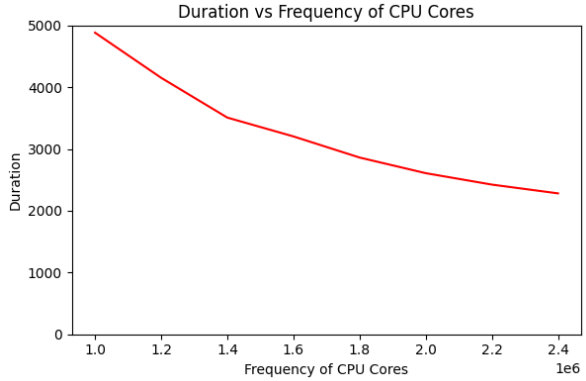


(b)

Image Process

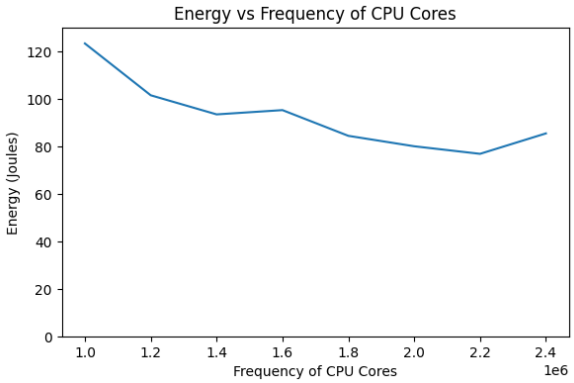


(c)

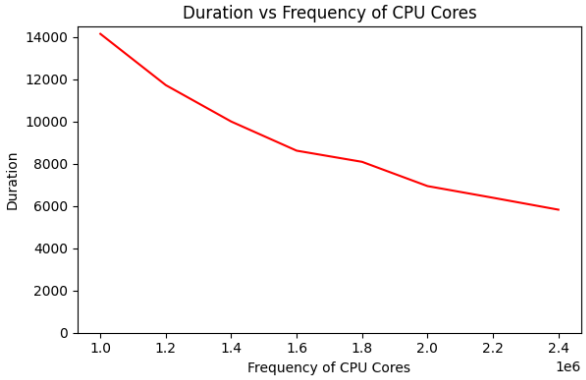


(d)

Encryption



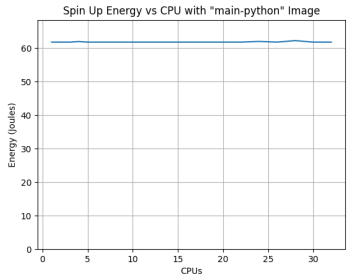
(e)



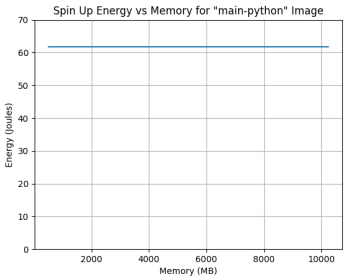
(f)

Figure 10: Energy Usage (a), Duration (b), and CPU Usage (c) for Float Matrix Multiplication, Image Processing, and Encryption as a function of vCPU frequency. vCPU allocation was fixed at 1.

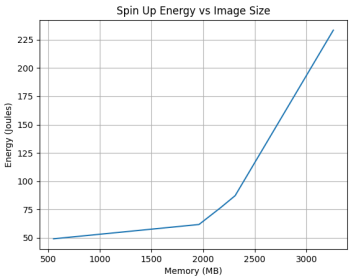
Spin Up Stage



(a)

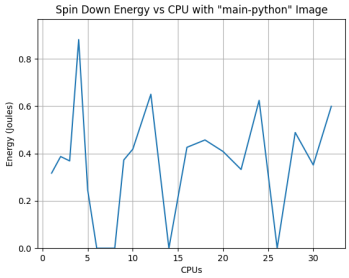


(b)

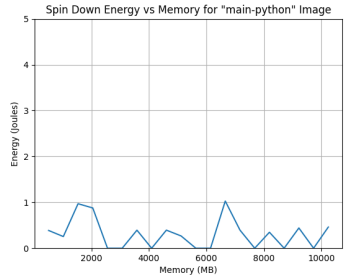


(c)

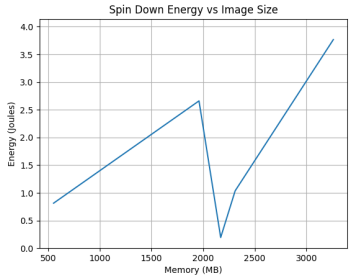
Spin Down Stage



(d)

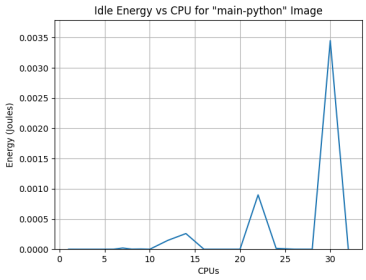


(e)

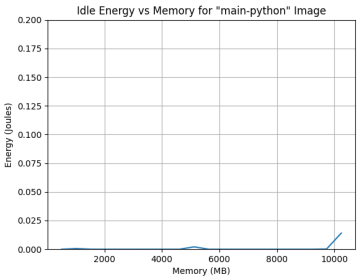


(f)

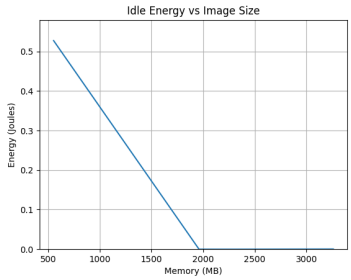
Idle Stage



(g)



(h)



(i)

Figure 11: Energy Usage vs CPU (a, d, g), Memory (b, e, h), and Image Size (c, f, i) for the Spin Up, Spin Down, and Idle stages of a serverless function invocation.