EcoScale: Giving Old Servers New Life at Hyperscale

Jaylen Wang*[1], Zhiyang Pan*[1], Udit Gupta[2], Akshitha Sriraman[1]
[1]Carnegie Mellon University, [2]Cornell University

I. Motivation

To reduce emissions at hyperscale, previous work has proposed reusing older servers [1]–[3], instead of continuing to upgrade servers every 3-6 years [4]. However, upgrading servers typically improves performance and energy, which drive server upgrades today [2], [5], especially to support latency-sensitive data center services. Thus, it is critical to extend server lifetimes in a way that achieves latency-sensitive services' performance and energy goals.

II. Key Insights and Contributions

Key Insights. Our key insight is to extend server lifetimes while avoiding performance loss by exploiting the trend of building services using numerous, distributed *microservices* [6]. Microservices enable optimizing service components independently [7]. We leverage this benefit to *reuse* older servers by running latency-tolerant microservices on them.

While previous scheduling systems have considered resource provisioning for microservices on homogeneous hardware [8], [9], they don't factor in the extra dimension of carbon emissions introduced by different generations of servers. Thus, to leverage this insight, we address two challenges. First, we must determine which microservices to run on older servers and under which operating conditions (e.g., at what loads). Second, existing systems do not consider the emissions vs. performance implications of running microservices on older servers when faced with diverse run-time conditions, e.g., variations in load and carbon intensity, which is the amount of carbon emissions generated per unit of energy consumed [10].

Contributions. To address these challenges, we develop a methodology and associated framework, *EcoScale*, that runs latency-sensitive services on older servers to reduce emissions without performance losses. *EcoScale* uses a new carbon-aware cost function that rewards running more microservices on older servers, while ensuring the service meets its performance goals without greatly increasing operational emissions.

EcoScale assumes a service can run on a set of older, i.e., deployed past their expected lifetime, and newer servers to profile performance. To account for carbon, *EcoScale* calculates C_{new} and C_{old}, the rate of carbon emitted by older and newer servers, respectively. We then define a constant "C", where $C = C_{new}/C_{old}$, representing the number of older servers that emit carbon at the same rate as a single newer server.

Given C, our key observation is that scheduling policies can save carbon any time a microservice can be placed on $\leq C$ older servers instead of a newer server. We define the scaling policy as the number of older servers to scale out to, through multiple replicas of the same microservice. Thus, *EcoScale*'s high-level goal of reducing emissions transforms into finding a microservice scheduling and scaling policy, the "*carbon-saving plan*", that replaces newer servers with the minimum number of older servers, up to $C\times$, while maintaining performance Service Level Objectives (SLOs).

*Both authors contributed equally to this work.

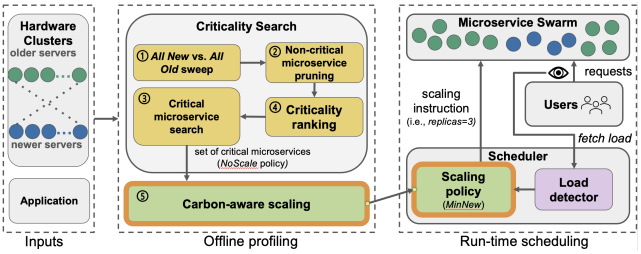

Fig. 1. *EcoScale*'s high-level system design.

III. EcoScale

We briefly discuss the two phases, offline profiling and run-time scheduling, of *EcoScale*'s design as shown in Fig. 1.

Offline profiling phase. We describe the two main steps of offline profiling. We refer to the numbers in Fig. 1 (e.g., ①).

Criticality search. EcoScale first identifies microservices that can run on one older server while maintaining performance. We call such microservices "non-critical". *EcoScale* first sweeps from low to high service loads on a cluster where all the servers are newer, i.e. "*All New*" (①). This step identifies the performance SLO that must be achieved.

EcoScale prunes the non-critical search space by eliminating microservices that can be run on older servers, by analyzing per-microservice latencies recorded from the homogeneous sweep on all older servers (②). To identify such performance-insensitive microservices, *EcoScale* notes microservices that exhibit minimal latency increases at peak load conditions. *EcoScale* forms the "criticality ranking" (③) by assessing how much placing each microservice on older servers increases service tail latency compared to the "All New" scenario.

EcoScale then performs a "greedy search" to find the minimum number of newer servers needed to meet the SLO without scaling out any microservice (④). *EcoScale* begins by placing all microservices on older servers, then iteratively moves the next most critical microservice onto newer servers based on the ranking until the SLO is met. Here, we note the resulting microservice scheduling policy as "*NoScale*".

Carbon-aware scaling. With *NoScale* identified, *EcoScale* searches for policies that place critical microservices on

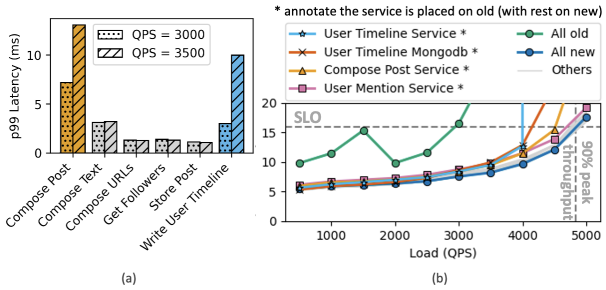

Fig. 2. (a) A subset of per-microservice tail latencies for an "*AllOld*" configuration at an input load of 3000 and 3500 QPS. Most microservices (those in gray) are not impacted by greater QPS, except for Compose Post and Write User Timeline; (b) Load vs. tail latency when placing the named microservice on an older server, with the rest of the microservices running on new. "*AllOld*" runs all microservices on older servers. Latency increases compared to "*AllNew*" indicate microservice performance sensitivity.

multiple older servers, by scaling out to up to C replicas, instead of on newer servers (⑤). To perform this search, *EcoScale* first moves the lowest ranked performance-critical microservice from a newer to an older server and runs it on two older servers, where the second server is a replica of the first. *EcoScale* then increases the number of replicas until one of two conditions occurs. First, if there are already C replicas, then more replicas would lead to increased emissions. Second, if the plan satisfies the SLO at all load conditions, then *EcoScale* has identified the minimum number of necessary replicas. In the second case, *EcoScale* then repeats the process, except it moves the two lowest ranked microservices, and so on. The resulting scheduling policy is the most carbon-efficient within the search, and we call that "*NewMin*".

Run-time phase. At run time, *EcoScale* evaluates the microservice scheduling and scaling policies in terms of meeting the SLO and reducing emissions. *EcoScale* employs a piece-wise linear model relating a policy's run-time load thresholds with scheduling operations. To obtain the run-time loads, *EcoScale* employs an event-based load detector that monitors and estimates the Queries Per Second (QPS) using a circular buffer in accordance with previous work [6]. Once load thresholds are met, *EcoScale* performs the corresponding scaling operation. Finally, *EcoScale* periodically examines the tail latency to determine if the SLO was met in the last period.

IV. Key Results

We evaluate *EcoScale*'s ability to maintain SLOs and reduce emissions by comparing the performance of DeathStarBench's [7] `Social Network` application on two generations of Intel servers, one with an older Ivy Bridge and the other with a newer Skylake processor with similar amounts of DRAM and disk. To evaluate *EcoScale*'s run-time capability, we use two production-level diurnal load traces from Google [11] and Wikipedia [12]. We overlay the workload traces with a trace of California's grid intensity during the first day of the year in 2023 [13].

To measure C_{old} and C_{new} for each microservice, we use Intel's Running Average Power Limit (RAPL) energy-reporting interface [14]. Multiplying the power by the trace's carbon intensity provides the operational emissions. As in previous work [1], we treat our older server as operating in its

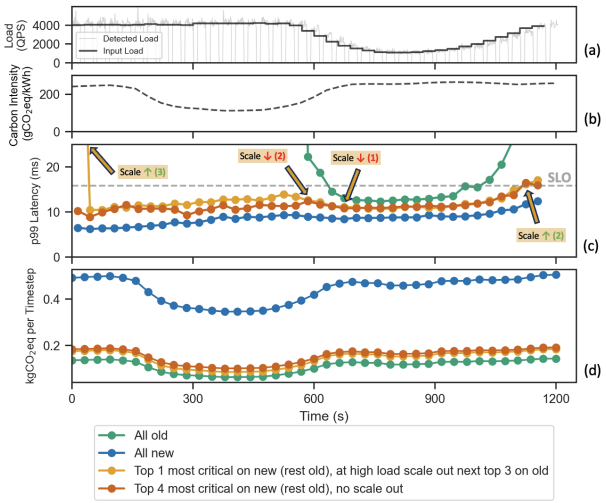

Fig. 3. Online metrics for the Google workload trace reported over time. (a) The input load (from a workload trace), and the detected load by *EcoScale*. (b) The carbon intensity trace over time matched in time with the load trace. (c) Tail latencies over time for four configurations; note how the "*AllOld*" condition only achieves the SLO at low loads, and placing a single microservice on a new server with scaling can achieve the SLO across loads. (d)Estimated carbon savings over time for the same configurations.

second life, and thus having zero embodied emissions. To calculate our newer server's embodied emissions, we use ACT's carbon modeling [3] along with the server's specifications (i.e., CPU and DRAM capacities) to estimate 432 kg CO_2e per newer server. These calculations result in $C = 3$.

We now evaluate *EcoScale* in terms of maintaining SLO and the end-to-end carbon savings it achieves.

Evaluating the offline profiling phase. We record the end-to-end $99^{th}\%$ tail latencies for `Social Network` while sweeping from low to high input loads, when one microservice is placed on an older server while the rest are placed on a newer server. Fig. 2(a) shows the results for each microservice.

When run on an older server, most microservices (e.g., "Store Post"), result in $< 5\%$ end-to-end peak saturation throughput loss. Some microservices, however, such as "User Timeline", cause a high ($\sim$17%) loss, indicating criticality.

To evaluate our pruning method, we record per-microservice $99^{th}\%$ tail latencies at a medium and high load point when all microservices run on older servers. Fig. 2(a) shows a subset of microservices with high tail latency increases. When comparing the identified microservices to the exhaustive results in Fig. 2(b), the same non-critical microservices are revealed.

Evaluating the run-time phase. We evaluate the two scheduling policies that *EcoScale* identifies, *NoScale* and *NewMin*. For each policy, we evaluate *EcoScale*'s ability to maintain tail latency under the SLO, despite run-time load fluctuations. As Fig. 3 shows, "*AllNew*", where all microservices are placed on newer servers, has the best performance meeting the SLO at all times and "*AllOld*" has the worst performance violating the SLO $\sim$67% of the time. *NoScale* at high load violates the SLO 7.7% of the time, while *MinNew* violates 2.56% of the time. This result allows for an offline selection between the two policies based on expected or predicted run-time carbon intensity and load conditions.

Carbon savings. For a server scheduling and scaling policy,

we calculate the generated emissions for each time step. Fig. 3(d) compares the emissions when running the *AllNew*, *AllOld*, *NoScale*, and *MinNew* configurations. When summing the emissions over the day, compared to *AllNew*, *NewMin* achieves a 67.3% carbon saving. We repeat the same calculations using the Wikipedia trace, omitting the full results for brevity, to find that *NewMin* achieves a 72% emissions saving.

REFERENCES

[1] J. Switzer, G. Marcano, R. Kastner, and P. Pannuto, "Junkyard Computing: Repurposing Discarded Smartphones to Minimize Carbon," in *ASPLOS*, 2023.

[2] J. Lyu, J. Wang, K. Frost, C. Zhang, C. Irvene, E. Choukse, R. Fonseca, R. Bianchini, F. Kazhamiaka, and D. S. Berger, "Myths and Misconceptions Around Reducing Carbon Embedded in Cloud Platforms," in *HotCarbon Workshop*, 2023.

[3] 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 *ISCA*, 2022.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition," *Synthesis Lectures on Computer Architecture*, 2013.

[5] J. Wang, U. Gupta, and A. Sriraman, "Peeling Back the Carbon Curtain: Carbon Optimization Challenges in Cloud Computing," in *HotCarbon Workshop*, 2023.

[6] A. Sriraman and T. F. Wenisch, "μTune: Auto-Tuned Threading for OLDI Microservices," in *OSDI*, 2018.

[7] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *ASPLOS*, 2019.

[8] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *ASPLOS*, 2021.

[9] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "{FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices," in *OSDI*, 2020.

[10] U. Gupta, Y. G. Kim, S. Lee, J. Tse, H.-H. S. Lee, G.-Y. Wei, D. Brooks, and C.-J. Wu, "Chasing Carbon: The Elusive Environmental Footprint of Computing," in *ISCA*, 2021.

[11] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *ISCA*, 2014.

[12] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, 2009.

[13] "Electricity Maps." [Online]. Available: https://www.electricitymaps.com/

[14] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: memory power estimation and capping," in *ISLPED*, 2010.