Righteous: Automatic Right-sizing for Complex Edge Deployments

Aniruddha Rakshit[1], Salil Reddy[2], Rajiv Ramnath[2], Anish Arora[2], Jayson Boubin[1,*]
School of Computing, Binghamton University[1]
Department of Computer Science and Engineering, Ohio State University[2]

***Abstract*—Edge deployments perform complex deep learning inference and analysis in the wild in highly resource constrained environment. They are positioned everywhere from our largest cities to the bottom of our oceans, and often necessitate significant financial resources and labor to create and deploy. These properties make correctness of edge deployments simultaneously extremely important and difficult to verify a priori. In the past decade, a series of IoT and cloud testbeds have emerged to facilitate this testing. They provide users with access to resources and, less often, sensors that can be used to emulate workloads before deployment. While developers can use these resources to verify the correctness of their configurations, often users would like to "right-size" their deployments – that is, to find a minimal resource configuration that guarantees correctness – to decrease cost and prevent over-provisioning. The current suite of cloud and IoT testbeds does not provide this capability. We present Righteous, an automatic deployment right-sizing tool for edge deployments. Righteous treats configuration as a hyperparameter optimization problem, testing hyperparameter combinations to find a near-optimal configuration as quickly as possible. Righteous uses a new optimization algorithm, informed Pareto Simulated Annealing (iPSA) to find near-optimal configurations faster than other leading approaches. We use Righteous in conjunction with the PROWESS testbed to optimize a drone swarm deployment workload. Our results demonstrate that Righteous configurations use up to 3.5X less resources than those identified by leading hyperparameter tuning and resource allocation techniques, and does so up to 76.3X faster.**

***Index Terms*—resource allocation, edge computing, optimization, UAV**

I. Introduction

Edge computing constitutes new and expressive level of the deployment hierarchy for modern workloads [44]. Fundamentally, edge computing connects sensors on the ground with increased intelligence, enhanced security, and rich configurability. New and interesting sensors are driving innovation, new hardware platforms and accelerators are deploying AI in the wild, and privacy concerns are growing.

Diverse applications can be run in edge contexts. Range in size from minuscule far-edge sensors [28] to massive content delivery networks and are deployed in buildings and cities [6], crop fields [13], forests [17], and even under the ocean [50]. These devices connect to, or are themselves sensors that collect useful and interesting data, process that data via artificial intelligence, and often transmit it back to stakeholders for analysis. AI inference from edge deployments are used to monitor noise complaints in large cities, track crop health over the growing season, track and monitor wildfires, and more.

While edge deployments are useful and solve real problems, their diversity and complexity makes correctness difficult to verify in development [5], [14], [35]. It can be unclear whether software, hardware, models, sensors, and networks will behave well in concert without complete implementation and often formal verification. Building and deploying these systems can be expensive, with large deployments costing millions of dollars. Deployments may also require custom devices, networking hardware [43], or sensors that require additional development and expense [28], [43]. Users generally work to verify the feasibility and correctness of their deployments before they commit to costly hardware purchases, or while they are developing aspects of the deployment through either the use of simulation, custom testbeds, or community testbeds.

For this reason, significant investments have been made in recent years in edge, cloud, and IoT testbeds [8], [9], [12], [18], [19], [23], [32], [37], [38], [49], [49]. Testbeds and testbed platforms like Chameleon, the Platforms for Advanced Wireless Research (PAWR), GENI, PROWESS, Colosseum, Kansei, Mirage and many others are allowing users to build and test platforms with diverse sensors, compute platforms, and network technologies while requiring minimal upfront investment. Many users opt to use simulation environments or custom testbeds to similar effect. While custom testbeds require additional monetary investment and simulation platforms may not generalize to real-world conditions, these techniques are still valuable and almost necessary for verifying large IoT deployments. Users rely on these techniques to test changes in potential deployments, select hardware, software, and models, and very correctness.

Verification of correctness is however, only one aspect of edge deployment development. As researchers test their deployments in simulation or using testbeds, they manually configure hardware, resource allocations, networks parameters, AI models, and more. One principle benefit of testbed and simulation platforms is the facilitation of this tuning process. Users can swap in and out pieces of their deployment and observe the effects. Ultimately, researchers seek a configuration that satisfies all of their goals (e.g., meets latency or accuracy requirements for end users) while also meeting power, form-factor, and cost constraints. We refer to this process as " Right-sizing ".

For simple deployments, right-sizing can constitute changing only a few variables. For large deployments, the space of candidate configurations can explode exponentially. Deployments may encompass multiple device types with configuration options including network substrate, CPU and memory allocation, power and battery constraints, software and AI models. Additional optimizations may also be tested including early exits for AI models, device duty-cycling, or multi-tenancy. Complete exploration of these options can quickly become impossible, and even manual tuning can necessitate the exploration of hundreds or thousands of configurations to meet performance goals.

Parameter tuning in high-dimensional spaces is a well-researched topic. Optimization algorithms in operations research, resource allocation, computer scheduling, and AI frequently explore high-dimensional spaces. Techniques like Bayesian optimization [7], economic resource allocation via the Boston mechanism [27], and multi-resource clustering algorithms [26] are used to simplify high-dimensional spaces and find strong candidate solutions in different domains. We show that these technique are not properly suited to edge deployments.

We present Righteous, an edge deployment right-sizing tool for simulators and testbeds. Righteous models edge deployments as sets of containers, whose hardware, software, models, and networks can be easily manipulated. Righteous treats these resources as hyperparameters to be tuned by an optimization algorithm. As shown in Figure 1, Righteous takes these containers, along with user-specified performance goals, and tests them repeatedly to determine a configuration that meets goals with the lowest possible resource footprint.

Righteous uses a new optimization algorithm called Informed Pareto Simulated Annealing (iPSA) to quickly identify feasible and near-optimal configurations for these containers based on user-provided goals. We test each hyperparameter configuration by automatically shaping and deploying Kubernetes pods that match our configuration. In this paper, we show that Righteous and iPSA outperform other state of the art optimization techniques from computer scheduling and AI literature. Righteous finds up to 3.5X better configurations than leading hyperparameter tuning and resource allocation techniques, and does so up to 76X faster. Righteous is never out-performed in our testing by optimization techniques, and finds allocations only 10% more provisioned than optimal configurations identified by exhaustive grid search, a 1400X slower technique. We show that Righteous experiments can be easily parallelized unlike other optimization techniques, leading to saturation of testbed allocations and decreased runtimes for each evaluation at no cost.

In section 2 of this paper, we describe the motivation for Righteous and provide background on edge deployment optimization. Sections 3 discusses the iPSA algorithm and the design of the Righteous testbed tool. Section 4 outlines a case study used to evaluate the performance of Righteous compared to other optimization algorithms. Section 5 presents results of our experiments. Sections 6 and 7 provide limitations, future work, and conclusions.

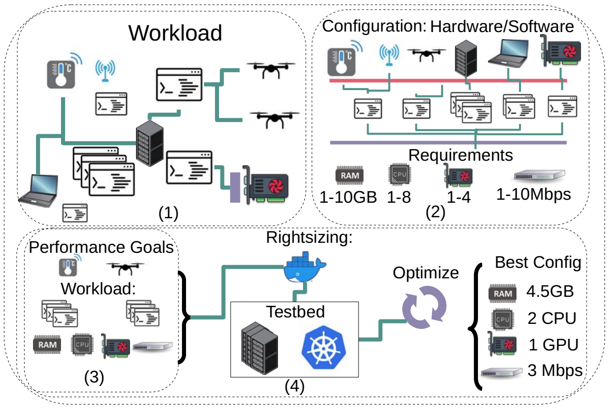


Figure 1. Righteous Optimization Process: 1) Provide a workload as set of containers, 2) Define resource configuration space, 3) Supply performance goals, 4) Use iPSA and Kubernetes to evaluate candidate configurations.

II. Background

The concept of right-sizing is common across engineering [41], computing [10], economics [22], healthcare [1], and even in the natural world [47]. This process of finding just the right configuration to fit a problem has been rigorously defined mathematically through optimization algorithms. In this section, we will detail some background on optimization techniques and applications relevant to right-sizing for edge deployments.

A. Resource Allocation

A foundational concept in optimization is resource allocation. Resource allocation problems pose the following question: given some set of resources S and requests R, how can we best assign our resources $S_i \leftarrow R_j$ to maximize the performance of our system? Resource allocation problems are common in economics, healthcare, and computing where resources (e.g., money, doctors, CPUs) are scarce and must somehow be assigned, usually for some globally maximum benefit. Assignments are based on a policy P. P may assign resources to hospitals or patients based on need or perceived benefit [33], or may assign CPUs to interactive processes over daemons, or may assign high-achieving students to magnet high schools [22]. The goal of a resource allocation is to select an algorithm $F(R, S)$ that maximizes some quantities that reflect policy enforcement.

Policies in resource allocation focus on high-level concepts like fairness, priority, and envy minimization. In human-centric systems like healthcare, resources are often indivisible. Algorithms like first-choice maximality [27] assign resources to agents in perpetuity based on a policy that does not adapt after assignment. In computing, resources are often divisible, requests are ephemeral, and contention is high. For example, in single-CPU scheduling, many processes will contend for access to one CPU. Over a processes lifecycle, it will periodically require CPU access, perform a calculation, and then wait for some I/O operation. The length of CPU and I/O bursts are not known prior to their execution, making a-priori resource allocation unclear. Therefore, scheduling algorithms often rely on a posteriori information [39].

As computer systems grow in complexity, this lack of clarity persists. Complex software running across institutions [12], large IoT and Edge deployments [6], [38], and planet-scale data centers [15] must be profiled in realistic scenarios to truly understand resource needs. This profiling process can be simplified in scenarios where resources can be over-provisioned. If a reasonable upper bound for resource consumption can be determined, the system can be deployed. Edge applications often violate this assumption. These applications have constraints on form factor, power consumption, and network substrates that may be impossible to over-provision without a posteriori knowledge. They must also satisfy models whose complexity often scales with accuracy. The purpose of Righteous is to use accurate profiling and incisive optimization to quickly determine the proper configuration for an IoT deployment. In the rest of this section, we will provide background on 1) methods for testing Edge application performance and 2) relevant optimization algorithms for resource allocation.

B. Computing Testbeds

Testbeds are often used to approximate real-world performance for compute workloads before deployment. Testbeds allow users to deploy workloads in representative conditions, generally using shared resources. Testbed offerings have improved in recent years [8], [12], [18], [32], [37], [38]. Testbeds for edge and IoT applications specifically [12], [31] allow users to either test applications on bare-metal hardware, or provide light-weight virtualization or containerization that can constrain applications to edge footprints. Publicly available testbeds like Chameleon [32] and GENI [8] provide users access to share resources that can be used to emulate their workloads. PROWESS [12], an open testbed software platform, allows users to instantiate their own testbeds for sensor and network dependent applications.

Users may assess the correctness of their applications on these testbeds, but to right-size an application, they must explore myriad hardware and software configurations. In the next section, we rigorously define right-sizing and present candidate solutions from recent literature.

C. Optimization for Right-Sizing

As shown in Figure 2, we define right-sizing as a multi-objective optimization problem. Applications $A = \{s_1, s_2, ..s_n\}$ are sets of software artifacts (s_n) that request hardware resources $H = \{h_1, h_2, ...h_m\}$. Hardware resources are linked to devices $d_i \in D, d_i \leftarrow \{h_1, h_3, h_7\}$, and each software artifact must be placed on one such device, thus sharing its resources $d_i \leftarrow \{s_1, s_4\}$. Software components can be linked by network resources, therefore behaving as a bi-directional graph. Applications also contain sets of performance goals $G = \{g_1, g_2, ...g_n\}$. The goal of our optimization technique is to determine the minimal set of hardware that can support our application and meet its goals $argmin(H) \ni g_i \leq p_i \forall i$

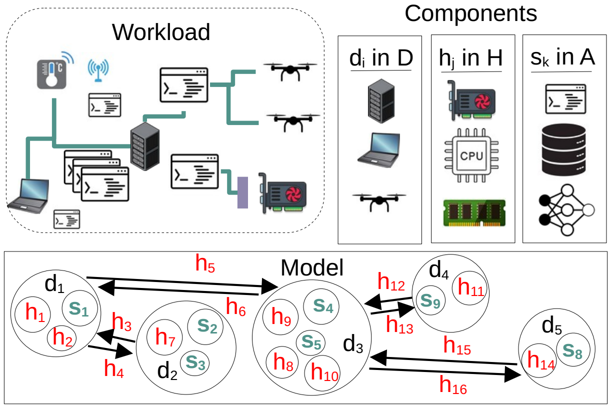


Figure 2. Our theoretical model for right-sizing decomposes workloads into devices which house hardware and software. Right-sizing finds a minimal set of hardware which allows software to meet performance goals.

Before testing, the resource utilization of each component is unclear. We define the following evaluation function $R(A, H) = P$ to determine performance of a candidate hardware $P = p_1, p_2, ...p_n, |P| = |G|$ where p_i represents performance with respect to g_i. The goal of right-sizing is to find H_{min}, the minimal set of hardware such that $R(A, H_{min}) = P_{min}$ and $p_{min,1} > g_i \forall i$. This problem is NP-hard. Like many resource allocation problems, this problem corresponds to a multiple knapsack problem [40], where every $d_i \in D$ corresponds to a knapsack of capacities with respect to its hardware resources. Right-sizing is additionally complicated by the lack of a priori knowledge of the weight (i.e., performance) of the applications, and the continuous space across which compute resources can be allocated.

Due to the size of the continuous search space across $|H|$ hardware allocations, exhaustive search is impractical. Grid Search [36], where each continuous h_i is discretized into a set of candidate allocations and exhaustively searched, will yield some meaningful H_{min} whose quality is dependent on the granularity of the search space. Unfortunately, Grid Search scales exponentially with $|H|$. In large edge systems, each new hardware device can increase $|H|$ by multiple factors if accounting for is various resources (RAM, CPU, ingress and egress network capacity, etc). Therefore, we require an optimization algorithm that can nimbly navigate the search space.

$$R(A, H) = P \tag{1}$$

$$H_{min} = argmin(H_i) \ni p_i < g_i \forall i \tag{2}$$

$$\min_{h \in H} f(R, h) = H_{min} \tag{3}$$

We define our edge right-sizing problem in equations 1-3. The goal of right-sizing is to find H_{min} via an optimization algorithm $f(R, h)$. Our challenge is to identify or construct this function. Many optimization algorithms exist for quickly navigating large spaces. However, via the no free lunch theorem [2], all optimization algorithms will perform equally well across a large set of problems. We must, therefore, identify an optimization algorithm suitable for solving this specific right-sizing problem. Two candidate classes of algorithms are 1) Bayesian Optimization, and 2) cluster resource allocation.

D. Prior Work

Bayesian optimization (BayesOpt) is a widely used optimization technique based on Bayesian statistics. BayesOpt seeks to the inputs to a function that will result in globally optimal outputs [25]. BayesOpt uses a Gaussian process to iteratively generate a posterior probability distribution across the search space. Using an acquisition function across the prior (e.g., expected improvement), BayesOpt selects a sample point which improves its understanding of the search space per some criteria.

BayesOpt has emerged as a best-in-class optimization technique for expensive black box functions [45]. BayesOpt has found extensive use in machine learning [7], materials science [48], physics [16], and other domains where search spaces are large and evaluations are expensive. Computational right-sizing has also been performed using BayesOpt. CherryPick [3] uses BayesOpt to search a large space of potential cloud deployment configurations. RAMBO similarly uses BayesOpt for performance tuning of microservices in cloud environments [34]. BOAT similarly uses structured BayesOpt to automatically right-size complex systems with a focus on machine learning [21]. StreamBed predicts suitable deployment configuration using Bayesian Optimization for stream processing [42]

BayesOpt is far from the only technique used to optimize system-level hyperparameters. Decades of research on resource allocation across operating systems, distributed systems, and data-centers. Here, we mention some recent and relevant works focused on compute placement and optimization. Cilantro is an online learning system that adapts to changing job conditions, estimating resource-to-performance mappings and handling uncertainty in learned models by working within confidence bounds [10]. Chameleon is a system for tuning video analytics applications to fit the form factor of available compute resources [29]. Chameleon uses a version of greedy hill-climbing to iteratively tune performance knobs. Ekya, another video analytics optimizer, uses a thief schedule to iteratively re-distribute resources across groups of video streams [11].

Each of these techniques is performant in its domain, but may not apply well to edge deployment right-sizing. In Section 3, we describe iPSA, an optimization algorithm for IoT right-sizing. In section 5, we will demonstrate its improved performance with respect to BayesOpt and Ekya specifically.

E. Toy Example: Optimization-based rightisizing at present

To illustrate these techniques, consider a resource-limited edge node: the Nvidia Jetson Orin Nano. The Jetson Nano is used often for accelerating robotics and AI workloads, with its Orin class containing a multi-core ARM CPU, memory, a 1GB network link, and embedded ampere GPU. The capacity and quality of the CPU, memory, and GPU are variable, with 9 potential offerings of varying computational power, where prices and power consumption increase as computational power increases. Given an edge application (e.g a computer vision workload), which of these hardware offerings, if any, is best to select? Given complete knowledge, a researchers may reasonably select the least expensive device that meets all computational constraints.

To determine the best platform for a workload, a researcher could buy all existing platforms and perform tests, but this is also not cost effective. It would be more cost and time effective to use testbed resources to approximate the hardware device and proceed accordingly. Using grid search with 10 discrete allocations for each of the Jetson Orin Nano's four h_i, this would constitute 10,000 sample points, potentially prohibitive for workloads that run for minutes before returning results.

Techniques like Ekya and Hyperopt are more incisive. Hyperopt samples based on a Gaussian prior and expected improvement. This sampling process is could use the same grid as Grid Search, or a continuous space, and would intelligently sample based on the prior. While effective, Hyperopt assumes little about the underlying shape of the function, and therefore spends precious evaluations exploring the function unnecessarily. Ekya takes a similar approach, learning the function over a series of iterations by starting with a candidate allocation and adding, removing, or trading resources from S_i to S_j to identify a maximal allocation. Both technique take significantly fewer evaluations on average to minimize an unknown function that grid search, but both treat the function as a black box. Ultimately, we would like an algorithm that explores this search space in a targeted way, relying on fundamental aspects of edge workload performance to avoid poor candidate solutions.

III. Design

Righteous is a workload right-sizing tool for computer systems testbeds. Righteous accepts workloads as sets of containers and resource allocations. It uses a novel optimization algorithm to quickly pair down their allocations and identify a minimal set of hardware resources for correct execution. In this section, we describe the system overview for Righteous as well as our novel iPSA optimization algorithm.

A. System Overview

Figure 3 shows the high-level design of Righteous. The Righteous optimization process contains three key components: workload definition, optimization, and runtime.

Workload Definition: Users provide workloads to Righteous as sets of containers. Containers are an OS-level virtualization mechanism that rely on the host kernel to isolate resources as opposed to a hypervisor. Containers are widely used to manage dependencies, deploy micro-services, and migrate applications. Containers and heavier-weight virtual machines are also used widely in computing testbeds [8], [12], [32]. We chose to implement Righteous using containerization, but Righteous could also work with virtualization.

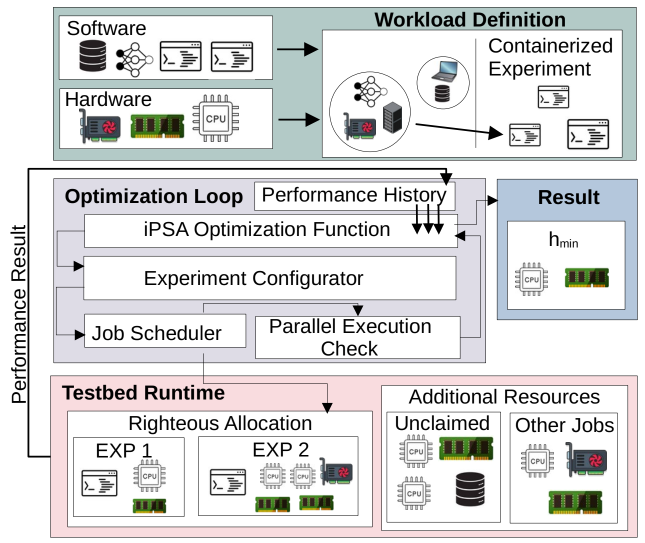


Figure 3. Righteous Overview: Users provide software as containers, hardware resource limits, and placement information. Righteous uses iPSA to determine candidate resource allocations. Candidate allocations are configured and scheduled on a testbed, with potential for parallelism. Performance results are returned to iPSA, and experiments are run until stopping conditions are met.

Righteous accepts workloads as sets of interconnected containers. Each container consists of some $\{s_i, s_j...\} \subset A$ software artifacts positioned on a device d_k. Righteous also accepts hardware limits $\{l_{min}, l_{max}\}$ for each container where $l_{min,i} \leq h_i \leq l_{max,i} \forall h \in H$. Limits can pertain to any hardware resource that modern container platforms can isolate (e.g., the number or percent of CPUs, amount of RAM, network bandwidth) or select (e.g., number or presence of GPUs, storage medium). As shown in figure 2, network links between containers can also be considered for optimization.

Optimization Loop: Per equations 2 and 3, the Righteous optimization process finds the minimum hardware set H_{min} that can execute application A while satisfying a set of performance goals G. These are provided as inputs to the optimization loop. Righteous relies on informed pareto simulated annealing (iPSA), a novel optimization algorithm described in section 3.2. iPSA, like most optimization algorithms, relies on performance history to inform outputs. Righteous continually calls iPSA to generate candidate hardware allocations h^i_{cand} based on performance history of prior history $\{h^1_{cand}, h^{i-1}_{cand}\}$. iPSA provides as its principle output a candidate hardware allocation h^i_{cand} to be tested, or a stopping condition. If iPSA does not elect to stop, h^i_{cand} will be sent to the experiment configurator for evaluation.

The Righteous experiment configurator constructs an experiment from the hardware allocation provided. This process is testbed-specific. In section 4, we will describe how the Righteous configurator creates a kubernetes deployment for testing on the PROWESS testbed. The configurator's principle output is a well-formed file fully describing our application and hardware specification $A(H)$ that is capable of execution on a computing testbed.

Righteous, as a testbed tool, must schedule jobs to 1) assure access to requested resources, and 2) avoid disrupting the experiments of other users. Righteous' scheduler integrates with testbed scheduling software to run evaluations within testbed constraints. The job scheduler can also optionally probe the testbed for unallocated resources that could potentially be used for parallel execution. If sufficient unused resources exist, Righteous' parallel execution check will request another evaluation from iPSA. If iPSA can generate another experiment without outstanding experimental data, it will be scheduled for potential parallel execution.

Testbed Runtime: Candidate allocations are configured and scheduled on the testbed, potentially with multiple experiments executing in parallel. The testbed will execute the application as a set of interconnected containers per the configuration provided by the Righteous optimization loop. Performance results are returned to iPSA and incorporated into performance history. After execution, iPSA is continually queried and experiments are executed until a stopping condition is met.

B. The iPSA Optimization Algorithm

Pareto Simulated Annealing [20] is multi-objective combinatorial optimization (MOCO) algorithm that solves $argmax(f_1(x), f_2(x), ...f_n(x))\ for\ x \in D$ where x is a set of hyperparameters resulting in a feasible solution. Pareto simulated annealing solves this problem by performing the following steps:

1) Select a starting sample of feasible hyperparameters sets S
2) For all $x \in S$ construct a neighborhood $y \leftarrow V(x)$, the set of all hyperparameter sets that can be reached by making a simple move from x (e.g., $x_1, x_2, ...x_k - T, ...x_n$).
3) If y is not dominated by x, $M \leftarrow y$ where M is the set of feasible solutions
4) Accept y with probability $P(x, y, T, \Lambda)$ where T is temperature, Λ is a set of random weights, and P returns a probability based on the Chebyshev metric [20].

The stopping conditions and annealing schedule (rate of temperature degradation) are provided by the user. Ultimately, the cheapest non-dominated feasible configuration is returned.

PSA is both high-complexity and probabilistic. Depending on the stopping condition, annealing schedule, and neighborhood selection, simulated annealing can have a considerable runtime. Due to its probabilistic nature, simulated annealing (like the metallurgical theory upon which it is based) requires a long period of temperature lowering to converge approximate a near-optimal solution [20]. PSA is generally eschewed for modern black-box MOCO problems in lieu of Bayesian optimization, as Bayesian optimization's performance is generally superior. Certain properties of PSA, however, inspire our informed PSA algorithm that outperforms even Bayesian optimization for the compute right-sizing problem. iPSA takes inspiration from PSA but removes both the probabilistic component and increases the annealing schedule to deliver an algorithm with faster convergence for a specific class of objective function.

Intuition: Architectural parameters for IoT workloads often improve performance when increased (e.g., more CPU $\leftarrow$ faster runtime) or hurt performance when decreased (e.g., slower networks, more missed deadlines) as demonstrated in equation 4. Using this intuition, we present an optimization algorithm based on PSA that relies on an approach similar to hill-climbing to eliminate the probabilistic component of PSA and forgo a slow annealing schedule. iPSA should return a near optimal configuration in less evaluations than approaches that sample an objective function with no intuition about its underlying shape.

C. iPSA Overview

iPSA avoids the probabilistic uncertainty and high complexity of PSA by utilizing the underlying share of resource utilization in edge workloads. iPSA is informed by equation 4: the principle that adding a hardware resource to an IoT deployment, all things being equal, will either increase its performance or have little effect on performance. Conversely, decreasing a resources allocation from an IoT deployment will either decrease its performance or have little effect.

$$\forall p_i \in P : \begin{cases} R(A, H_i - T) <= R(H_i) \\ R(A, H_i + t) >= R(H_i) \end{cases} \tag{4}$$

Using this principle, we implement iPSA, a polynomial time greedy approximation for PSA suited for compute resource minimization.

Algorithm 1 Informed Pareto Simulated Annealing

```
1:  procedure iPSA($S$, $T_0$, $\Delta_T$, $G$, $L$)
2:      $T \leftarrow T_0$
3:      $B \leftarrow \{l_{min,i} + l_{max,i}\}$
4:      $D \leftarrow -1$
5:      while T $> 0$ do
6:          $M \leftarrow \phi$
7:          $x \leftarrow B$
8:          $Y \leftarrow V'(x, D * T, L)$
9:          for $y \in Y$ do
10:             $P_y = Eval(x, G)$
11:             if $x$ does not dominate $y$ then
12:                 $M \leftarrow y$
13:                 if $P_y < P_b$ then
14:                     $B = y$
15:         $P_F \leftarrow Eval(argmin(M), G)$
16:         if $P_F < P_b$ then $B$ = argmin(M)
17:         if $G, P_b | \exists G_i > P_b, i$ then
18:             D = 1
19:             $T = T - \Delta_T$
20:     return $B$
```

Algorithm 1, the informed Pareto Simulated Annealing (iPSA) algorithm, approximates a pareto-optimal resource allocation scheme for a set of software artifacts S passed as containers. iPSA accepts a set of jobs S, an initial temperature T_0, and a temperature modifier Δ_T. M is a set of non-dominant hyperparameter sets that form the basis for the iPSA solution. Values in M navigate to and along the pareto curve as iPSA progresses. M originally contains a single set of hyperparameters for each resource equal to the midpoint of the limits of that resource. This starting point, in the middle of the parameter space makes no assumption about the relationship between resource limits and performance. T is temperature, the amount by which hyperparameters change across iterations. T is initialized to T_0. B is the best hyperparameter combination found by the algorithm, initialized to the null set ϕ.

iPSA iterates over temperature, decreasing T by Δ_T until T reaches 0. At each iteration, a neighborhood Y is constructed for the best previously identified hyperparameter set B. For each hyperparameter $h \in H$ in B, $V'(x, D * T, L)$ returns a complete hyperparameter set $H' \leftarrow \{h_1, h_2, ...h_i + D * T, ..., h_n\}$. One hyperparameter is modified by $D * T$ for each H'.

Each neighbor in Y (y), is evaluated on the testbed and its performance results P_y are returned. If it is not dominated by x, then it is added to M as a candidate solution. If y's score is also lower than B (or $B = \phi$), then y is therefore the best non-dominant hyperparameter set, and B is set to y. Once all y have been evaluated, set F is constructed by combining all $y_i \in Y_i$ where Y_i is not dominated by x. This hyperparameter set, Y_f is then evaluated and compared to B. Finally, if a feasible, non-dominant hyperparameter set has been found, the temperature is decreased and iPSA will decrease hyperparameters for the next iteration. If not, the temperature remains at T_0 and iPSA will increase hyperparameters in an attempt to find a feasible hyperparameter set.

This algorithm is similar to conventional Pareto Simulated Annealing, with some important changes. First, PSA's $V(x, T, G)$ finds a random neighbor $y \in Y$ surrounding x by taking making a 'simple' hyperparameter change. This function is sampled repeatedly at every temperature step to build a neighborhood against which to compare values in M. iPSA's $V'(x, T, G)$ returns an array of size $|H|$ that either increases or decreases each hyperparameter $h \in X$. Instead of randomly sampling V as a black box, iPSA deliberately navigates to the pareto bound via principle 4. Given that our pareto bound is defined by performance proximity to goal G, we know that $eval(x) > G$ reflects resource allocations greater than necessary to meet goals, and $eval(x) < G$ reflects insufficient resource allocation. For that reason, $V'(x, T, G)$ either increases or decreases resource allocation (not both) depending on goal performance.

This modification allows for two additional optimizations. First, due to our knowledge about the underlying shape of our function, we can hasten the annealing schedule. We test each hyperparameter to determine its impact on the system in isolation. All updated hyperparameters that continue to meet goals are aggregated and tested together. This allows for single steps in the annealing schedule to be 1) proportional to the number of hyperparameters, and 2) deterministic. Second, as we show in section 5, iPSA is also tolerant to a high cooling rate. In the next section 4, we describe the implementation of Righteous and our autonomous UAV case study. In section 5, we present the results of Rightous and iPSA as compared to other optimization approaches.

D. Toy Example: Optimization-based rightisizing with iPSA

Using the Nvidia Jetson Orin example from section 2, iPSA would behave as follows. iPSA would begin by sampling the midpoint hardware allocation: $H_{iPSA} = (max(h_i) + min(h_i))/2 \; \forall H$. Should $Eval(G, H_{iPSA})$ meet all goals, iPSA assumes that increasing any h_i from h_{iPSA} would result in similar performance and higher cost, and will begin to decrease all h_i. Alternatively, if H_{iPSA} does not meet all goals, iPSA assumes H_{iPSA} is insufficient and increases all h_i and re-evaluates. By leveraging these assumptions about the underlying function, we hypothesize that iPSA will outperform approaches like Ekya and Bayesian Optimization that treat the edge system as a black box.

IV. Implementation and Deployment

To test the performance of Righteous, we implemented Righteous as a tool for the PROWESS testbed. In this section, we describe our testbed environment, tool implementation, and a UAV-based edge right-sizing case study.

A. PROWESS Testbed and Righteous Tool

PROWESS [12] is a Kubernetes-based testbed focused on wireless and edge systems. PROWESS allows interested institutions to connect and slice compute resources across their networks to facilitate edge based experimentation. Unlike large cloud testbeds, PROWESS can be constructed on premises at low cost using open-source software. On-premises implementation allows users to expose and share interesting sensors and hardware at low latency, and allows PROWESS workloads to benefit from fast institutional networks between edge nodes. PROWESS testbeds use a hub and spoke model, with one or more highly provisioned core-hubs connected to lightly provisioned edge hubs. Core-hubs act as an ingress point for users, running a user-facing web application and PROWESS software for workload scheduling, storage, and resource isolation. Edge hubs are compute resources positioned at access points across the institutional network. Edge hubs can expose access to sensors or simply act as available compute.

Our PROWESS testbed consists of one core hub and three edge hubs. Our core hub is a server with two 24-core Intel Xeon 5317s, an Nvidia A5000 GPU, 256GB of RAM, 18TB of storage, and 2 25Gb network cards. Our edge hubs are HP 300 edge systems. Each system is equipped with an Intel i7 8650U, 8GB of RAM, and a 1Gb network card. All systems run Ubuntu 20.04 LTS Linux, and are deployed on a 40GB institutional network across one building.

PROWESS uses Kubernetes to schedule and distribute experiments across its hubs. Users provide PROWESS with resource reservations and containers that compose an experiment as shown in figure 2. PROWESS then schedules these experiments across its cluster based on resource availability and returns results to the core-hub for storage.

We implemented Righteous as a tool for PROWESS. Optionally, users can submit experiments that they would like to right-size. Instead of resource reservations, users provide resource limits and performance goals. Righteous then schedules experiments on PROWESS to right-size the workload using iPSA. Righteous is implemented in Python 3.11 and will be made open source upon publication.

B. Case Study: Agricultural UAV Swarm

To test Righteous, we designed an edge based case study that suits Righteous' goal to optimize unclear and complicated deployments. We opted to use an edge based case study focused on unmanned aerial vehicles (UAV). UAV are highly maneuverable sensors provisioned with onboard compute. They have been used in a myriad of deployments, piloted by software to find crop diseases [13], map wildfires [30], deliver packages [4], and more. UAVs Mobility, sensing capacity, and onboard compute provisioning, and small battery capacities make them challenging to use. Interplay between networks, onboard and off-board software, model selection, and edge provisioning can have unexpected effects on deployments that are hard to model without real-world testing [14]. We view this class of application as a challenge for Righteous.

Our case study follows a UAV swarm of size 1-5 as it maps a crop field. We draw inspiration and real-world results from a 3-UAV autonomous swarm used to scout crops in prior work [13]. This swarm used 3 DJI Mavic UAVs to partition and map an 85-acre crop field for a specific crop health condition, soybean defoliation. A centralized edge device maintains the map and dispatches UAVs to sample regions of the field. Each UAV is connected to the centralized edge device via a controller (i.e., laptop or small edge device). As UAVs fly across the field, they capture images, the images are processed in real-time via image classification [51], and the UAVs determine their next sample point based on reinforcement learning.

When designing a deployment like this, it can be unclear where software artifacts should sit (i.e., onboard the UAV, on the central device, on the controller), and what resources they require. We used Righteous to identify bottlenecks in this deployment and assist with compute provisioning.

1) Case Study Implementation: Figure 4 shows an overview of our implementation, where components of a typical UAV swarm deployment are modeled as containers which are then collectively deployed on the PROWESS testbed for each experiment. The implementation consists of the following containers:

1) **UAV** Maps to a single UAV in the swarm, and simulates a UAV's typical functions such as flying, capturing images, and transmitting the images to a central server for inference. It traverses the underlying crop fields modeled as a grid where movement across grid blocks

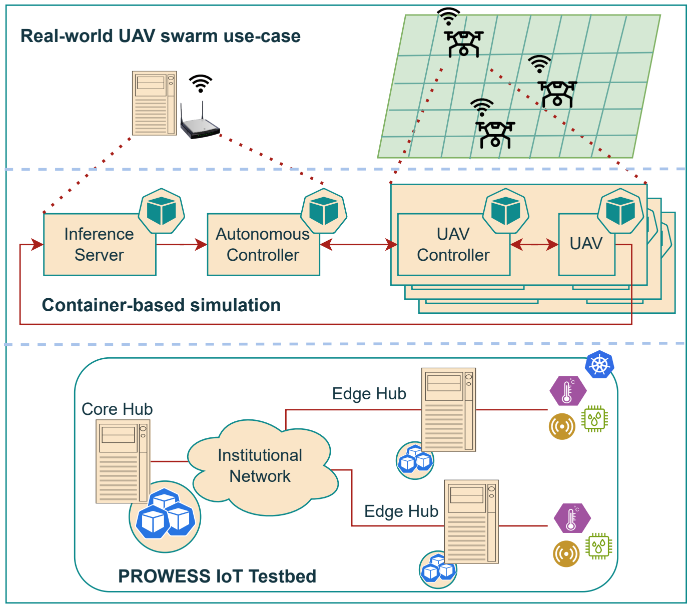
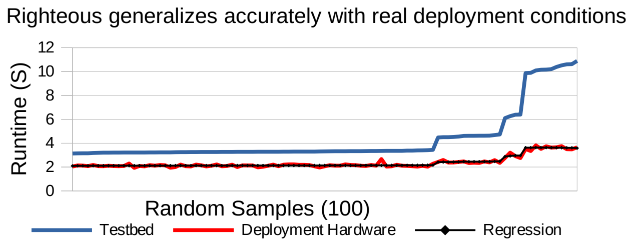


Figure 4. Implementation Overview



(A) Regression Statistics

Regression $-0.005x^3 + 0.088x^2 - 0.21x + 2.05$

	df	SS	MS	F	P
Regression	3	22.26	7.41	647.9	0
Residual	96	1.08	0.01		
Total	99	23.35			

R^2 Value
0.95
Avg. Error
3.2%

(B) Regression Performance

Figure 5. Righteous performance can be mapped easily to real-world performance.

decrements the UAV's battery, used as an indicator of its health.

2) **UAV Controller** Deployed per UAV, it interacts with the central controller to move each UAV as well as track its global and local position in the grid.
3) **Autonomous Controller** Monitors the health and positioning of all UAVs in the swarm and optimizes their movement based on output from the inference server.
4) **Inference Server** A server to which UAV swarm offloads compute-intensive tasks. In our crop scouting use-case, such a server performs inference on the incoming crop images to determine crop health [51].

An experiment sample consists of a small section of a complete deployment of above containers. A sample end when each UAV in the swarm has made one complete navigation decision including movement, classification, and path planning. The experiment run time is recorded for a specific configuration of resources. Experiments are controlled through a kubernetes manifest file in which the control parameters, the resource configurations (CPU, RAM, storage, network, etc.) of each container, are specified.

Leveraging containers and a kubernetes-based testbed for deployment allows us to scale and control our experiments at a granular level. In addition to making resource configuration simple, kubernetes CNI plug-ins (Calico, Cilium, etc.) which support fine-grained network tuning make network slicing control trivial during experimentation. Moreover, the ease and flexibility of scheduling container workloads across nodes in the cluster allows for experiments focused on optimizing node placement which are critical to identifying edge deployment bottlenecks.

2) Mapping to real-world performance: Testbed performance is only a predictor of real-world performance. When executing workloads on testbeds, it is necessary to compare to ground-truth data from execution on candidate hardware to properly gauge performance. Prior work has provided simple methods to predict performance characteristics [46] for transition from testing to deployment and for large scientific computing workloads. These methods involve sampling workloads under various conditions and building a performance model.

To test Righteous' ability to generalize performance to real-world contexts, we tested our case study implementation on candidate deployment hardware. We executed our single UAV case study on similar candidate hardware to prior work [13]: a Lenovo Thinkpad x1 Carbon Laptop with an 12-core Intel i7 CPU and 32GB of RAM. We also ran our single UAV workload on PROWESS for comparison. We ran executed 200 randomly generated configurations of hyperparameters on both the edge device and testbed. We then built a 3rd degree polynomial regression based on the first 100 random samples.

Figure 5 shows the results of our polynomial regression on our test set. Our regression achieves an R^2 value of 0.95, demonstrating strong fit to our test data. Test data on deployment hardware is significantly noisier than data executed on PROWESS. This is expected, and is due to the lack of isolation the software is provided with respect to other applications in a deployment context. Furthermore, it can be seen that PROWESS executes Righteous workloads significantly slower than deployment hardware. This is due to seccomp overhead and is used on this testbed for security reasons.

V. Results

We tested the performance of our Righteous tool and PSA algorithm on our Multi-UAV case study using the PROWESS

Table I
RIGHTEOUS HYPERPARAMETER SEARCH SPACE.

Resource	Min	Max	γ	Total
RAM	500MB	3200MB	100MB	28
CPU	0.5	3.2	0.25	11
Network	0.5Mbps	10Mbps	500Kbps	20

testbed described in Section 4. In this section, we describe our experimental design and results.

A. Experimental Design

To evaluate the performance of iPSA, we implemented three additional algorithms for hyperparameter optimization and resource allocation. Each algorithm solved the optimization problem outlined in equation 3. H_i was defined as a set of 12 resource-related hyperparameters: three for each of our 4 container classes. Each container's CPU allocation [0.5-3.2 CPUs], RAM [500MB - 3200MB], and ingress/egress network bandwidth [0.5MB-10MB] were treated as separate hyperparameters for optimization. Righteous used one of four algorithms, iPSA, Grid Search, Hyperopt and Ekya, to sample the hyperparameter search space and return a candidate H_{min}.

Our first point of comparison is Grid Search [36]. Grid Search is a simple hyperparameter optimization technique that discretizes the search space by some granularity γ and evaluates every point. Grid Search is effective for small search spaces, but its time complexity grows exponentially as the number of hyperparameters increases. For functions that are expensive to evaluate, like our workload, Grid Search can be prohibitively time consuming. We implemented Grid Search as an upper bound for comparison. We discretized our search space as shown in table I. A complete Grid Search of this space would require $1.43 * 10^{15}$ unique evaluations of our test case, which is infeasible. We therefore constrained our Grid Search further, changing RAM, CPU, and Network hyperparameters uniformly across all containers of all types, resulting in 6160 unique evaluations. Given the number of evaluations and runtime (average 41 seconds) of Grid Search samples, we ran Grid Search for only our 4 UAV swarm.

Hyperopt [7] is a state of the art Bayesian optimization software package used in many disciplines. As described in section 2, hyperopt is used in many disciplines to optimize black-box functions. We configure hyperopt to sample the hyperparameter space outlined in table I with no significant modifications.

Ekya [11] is a system for continuous learning in edge-based video analytics workloads. Like many edge and IoT applications, Ekya contains an online optimization algorithm for resource allocation. Ekya uses a thief scheduling algorithm to re-allocate resources from performant and over-provisioned components to under-performant components. Ekya's thief scheduler (referred to simply as Ekya) performs right-sizing online. Unlike our approach, however, Ekya does not converge to a minimal configuration with respect to resource utilization, but an optimal configuration with respect to performance. Ekya therefore assumes that a workload is presently saturating its system and re-allocates resources from one component to another. We modified Ekya slightly to suit hardware right-sizing. We allow Ekya's thief scheduler to 1) steal resource from a component and unallocate them, and 2) re-allocate freed resources. This allows Ekya to search the entire hyper-parameter space for a minimal hardware allocation.

iPSA was implemented as described in section 3. iPSA's start point was the mid-point of the resource allocation limits for each hyperparameter. iPSA used the same search space as all other algorithms described in table 1 with a temperature starting at 5, meaning $h_i' \in H' = h_i \pm D * 5 * \gamma_h$ at T_0. After each exploration of V', the temperature decreased by 1, resulting in 5 total temperature decreases. Both the exploration step and temperature schedule for iPSA are significantly lower than PSA [20], [41] due to intuition from principle 4.

To test the performance of our system as IoT workloads scale, we executed our test case using single UAV and swarms of size 2-5. For each swarm member added to our experiment, additional software components and resources are required, and bottlenecks change. We tested UAV swarms of up to size 5, as larger swarms exceeded the resources available on our PROWESS testbed. For each swarm size, we optimized to meet a single runtime goal based on execution time. Execution goals for each swarm size are as follows: 1 UAV: 5 seconds, 2 UAV: 7 seconds, 3 UAV: 9 seconds, 4 UAV: 11 seconds, 5 UAV: 15 seconds.

B. Performance Analysis

Figure 6 shows the performance results for Ekya, Hyperopt, iPSA, and Grid Search. For each algorithm, ran identically configured experiments for swarms of size 1-5. Each algorithm made no a priori assumptions about the underlying workload. Hyperopt results are split into two categories: Hyperopt 100 and Hyperopt 500. Unlike iPSA and Ekya, hyperopt does not have a clear stopping condition, so we executed Hyperopt for both 100 and 500 evaluations.

Figure 6 (a) shows the percent resource utilization of the testbed for the H_{min} returned by each algorithm. We see that iPSA returns configurations that use 1.33-3.57X less resources than all other algorithms. We also see that this performance is maintained as workloads scale. iPSA produces consistent results as resource requirements are increased, where Hyperopt and Ekya falter. This can be see in figures 6 (b) and 6 (c). Figure 6 (b) shows the total number of evaluations per workload for each swarm size. As mentioned, Hyperopt 100 and 500 have fixed evaluation sizes, but Ekya and iPSA do not. We see that Ekya searches less configurations as swarm size (and therefore resource allocations) increase. At the outset, Ekya's thief scheduler evenly splits resources across all containers. It then selects two containers (or one container and the system's unallocated resources) and swaps resources between, tracking the lowest cost feasible configuration. Each time it selects a pair of containers, it sets the allocations for all containers to this best performant configuration. This allows Ekya to quickly fall into local minima. While iPSA can also

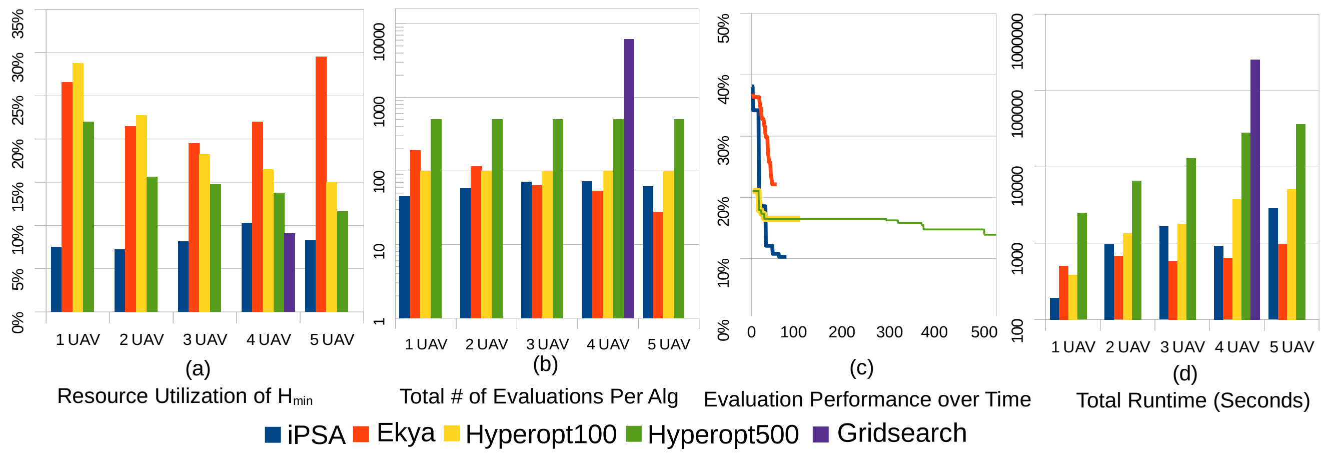


Figure 6. Righteous finds better configurations (a) with less evaluations (b-c) than other approaches and decreases overall runtime (d) through better guess selection.

experience this issue, the directed nature of its optimization improves performance over Ekya.

iPSA outperforms Hyperopt for different reasons. Hyperopt's black box function approximation process steadily improves performance, as seen by performance improvements between 100 and 500 evaluations. This is, however, a slow process. The hyperparameter space defined by even a relatively small IoT workload is far too large for Hyperopt to learn in so few optimization. For this reason, iPSA's targeted approach prevails. Ultimately, iPSA finds improved configurations far faster than Hyperopt, and faster than Ekya so long as Ekya avoids local minima.

These effects can be observed in Figure 6 (c), which shows each algorithm's best configuration at each evaluation step for our 4 UAV experiment. Ekya and iPSA are able to quickly find feasible, low utilization configurations, but Ekya also quickly finds a local minima and ceases optimization. iPSA is able to avoid similar local minima and find a configuration that uses 2.14X less resources. Hyperopt, conversely, samples broadly across the search space, methodically finding improvements. However, it does not eclipse iPSA even after 7X more evaluations.

Ekya and Hyperopt also increase runtime relative to performance. Figure 6(d) shows the runtime of each set of experiments in seconds. We can see both that 1) even a relatively simple deployment with only a few components to optimize can take hours or days to optimize with incorrect configuration, and 2) that iPSA minimizes runtimes. Because iPSA optimizes toward the Pareto bound, it spends less time executing configurations that are woefully under-provisioned or overly provisioned. Ekya and Hyperopt do not have this guarantee. Hyperopt, as it treats our problem as a black box, must sample broadly, resulting in increased average evaluation times 13.4-44X over iPSA. Specifically, in the 4UAV runtime of iPSA decreased because its final configuration was very close to the initial iPSA starting point, leading to an early exit.

Figure 7 shows the relationship between these algorithms and a) the frequency of their guesses with respect to resource utilization and runtime, and b) the progression of their guesses across evaluation steps. Images in figure 7 compress our 12-dimensional search space to two key results, total resource utilization of the system and workload runtime (our performance metric). This allows us to differentiate ideal H_i with (low resource allocations and low runtimes) from H_i with high resource allocations, or improper small allocations resulting in large runtimes.

Grid Search, whose guesses are predetermined, guesses widely and methodically across the search space. The most runtime-utilization pairs can be seen at moderate resource allocation levels resulting in small runtimes. These are over-provisioned but feasible configurations. Hyperopt similarly finds many feasible but over-provisioned configurations. Because Hyperopt treats our function as a black box, it guesses widely across the search space like Grid Search. It uses this information, however, to eventually navigate to ideal more H_i. We can see from Figure 7 that Hyperopt's early guesses vary widely, but later guesses focus near the ideal range.

Ekya navigates from highly provisioned guesses quickly to constrained guesses. Ekya's initial guesses are over-provisioned, resulting in low runtimes. As Ekya quickly pairs down resources, it ultimately reaches a point where guess runtimes begin to rise above performance goals. Ekya is ultimately unable to navigate out of the local minima and therefore returns a feasible but sub-optimal guess.

iPSA navigates the Pareto curve from the midpoint of the search space, which is over-provisioned in this case, to lower provisioned and nearly optimal configurations. We can see from Figure 7 (b) that iPSA navigates toward ideal H_i and slightly changes guesses resulting in improved feasible guesses and unsuccessful non-feasible guesses. The ideal region, outlined in green in Figure 7, is shown for both guess order

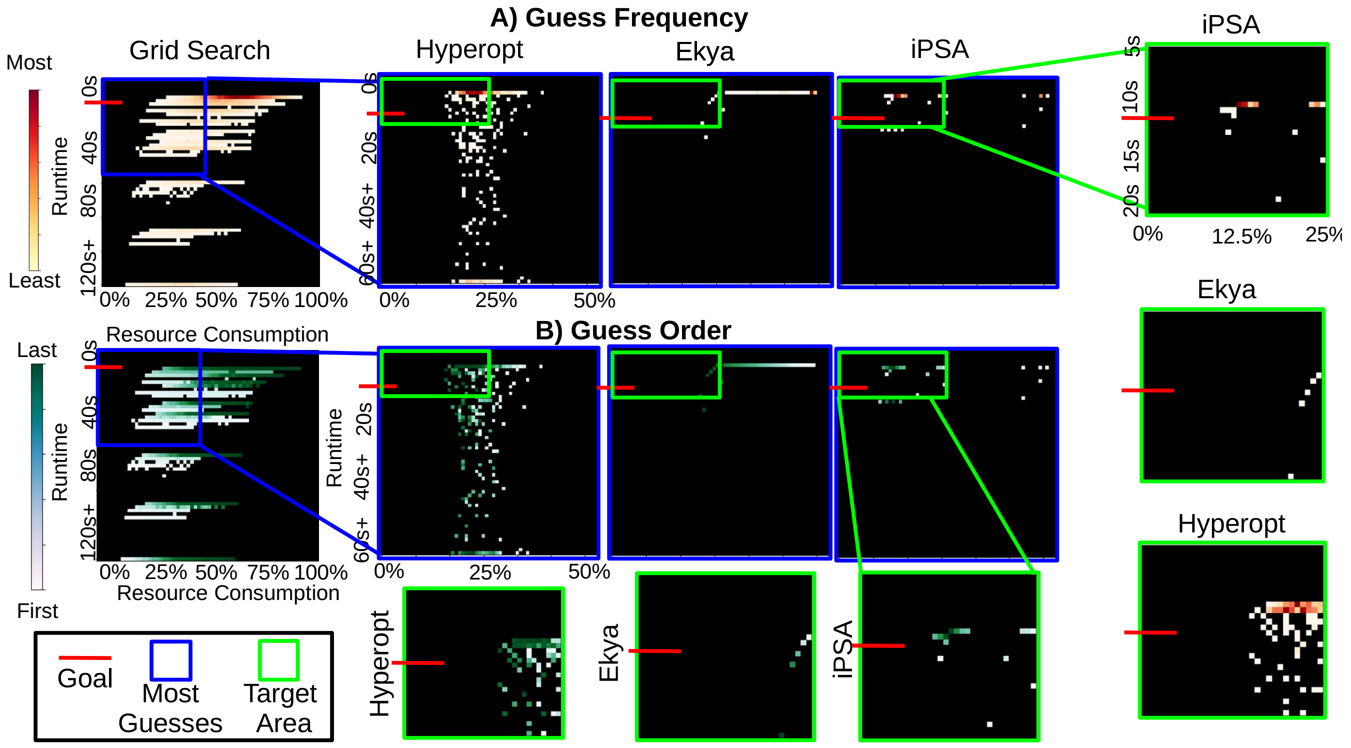


Figure 7. Righteous quickly narrows guesses down to ideal regions of the hyperparameter space as compared to other techniques, producing higher quality results faster than competitors.

and frequency for Ekya, iPSA, and Hyperopt. Ekya spends little time in this region, quickly falling into local minima. Hyperopt guesses broadly within the ideal region but is not quite capable of improving above iPSA. iPSA spends the majority of its runtime in this region (77% of guesses, as opposed to Ekya: 9% and Hyperopt: 49%). By navigating quickly along the pareto bound and guessing informatively, iPSA is able to outperform Hyperopt with 7X less guesses.

C. *Parallel Execution*

High levels of isolation and provisioning in computing testbeds provides an opportunity for parallel execution. Multiple experiments can be configured and executed at once so long as the testbed's resources can satisfy all requests. We added the following simple modification to our algorithms: for some guess H_i, if guesses $H_{i+1}...H_{i+N}, N \geq 0$ are know, we will run the maximal set of contiguous guesses $\{H_i...H_{i+k}\}$ that do not exceed available testbed resources.

Not all techniques benefit from parallelism. Bayesian optimization algorithms like Hyperopt execute iteratively and rely on information from previous trials to generate future guesses. Hyperopt can leverage parallelism, but will necessarily lose information resulting in decreased performance [7]. Ekya and iPSA both have exploration phases that are not adaptive and can be performed in parallel. Ekya's thief scheduler algorithm selects pairs of jobs such that job S_i steals resources from job S_j. This iterative stealing process is adaptive and can be parallelized. iPSA's neighborhood exploration process is likewise not adaptive and can be parallelized. Figure 8 shows the performance improvement garnered by iPSA and Ekya when their exploration phases are parallelized. We opted not to parallelize Hyperopt due to the aforementioned trade-off between parallelism and quality.

iPSA and Ekya both improve performance significantly via parallel experimentation. Ekya requires 2.01X-4X less evaluation steps, and iPSA requires in total 2.36X-3.55X less total evaluation steps when executing multiple workloads simultaneously. Furthermore, this improvement comes with no trade-off for both algorithms due to Kubernetes' isolation of testbed resources and independence of parallel workloads.

Runtime improvements for both iPSA and Ekya scaled slower than evaluations for both algorithms. Ekya runtimes improved by 1.3-3.5X where iPSA runtimes improved by 1.6-2.5X. Runtimes are decreased as parallel workloads tend to contain a mix of feasible and non-feasible guesses, where non-feasible guess runtimes can be large. Jobs whose runtimes are large can remain running after all other parallelizable jobs have finished, ultimately increasing runtimes relative to evaluations.

When considering parallelism, iPSA improves performance significantly over all other algorithms. iPSA finds better configurations (up to 3.5X compared to Ekya, and 2.9X compared to Hyperopt) and executes faster (up to 4.7X compared to Ekya, 76.3X compared to Hyperopt, and 1413X compared to Grid Search). Only Grid Search can find better configurations that iPSA, but Grid Search finds only a 10% improvement at

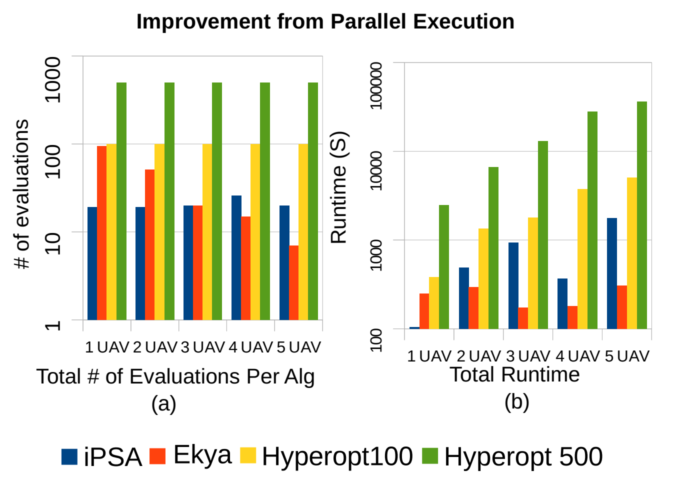


Figure 8. iPSA can be easily parallelized where other techniques can't, improving performance.

the cost of over 1000X additional runtime.

VI. Limitations and Future Work

Righteous was tested broadly with one representative edge workload across three resource types. It is likely that other types of workloads will perform differently when optimized by Righteous, but we believe that our results will generalize to other workloads. Furthermore, additional resources that are difficult to slice across workloads (e.g GPUs) will inhibit the performance of Righteous and the ability to right-size broadly.

Our UAV application can be mapped to deployment easily and automatically via regression, but this requires access to some candidate deployment hardware. Users may not have all hardware they require to properly map Righteous results to deployment conditions. Users can still incorporate Righteous into their development process to analyze applications for bottlenecks and approximate feasible resource requirements before deployment planning.

Righteous is based on our novel iPSA algorithm, which has several levers that this paper did not comprehensively explore due to space limitations. Instead we selected naive and understandable values as a user might select. We plan to explore temperature schedules in a followup work with additional evaluation for new applications. We also plan to explore the effects of model staging and early exits [24] on Righteous optimization in future work.

VII. Conclusion

Edge deployments are complicated, diverse, and impossible to comprehensively evaluate a priori. Users rely on computer systems testbeds to validate the correctness of their deployments, but optimizing deployments requires manual tuning or the employment of unsuitable techniques. We model right-sizing (i.e identifying a minimal hardware configuration that satisfies performance goals) as a hyperparameter optimization problem where resource allocations are hyperparameters. To solve this problem, we introduce Righteous. Righteous is an Edge deployment right-sizing tool that leverages the resource partitioning of modern testbeds and the novel iPSA algorithm to quickly right-size complex deployments. We tested Righteous on software from a previously deployed UAV swarm application. We found that Righteous was able to find feasible resource allocations that were up to 3.5X smaller than other optimization approaches. Righteous also found these allocations up to 7X faster than other optimization algorithms and over 1400X faster than naive exhaustive searches.

References

[1] Zahraa A Abdalkareem, Amiza Amir, Mohammed Azmi Al-Betar, Phaklen Ekhan, and Abdelaziz I Hammouri. Healthcare scheduling in optimization context: a review. *Health and Technology*, 11:445–469, 2021.

[2] Stavros P Adam, Stamatios-Aggelos N Alexandropoulos, Panos M Pardalos, and Michael N Vrahatis. No free lunch theorem: A review. *Approximation and optimization: Algorithms, complexity and applications*, pages 57–82, 2019.

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming 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, March 2017. USENIX Association.

[4] Barbara Arbanas, Antun Ivanovic, Marko Car, Tomislav Haus, Matko Orsag, Tamara Petrovic, and Stjepan Bogdan. Aerial-ground robotic system for autonomous delivery tasks. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 5463–5468. IEEE, 2016.

[5] Christian Attiogbé and Jérôme Rocheteau. Architectural invariants and correctness of iot-based systems. In *International Conference on Model and Data Engineering*, pages 75–88. Springer, 2022.

[6] Juan P Bello, Claudio Silva, Oded Nov, R Luke Dubois, Anish Arora, Justin Salamon, Charles Mydlarz, and Harish Doraiswamy. Sonyc: A system for monitoring, analyzing, and mitigating urban noise pollution. *Communications of the ACM*, 62(2):68–77, 2019.

[7] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015.

[8] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[9] Jan Beutel, Roman Lim, Andreas Meier, Lothar Thiele, Christoph Walser, Matthias Woehrle, and Mustafa Yuecel. The flocklab testbed architecture. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, page 415–416, New York, NY, USA, 2009. Association for Computing Machinery.

[10] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Cilantro: Performance-Aware resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 623–643, Boston, MA, July 2023. USENIX Association.

[11] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, Renton, WA, April 2022. USENIX Association.

[12] Jayson Boubin, Avishek Banerjee, Jihoon Yun, Haiyang Qi, Yuting Fang, Steve Chang, Kannan Srinivasan, Rajiv Ramnath, and Anish Arora. Prowess: An open testbed for programmable wireless edge systems. In *Practice and Experience in Advanced Research Computing*, pages 1–9. 2022.

[13] Jayson Boubin, Codi Burley, Peida Han, Bowen Li, Barry Porter, and Christopher Stewart. Marble: Multi-agent reinforcement learning at the edge for digital agriculture. In *Proceedings of the 7th ACM/IEEE Symposium on Edge Computing*, 2022.

[14] Jayson G Boubin, Naveen TR Babu, Christopher Stewart, John Chumley, and Shiqi Zhang. Managing edge resources for fully autonomous aerial systems. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 74–87. ACM, 2019.

[15] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[16] Matthew R Carbone, Hyeong Jin Kim, Chandima Fernando, Shinjae Yoo, Daniel Olds, Howie Joress, Brian DeCost, Bruce Ravel, Yugang Zhang, and Phillip M Maffettone. Emulating expert insight: A robust strategy for optimal experimental design. *arXiv preprint arXiv:2307.13871*, 2023.

[17] Tung-Chun Chang, Tirtha Banerjee, Nalini Venkatasubramanian, and Robert York. Quic-iot: Model-driven short-term iot deployment for monitoring physical phenomena. In *Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation*, pages 424–437, 2023.

[18] Tingjun Chen, Prasanthi Maddala, Panagiotis Skrimponis, Jakub Kolodziejski, Abhishek Adhikari, Hang Hu, Zhihui Gao, Arun Paidimarri, Alberto Valdes-Garcia, Myung Lee, et al. Open-access millimeter-wave software-defined radios in the pawr cosmos testbed: Design, deployment, and experimentation. *Computer Networks*, 234:109922, 2023.

[19] B. N. Chun, P. Buonadonna, A. AuYoung, Chaki Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: a microeconomic resource allocation system for sensornet testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, EmNets '05, page 19–28, USA, 2005. IEEE Computer Society.

[20] Piotr Czyżżak and Adrezej Jaszkiewicz. Pareto simulated annealing—a metaheuristic technique for multiple-objective combinatorial optimization. *Journal of multi-criteria decision analysis*, 7(1):34–47, 1998.

[21] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. Boat: Building auto-tuners with structured bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*, pages 479–488, 2017.

[22] Haluk Ergin and Tayfun Sönmez. Games of school choice under the boston mechanism. *Journal of public Economics*, 90(1-2):215–237, 2006.

[23] Emre Ertin, Anish Arora, Rajiv Ramnath, Vinayak Naik, Sandip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, Hui Cao, and Mikhail Nesterenko. Kansei: a testbed for sensing at scale. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN '06, page 399–406, New York, NY, USA, 2006. Association for Computing Machinery.

[24] Biyi Fang, Xiao Zeng, Faen Zhang, Hui Xu, and Mi Zhang. Flexdnn: Input-adaptive on-device deep learning for efficient mobile vision. In *Proceedings of the 5th ACM/IEEE Symposium on Edge Computing (SEC)*, 2020.

[25] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[26] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in {Multi-Resource} clusters. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 65–80, 2016.

[27] Xiaoxi Guo, Sujoy Sikdar, Lirong Xia, Yongzhi Cao, and Hanpin Wang. First-choice maximality meets ex-ante and ex-post fairness. *arXiv preprint arXiv:2305.04589*, 2023.

[28] Vikram Iyer, Hans Gaensbauer, Thomas L Daniel, and Shyamnath Gollakota. Wind dispersal of battery-free wireless devices. *Nature*, 603(7901):427–433, 2022.

[29] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 253–266, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Kyle D Julian and Mykel J Kochenderfer. Distributed wildfire surveillance with autonomous aircraft using deep reinforcement learning. *Journal of Guidance, Control, and Dynamics*, 42(8):1768–1778, 2019.

[31] Kate Keahey, Jason Anderson, Michael Sherman, Cody Hammock, Zhuo Zhen, Jenett Tillotson, Timothy Bargo, Lance Long, Taimoor Ul Islam, Sarath Babu, et al. Chi-in-a-box: Reducing operational costs of research testbeds. In *Practice and Experience in Advanced Research Computing*, pages 1–8. 2022.

[32] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. Lessons learned from the chameleon testbed. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 219–233, 2020.

[33] Haylee Lane, Mitchell Sarkies, Jennifer Martin, and Terry Haines. Equity in healthcare resource allocation decision making: a systematic review. *Social science & medicine*, 175:11–27, 2017.

[34] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. Rambo: Resource allocation for microservices using bayesian optimization. *IEEE Computer Architecture Letters*, 20(1):46–49, 2021.

[35] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. Systematically debugging iot control system correctness for building automation. In *Proceedings of the 3rd ACM international conference on systems for energy-efficient built environments*, pages 133–142, 2016.

[36] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: a big comparison for nas. *arXiv preprint arXiv:1912.06059*, 2019.

[37] Vuk Marojevic, Ismail Guvenc, Rudra Dutta, Mihail L Sichitiu, and Brian A Floyd. Advanced wireless for unmanned aerial systems: 5g standardization, research challenges, and aerpaw architecture. *IEEE Vehicular Technology Magazine*, 15(2):22–30, 2020.

[38] Tommaso Melodia, Stefano Basagni, Kaushik R Chowdhury, Abhimanyu Gosain, Michele Polese, Pedram Johari, and Leonardo Bonati. Colosseum, the world's largest wireless network emulator. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 860–861, 2021.

[39] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[40] David Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114(3):528–541, 1999.

[41] Sabri Pllana, Suejb Memeti, and Joanna Kolodziej. Customizing pareto simulated annealing for multi-objective optimization of control cabinet layout. In *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*, pages 78–85. IEEE, 2019.

[42] Guillaume Rosinosky, Donatien Schmitz, and Etienne Rivière. Streambed: Capacity planning for stream processing. In *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems*, DEBS '24, page 90–102, New York, NY, USA, 2024. Association for Computing Machinery.

[43] Abusayeed Saifullah, Mahbubur Rahman, Dali Ismail, Chenyang Lu, Ranveer Chandra, and Jie Liu. Snow: Sensor network over white spaces. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 272–285, 2016.

[44] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[45] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[46] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for {Large-Scale} advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[47] Ian von Hegner. A limbus mundi elucidation of habitability: the goldilocks edge. *International Journal of Astrobiology*, 19(4):320–329, 2020.

[48] Ke Wang and Alexander W Dowling. Bayesian optimization for chemical products and functional materials. *Current Opinion in Chemical Engineering*, 36:100728, 2022.

[49] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: a wireless sensor network testbed. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 483–488, 2005.

[50] Tayyaba Zainab, Jens Karstens, and Olaf Landsiedel. Lighteq: On-device earthquake detection with embedded machine learning. In *Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation*, pages 130–143, 2023.

[51] Zichen Zhang, Sami Khanal, Amy Raudenbush, Kelley Tilmon, and Christopher Stewart. Assessing the efficacy of machine learning techniques to characterize soybean defoliation from unmanned aerial vehicles. *Computer and Electronics in Agriculture*, 2022.