

Wayfinder: Automated Operating System Specialization

Alexander Jung
Lancaster University
United Kingdom
Unikraft GmbH
Germany

Cezar Crăciunoiu
Politehnica University of Bucharest
Romania

Nikolaos Karaolidis
The University of Manchester
United Kingdom

Hugo Lefeuve
The University of British Columbia
Canada

Daniel Oñoro Rubio
NEC Laboratories Europe
Germany

Felipe Huici
Unikraft GmbH
Germany

Charalampos Rotsos
Lancaster University
United Kingdom

Pierre Olivier
The University of Manchester
United Kingdom

Abstract

Operating system specialization is a well-known approach to optimize a specific application's performance, memory usage, security, or other important metrics. Specializing an OS for an application is typically a manual process that requires great expertise. Specialization through configuration lends itself well to automation; however, it is challenging due to the sheer size of the configuration space of modern OSES, the difficulty to quantify that space, the long time it takes to evaluate a configuration, and the large number of invalid configurations. Hence, existing attempts at specializing OSES automatically are limited to switching features on and off to minimize memory consumption or attack surface, and cannot target metrics such as performance.

We present Wayfinder, a framework specializing the configuration of OSES completely automatically and without expert knowledge. It can specialize all aspects of an OS configuration (compile-/boot-/run-time) towards any quantifiable performance, resource consumption, or security metric, for an application processing a given workload on a given hardware setup. Wayfinder consists of an automated OS benchmarking platform, and a neural network-based search algorithm driving the specialization process. This is achieved by learning on the fly which configuration parameters and values impact performance the most, and which ones lead to runtime failures. Optionally, a model pre-trained on one application can be reused to accelerate the specialization of related applications. We evaluate Wayfinder on two OSES,

four applications, and two target metrics: Wayfinder fully automatically identifies specialized configurations with up to 24% application performance improvement and 8.5% memory usage reduction compared to default configurations. We highlight the benefits of our neural network, reaching good solutions faster than competing approaches (random and Bayesian), and successfully transferring knowledge between related applications.

CCS Concepts: • Software and its engineering → Operating systems; Software configuration management and version control systems.

Keywords: Operating Systems, Specialization

ACM Reference Format:

Alexander Jung, Cezar Crăciunoiu, Nikolaos Karaolidis, Hugo Lefeuve, Daniel Oñoro Rubio, Felipe Huici, Charalampos Rotsos, and Pierre Olivier. 2026. Wayfinder: Automated Operating System Specialization. In *21st European Conference on Computer Systems (EUROSYS '26)*, April 27–30, 2026, Edinburgh, Scotland Uk. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3767295.3803589>

1 Introduction

The general-purpose nature of mainstream Operating Systems (OSes) comes at the expense of performance, resource consumption, or security [13, 15, 25, 46, 48, 70]. Thus, a common technique to trade off the cost of generality is *OS specialization* [8, 25, 33, 46], the process of tailoring an OS towards specific use-cases such as applications, workloads, or hardware platforms. OS specialization is a popular systems research topic, with works targeting I/O [15, 42, 57, 65, 70, 73], GPU or hardware accelerators [81–83], resource usage [2, 42, 44, 54, 55], security [45, 49, 80, 91], compatibility [9, 66, 67, 69, 90], or extensibility [14].

Most OS specialization efforts rely on expert developers manually tuning or re-implementing specific OS subsystems. Unfortunately, such approaches depend on expert domain



This work is licensed under a Creative Commons Attribution 4.0 International License.

EUROSYS '26, Edinburgh, Scotland Uk

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2212-7/2026/04

<https://doi.org/10.1145/3767295.3803589>

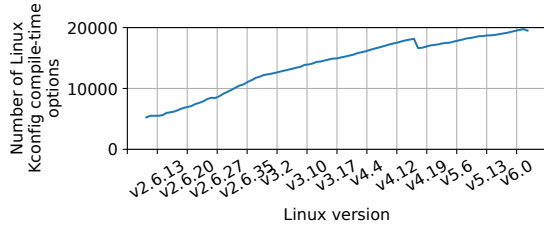


Figure 1. Linux compile-time configuration space over time.

knowledge, preventing the vast majority of users from reaping the benefits of these techniques. To address this problem, we need more automation in OS specialization.

We focus on OS specialization *through configuration*, where an OS enables users to customize its behavior and tune its functionality for specific use-cases through compile-, boot-, or runtime configuration parameters. Although the benefits of configuration specialization can be arguably more modest than those obtained by redesigning parts of an OS, they have been shown to be significant in many scenarios [39, 44, 45, 77, 88]. Further, this approach lends itself well to automation, and can be achieved without expert knowledge. Although prior work has explored automatically switching features on and off to reduce resource consumption [1, 40, 77] or attack surface [45, 88, 89], none can handle the challenges of the vast and complex configuration space offered by modern OSes when targeting other metrics such as performance.

We present Wayfinder, a framework for specializing the configuration of OSes, such as Linux, for applications *fully automatically*, and *without expert knowledge*. Wayfinder automatically specializes any aspect of an OS configuration, such as compile-time, boot-time, and runtime parameters, towards any quantifiable performance, resource consumption, security, or compatibility metric, for a target application processing a given workload on a given hardware setup. The framework includes an automated OS configuration, build, and benchmarking platform, designed to support reproducible testing. At its core, Wayfinder builds on DEEPTUNE, a novel neural network-based optimization algorithm, to drive the specialization process. The key idea of DEEPTUNE is to direct the search by selecting interesting candidate configurations through predicting 1) their performance on a target metric (e.g., throughput, memory usage) and 2) their likelihood to be valid (e.g., to compile, not to crash at runtime).

Wayfinder addresses several challenges. First, the configuration space of modern OSes is extremely large. As shown in Figure 1, Linux 6.0 exhibits about 20 000 compile-time options, on top of boot-time and runtime parameters. Some parameters take arbitrary numbers as values, which exacerbates the issue: overall, it is not possible to explore every configuration. Second, the search space is hard to define without expert knowledge. Many kernel parameters offer

little to no documentation: parameter types and valid ranges of values are unknown. Third, evaluating a configuration takes time, as it potentially requires building a kernel image, booting it, and running a test to evaluate the target metric. The time required to evaluate a configuration can vary significantly, and an automated OS tuning system should optimize the overall time to discover a specialized configuration. Finally, the search space contains many configurations that are valid on paper (e.g., on Linux, they satisfy constraints checked by KConfig), but lead to failure at compile-time, boot-time, or runtime. Our evaluation demonstrates that, when using a naive (random search) approach to optimize the configuration of a Linux OS, approximately one third of all attempts fail, which represents a significant number of wasted search iterations. Combined with slow evaluation, this severely limits the number of configurations that can be explored in a given time budget.

Wayfinder’s design tackles these challenges as follows. For OSes with large and hard-to-qualify search spaces (e.g., Linux), we determine the search space offline using a heuristic algorithm that infers both types and ranges for each OS parameter. Then, our online exploration relies on DEEPTUNE, a neural network optimization algorithm that progressively learns the OS configuration parameters and values with the most significant performance impact for a target metric. To speed up the search, DEEPTUNE learns to avoid parameter values that are likely to trigger failures, a feature that competing methods, such as random search or Bayesian optimization, lack. By default, the learning process starts from scratch for every application/metric to specialize towards. Optionally, after training a model to optimize for a given application, transfer learning can be applied, i.e., the model can be reused to accelerate exploration on other applications with similar characteristics.

We apply Wayfinder to specialize the configuration of two OSes, Linux and Unikraft [42], for popular cloud applications. We optimize for two target metrics: application performance and memory usage. Unlike competitors such as causal reasoning [38] or Bayesian optimization [39], Wayfinder scales to the vast design space of modern OSes such as Linux, and discovers specialized configurations several times faster from a random search baseline. Wayfinder is efficient at predicting failures, reducing crash rates from 30% to 10-25%. Lastly, we demonstrate that transfer learning is effective in accelerating search time and reducing crash rates to less than 10%. For example, with a model trained on Redis and applied to Nginx, the search is sped up by 24%, with crash rates below 5%.

Overall, this paper makes two core contributions:

- Wayfinder, an evaluation platform able to configure, build, run, and benchmark OSes automatically and without expert knowledge.
- DEEPTUNE, a novel neural network optimization algorithm that drives Wayfinder’s specialization process.

Table 1. Configuration space for Linux 6.0, including boot time options (kernel command line parameters), runtime options (writable files in `/proc/sys` and `/sys`), and compile-time options (obtained by parsing the Kconfig hierarchy).

Compile-time options					Boot-time options	Runtime options
bool	tristate	string	hex	int		
7585	10034	154	94	3405	231	13328

2 Background, Challenges, and Motivation

2.1 Specialization through Configuration

Specialization optimizes an OS for a given use-case (e.g., application, workload, hardware platform) and a given metric (e.g., performance, resource usage). There are many approaches to OS specialization [14, 42, 45, 49, 54, 69, 70, 81]. We focus on specialization *through configuration* [1, 40, 44, 45, 55, 66, 73, 80, 89], where an OS is specialized by fine-tuning its compile-time, boot-time, and runtime parameters.

Prior work explored OS specialization through configuration with the goals of reducing resource usage (e.g., memory footprint) [1, 40, 44, 55, 66, 73, 77] and attack surface [45, 80, 88, 89]. Some of these approaches are manual [42, 44, 66, 73, 80], hence they do not scale to more than a few applications given the size of the configuration space of real-world OSes such as Linux [44, 73, 80]. To address this, other works explored automated methods to specialize OS configurations [1, 40, 45, 77, 88, 89]. These efforts primarily focus on security (attack surface) or resource optimization (memory footprint). Their objective is thus to determine, for a given workload, which compile-time configuration parameters (e.g., kernel features or modules) can be disabled, and which ones are essential for the workload.

These approaches are not generic, and do not apply to performance specialization towards performance. Beyond enabling/disabling only compile-time features, we need to consider the full set of kernel options, many of them taking arbitrary values, with unclear/undocumented ranges of validity. To scale to many applications we cannot assume expert knowledge (i.e. no filtering of relevant options). In short, the size of the exploration space becomes quasi-infinite. Consider for example the size of the configuration space for the Linux kernel, illustrated in Table 1. The compile-time configuration of Linux includes more than 3000 options (out of the 20 000 total) taking arbitrary integers. Furthermore, due to its general-purpose nature, Linux must cater to a wide range of workloads, and thus many performance-critical configuration options are available only in the form of run-time parameters. Setting these parameters correctly to maximize performance, even manually and with expert knowledge, is complex, as demonstrated by the many performance tuning guides available online [10, 16, 21, 22, 28, 58, 63, 76, 87]. For

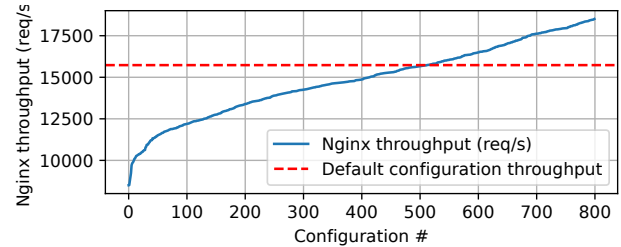


Figure 2. Nginx throughput for 800 random configurations of the Linux kernel.

these reasons, automated OS specialization through configuration with performance goals has not been explored in many prior works.

2.2 Optimizing OS Configurations for Performance

To further motivate OS performance specialization through configuration, we randomly generate 800 configurations of Linux v4.19 and evaluate their performance. For each random configuration, we configure and run a corresponding kernel in a KVM virtual machine, where we execute and benchmark an Nginx web server with wrk. We run these experiments on an Intel Xeon E5-2697 v2 (2x24 cores@2.70 GHz, 128 GB RAM). We want to obtain 800 valid configurations so when one fails (about a third of randomly generated configurations crash at runtime), we re-generate a random configuration until we obtain a valid one.

Figure 2 presents the results comparing the performance of each random configuration to the default one. The configurations are sorted in ascending performance order. The performance varies by as much as 80%, from under 10K up to 18K req/s. A key observation is that the throughput of the fastest configuration is 12% higher than the throughput obtained with the default configuration, even though we explored only a tiny fraction of the design space.

This motivates the usefulness of optimizing OS configurations for metrics such as performance. Still, the random search approach taken here is suboptimal: on this small space, 64% of the configurations perform worse than the default configuration. Further, one third of the configurations lead to situations in which the kernel cannot build/boot or crashes/hangs at runtime (referred to as a whole as *crashes* in the rest of this paper), wasting resources. Given the sheer size of the exploration space, and the lack of consideration for valid/invalid configurations, finding specialized configurations through random search would be unacceptably slow. Hence, we propose Wayfinder to drive the configuration space exploration using efficient optimization algorithms.

2.3 Optimization Algorithms for Configuration-Based OS Specialization

We approach the search for specialized OS configurations as an optimization problem. Past work used various approaches,

such as Bayesian optimization [5, 39], causal reasoning [38], and random search [39]. However, several issues cause these approaches to be inapplicable or inefficient:

Scalability. Bayesian optimization relies on Gaussian processes, which typically have a computational complexity of $O(n^3)$, and $O(n^2)$ for memory consumption. The complexity of causal analysis algorithms ranges from $O(n^4)$ to $O(n^3)$ [98], which limits them to relatively small problems.

Lack of incremental training. Adding new data points and updating the model typically requires retraining a Gaussian process in the case of Bayesian optimization. It also requires recomputing the causal graph for causal inference. Combined with the scalability issue, this leads the cost of each iteration of the optimization to grow exponentially.

Difficulty to fit both categorical and numerical parameters for high-dimensional data. Another well-known limitation of Bayesian optimization is its poor performance on problems with categorical input features [31] (i.e. features with a fixed set of values), and with high-dimensional inputs.

Overall, the very large configuration space we target makes both causal inference and Bayesian optimization poor fits. Random search performs well on such large spaces, but it suffers from a high crash rate for this problem (one third of the configurations randomly sampled fail). We further demonstrate these limitations in the evaluation (Section 4).

3 Wayfinder: Design and Implementation

The sheer size of the exploration space and the impracticality of relying on expert knowledge motivate the need for an *automated* exploration platform. That platform can be programmed to automatically configure, build, and benchmark series of OS images with the goal of specializing them towards specific applications and workloads.

Overview. Wayfinder is composed of a benchmarking pipeline (§3.1) that automatically builds, configures, and benchmarks OS and application images, and of DEEPTUNE (§3.2), an optimization algorithm that drives the exploration to find specialized OS configurations for a given application.

3.1 Automated Benchmarking Pipeline

Wayfinder takes as input YAML files representing the configuration space of the target OS (*job files*, discussed in §3.4) and scripts describing how to build and benchmark images of that OS—including the application under test.

With Wayfinder the exploration process specializing an OS for a given application consists of iteratively executing the following core loop: 1) build and boot an OS image based on a given configuration in a VM; 2) benchmark the target application running on that OS image; and 3) determine the next configuration to consider. The user provides the platform with a time budget or a number of iterations to run, after which the best configuration found is returned. Wayfinder offers a modular API to ease the integration of pluggable

search algorithms, which accomplishes step 3. These algorithms drive the OS specialization process for an application and workload/metric by deciding what configuration to explore next using various approaches, and we currently have support for the following:

- *Random search:* each subsequent configuration to explore is generated randomly without considering the exploration history.
- *Grid search:* all possible configurations are explored systematically, one parameter value after the other.
- *Bayesian optimization:* an approach balancing exploration (trying new configurations) and exploitation (trying to optimize further configurations known to perform well) using a probabilistic model built from the exploration history.
- *DEEPTUNE:* an ML-based approach balancing exploration and exploitation described below in §3.2.

These algorithms interact with Wayfinder through an API exposing various information such as the history of configurations explored, the corresponding performance¹ results, which configurations resulted in build failure or runtime crashes, etc. Once a configuration is selected for evaluation, the platform creates two corresponding internal tasks: a build task to create the OS image, and a test task to measure its performance. An optimization here is that the build task can be skipped if the differences between the current configuration to explore and the previous one only relate to runtime parameters and not boot/compile-time ones.

Wayfinder is built in 15K LoC of Go as a collection of microservices. It uses off-the-self components for persistence, monitoring and logging. The platform runs on a Linux host, and benchmarked OS images execute on top of QEMU/KVM.

3.2 DEEPTUNE

DEEPTUNE is the AI algorithm that drives Wayfinder’s automatic optimization. With our key assumption that the user has no performance-tuning expertise, faced with the previously-described gigantic configuration space, the intuition behind DEEPTUNE’s design is that we need to combine *exploration*, i.e., trying new parameters to find those that matter, and *exploitation*, i.e., optimizing parameters identified as important. This approach automatically subsets the configuration space to focus on the most impactful parameters (vs. heuristics that may require variable amounts of expertise), while ensuring that it does not become trapped in local optima. DEEPTUNE addresses the limitations we observe in existing automatic optimization algorithms such as Bayesian optimization and causal inference [38] (see §2.3) to achieve both high accuracy and scalability. It does so through (1) a new Neural Network (NN) model design that predicts

¹In the following, we use the term “performance” to refer to the output of one or more user-provided metrics, which may refer to any quantifiable measure(s) such as throughput, latency, memory usage, image size, etc.

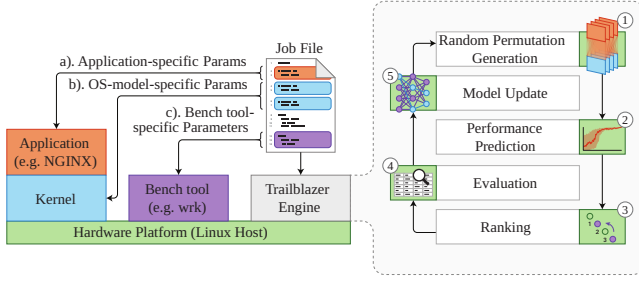


Figure 3. DEEPTUNE’s general process.

behaviors of randomly generated permutations, and (2) a new scoring function that ranks configurations based on the predictions. The former (1) is embodied in the DEEPTUNE Model (DTM), a multitask NN that predicts runtime performance and the likelihood that a configuration will crash at runtime. The DTM also provides a measure of uncertainty through a new mechanism based on Radial Basis Function (RBF) layers [47] and the Chamfer distance loss [26]. The scoring function (2) ranks candidate configurations by merging the model prediction, the predicted uncertainty, and the dissimilarity in relation to known configurations.

Figure 3 illustrates the key components of DEEPTUNE. DEEPTUNE starts with the random generation of a diverse pool of permutation candidates ① from Wayfinder. The DTM then estimates the performance of these candidates ②, and the scoring function ③ ranks them. The top permutation is evaluated by Wayfinder ④, and the DTM is updated ⑤. The algorithm iterates for a predetermined number of cycles or until a performance goal is achieved.

The DEEPTUNE Model (DTM). This section is designed to be read along with Figure 4. The DTM (Figure 4) is a multitask Neural Network (NN) that predicts whether a configuration will crash, its expected performance, and the performance uncertainty of these predictions. It consists of two branches: the *prediction branch*, which is a conventional NN that predicts permutation crash probability and performance, and the *uncertainty branch*, which is a Radial Basis Function NN that estimates the uncertainty of the predicted performance.

In the following, we note vectors in bold (\mathbf{x}), matrices in capitals (X), and scalars in lower-case (α). We can define the DTM as a function $F(\mathbf{x}) \rightarrow (\hat{k}, \hat{y}, \hat{\sigma})$ that maps a configuration permutation \mathbf{x} into the crashing probability \hat{k} , the expected performance \hat{y} , and the predicted uncertainty $\hat{\sigma}$. Each configuration can be divided as $\mathbf{x} = (\mathbf{x}^k, \mathbf{x}^n)$, with $\mathbf{x}^k \in \mathbb{N}^k$ the subset of discrete/categorical parameters (e.g., Nginx’s DEFAULT_QDISC parameter can be a string “pfifo”, “bfifo”, etc.) and $\mathbf{x}^n \in \mathbb{R}^n$, for the continuous/integer parameters. $F(\mathbf{x})$ can be divided into its uncertainty branch $F^u(\mathbf{x}) \rightarrow (\hat{k}, \hat{y})$ and prediction branch $F^p(\mathbf{x}) \rightarrow (\hat{\sigma})$.

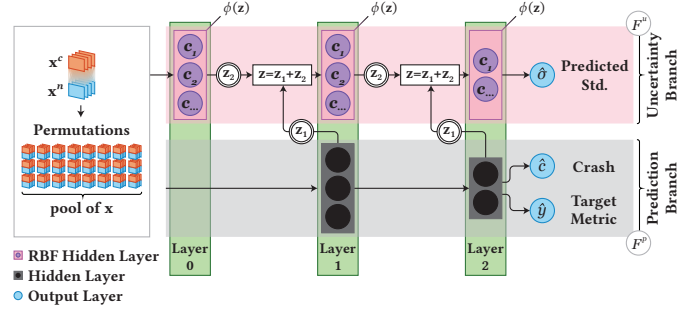


Figure 4. DEEPTUNE Model overview.

F^p is a conventional feedforward deep NN [41, 56, 86]. It consists of a sequence of dense layers with *Rectified Linear Unit (ReLU)* activation functions and dropout [86] layers. Its last layer outputs performance and crash predictions.

F^u (pink area in fig. 4) is specifically designed to estimate uncertainty. It consists of a stack of Gaussian Radial Basis Function (RBF) layers [47], each consisting of neurons ϕ that contain centroids $\mathbf{c} = (c_1, c_2, c_3, \dots)$. These centroids, learned throughout the training process, can be interpreted as learned features (or *prototypes*) from the dataset. Each RBF layer is parallel to a layer of the prediction branch. It takes as input \mathbf{z} , the concatenation of the features/latents output by the previous layer. The activation value ϕ is computed as:

$$\phi(\mathbf{z}) = \exp\left(-\frac{\|\mathbf{z} - \mathbf{c}\|^2}{2\gamma^2}\right) \quad (1)$$

γ is a smoothing parameter that controls how flat the activation curve of each neuron is. This parameter should be empirically fit: we find that a γ value of 0.1 is appropriate if input features are z-score normalized. The motivation behind this design is to be robust to outliers or totally new samples: the response of each neuron depends on the distance to the learned centroid or data prototypes (represented by the norm $\|\dots\|$), hence when an outlier appears, the output is low.

Training the DTM. We train the DTM end-to-end by minimizing the loss function $\mathcal{L} = L_{\text{CCE}} + L_{\text{Reg}} + L_{\text{Cham}}$. We import these three components from prior works:

- The categorical cross-entropy loss L_{CCE} [56] enables the DTM to identify which permutation might crash through learning historical data.
- The regression loss with uncertainty L_{Reg} [41] addresses the challenge of performance prediction while quantifying uncertainty. Initially contributed for computer vision tasks [41], we can use it to allow the DTM to both estimate the expected performance profile of a given permutation and provide an estimation of the expected error of the prediction. This enables better choices in the next permutation sampling policy.
- The regularization loss L_{Cham} [26] for RBF layers enables the DTM to learn centroids \mathbf{c} that fit the training

data. Intuitively, L_{Cham} is the *Chamfer* [26] distance, initially contributed in computer vision to compute the distance between two point clouds [12]. Here, it computes the distance between the centroids and the input data; minimizing it effectively distributes centroids such that they fit the training data distribution.

While L_{CCE} , L_{Reg} , and L_{Cham} exist from prior works in other fields [12, 26, 41, 56], we contribute a new way to use them in \mathcal{L} to solve our problem.

The scoring function. We must strike a balance between exploration and exploitation when selecting the next configuration to try. We thus contribute a scoring function $\text{sf}(\mathbf{x}, X)$ that leverages both the dissimilarity of a sample \mathbf{x} compared to known sample points X , and their estimated error or uncertainty $\hat{\sigma}$ to prioritize exploration in under-explored regions, while exploiting areas with greater expected improvements.

Specifically, the scoring function starts by calculating the dissimilarity $\text{ds}(\mathbf{x}, X)$ between the candidate point \mathbf{x} in the search space and known sample points X . This dissimilarity metric accounts for the diversity of the samples and indicates regions that have not yet been thoroughly explored:

$$\text{ds}(\mathbf{x}, X) = 1 - \frac{1}{(1 + \|\mathbf{x} - X\|_2^2)} \quad (2)$$

We calculate the final score by combining the sample dissimilarity and the predicted sample uncertainty, represented by $F^u(\mathbf{x})$ (which maps \mathbf{x} to $\hat{\sigma}$, see Figure 4), using a predefined weight α :

$$\text{sf}(\mathbf{x}, X) = \alpha \text{ds}(\mathbf{x}, X) + (1 - \alpha) F^u(\mathbf{x}). \quad (3)$$

Our experiments indicate that setting α to 0.5 strikes an effective balance between exploitation and exploration.

The justification behind our choice of loss and scoring functions is as follows: the loss function trains the system to simultaneously identify invalid (crashing) configurations and predict performance, while the scoring function uses those predictions to strategically choose new configurations that balance exploration vs. exploitation. The RBF branch adds a mechanism that allows DEEPTUNE to estimate its confidence/uncertainty.

Our implementation of DEEPTUNE optimizes for a single metric at a time. However, it can be extended to handle multiple metrics by adding additional output layers to F^p and F^u . This modification allows the DTM to make predictions for multiple targets simultaneously. During the scoring phase, we apply equation 3 to each target metric to obtain individual scores. Then, we calculate a representative score for each permutation sample by taking a weighted average, or using another aggregation method, of these individual scores.

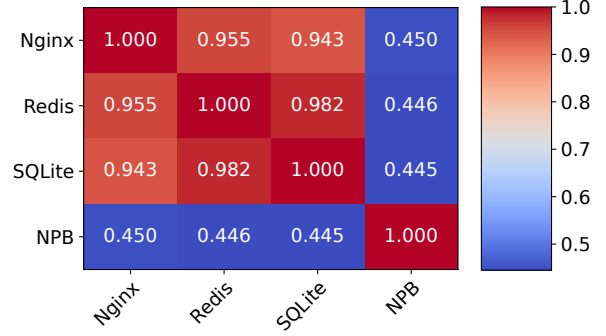


Figure 5. Cross-similarity matrix: a value close to 1 at the intersection of a column and a row means that the performance of the applications is impacted by similar parameters.

3.3 Transfer Learning

By default, each process of optimizing for one application and metric starts with a blank model, and parameters impacting performances/crashing must be re-learned from scratch. Transfer learning [36] consists of pre-training a model on one application, and re-using it to speed up the optimization process for another related application. The intuition behind transfer learning lies in the similarity of features between tasks: when applications share characteristics and the metrics to optimize towards are similar, it is probable that a model pre-trained on one application will be useful for the other. In other words, the performance of two applications may be sensitive to the variation of the same configuration parameters. For example, both Redis and Nginx are network-intensive: when an instance of DeepTune is trained on one, the subset of parameters and values that matter for the other (e.g., network stack parameters) has already partially been identified, speeding up the search. Conversely, that particular instance will be ineffective with NPB which is CPU-/memory-intensive.

To confirm that hypothesis, we build a cross-similarity matrix (Figure 5) to assess the similarity/differences between the most performance-impactful kernel configuration options for the four applications considered in the evaluation (see §4 for details). To create that matrix, we first collect 2,000 random Linux configurations for each application. Then, we use a feature importance algorithm [17] to determine the importance of each configuration option in predicting performance. Finally, we treat the importance scores as vectors and compute the Euclidean-norm distance between them. The parameters with the highest performance impact are similar for Nginx, Redis, and SQLite, which are all system-intensive. Redis is closer to SQLite than it is to Nginx, which is unsurprising as Redis and SQLite are both databases. NPB, on the other hand, exhibits sensitivity to different parameters due to its compute and memory-intensive nature. This suggests that transfer learning can be effective for certain classes of applications in Wayfinder.

3.4 Defining the Exploration Space

Wayfinder requires a description of the configuration space to start the exploration process (the *job file*, as mentioned in §3.1). That description includes the list of configuration parameters, their types, and the possible values they can take. Modern systems, such as Linux, have complex configurations, so completely describing the configuration space is challenging. Although some information can be statically obtained for compile- and runtime parameters (e.g. by analyzing `Kconfig` files and kernel command line parameter descriptions [53]), it is not the case for runtime parameters: these can be poorly or not documented, and for many parameters the range of valid values cannot be statically determined, because it depends on runtime aspects (e.g., the amount of available RAM).

We developed the following heuristic approach to determine the Linux configuration space. Linux offers multiple runtime configuration options through virtual file systems, e.g. `/proc/sys`, `/sys`. We first determine all configuration options by booting a VM with the version of Linux under investigation, and listing writable files in these paths. For each writable file, we read it and assume the value returned corresponds to the default value for the corresponding configuration options. Next we determine the type of the option by checking the type of the default value. If it is a number and equals 0 or 1, we assume the option is boolean. If it is neither 0 nor 1, we treat it as an arbitrary integer.

Finally, we estimate the range of possible values for the option by scaling up and down the default value several times by a high factor (10) and attempting to set the option to these new values by writing to the corresponding pseudo file. If the write operation succeeds and the VM does not crash, we consider the new value to be in the valid range for that option. The exploration of configuration values is left intentionally coarse, as it will be the task of Wayfinder to find performance improvements. Note that this technique excludes configuration parameters that are not numbers (e.g., strings), for which valid values would be very hard to determine automatically. There are very few non-numeric runtime parameters (see Table 1), and for these, we call back to manual exploration when necessary. Wayfinder will explore categorical parameters one by one, with no assumption of relationship (e.g., linear) between the values they can take. String parameters will not be explored beyond the values that can be automatically extracted. Beyond Linux, this approach should generalize easily to other OSes. We argue that more formal specifications of kernel parameters would be desirable in the future, taking inspiration from the system call interface definitions contributed by the fuzzing community [32].

3.5 Integration in Deployment Workflows

Practicality: Integrating Wayfinder in Common Deployment Workflows. We envision Wayfinder to be used

in the testing/evaluation phases of an application’s development. Equipped with a workload and a machine representative of deployment conditions, the system optimizes the kernel’s configuration for the application running in these conditions. At that stage, if a configuration identified by the platform is to be deployed to production, an engineer should review the deployment to ensure that it meets production requirements. While Wayfinder can check that the system is functional as part of its benchmarking step, like any AI tool it cannot exonerate a site reliability engineer (and, typically, a CI pipeline) from checking the configurations it pushes to production. If a configuration does not meet production requirements, users can further constrain the exploration to weed out that part of the space:

- *Constraining the parameters covered by the search.* Users can optionally specify a set of parameters which should take fixed values and will not be varied by Wayfinder’s search process. This is for example useful to ensure that relevant security options (ASLR, etc.) are not disabled by Wayfinder. Wayfinder can also be instructed to favor varying certain parameter types (compile-time/boot-time/runtime), which is useful, e.g., when the kernel to optimize cannot be rebooted.
- *More comprehensive benchmarks.* Users can also extend the benchmarking tool used by Wayfinder (purple in Figure 4) to check additional functionalities of the deployment (e.g., run a test suite). If the check fails, Wayfinder will learn the kernel configurations that cause the misbehavior.

Overall, our design builds on a simple assumption: while users need not have expert knowledge on OS performance optimization, we assume that they can check if a deployment meets their production requirements.

Security Considerations. In security-sensitive scenarios, relying without expert knowledge on a fully automated framework raises legitimate safety concerns. The threat model here is as follows: without awareness of security-relevant options, Wayfinder may overlook, disable, and optimize out such options, leading to kernel configurations which may be exploited in the field if deployed as is.

As mentioned previously, we can enable a security-aware search mode by fixing security-critical parameters to safe values. In a context where we envision no particular expertise from the user, this means that we assume the capacity to identify such important security-related options. We believe that this assumption is justified: commonly-used security parameters can be identified by Wayfinder’s developers, requiring no effort/expertise from the user. It is also safe to assume that the user is aware of any non-standard security parameter that should be set to a particular value, otherwise they would not be able to run their application securely in the first place. Finally, as mentioned above, like with any

other automated tool, Wayfinder’s output should be checked by a reliability engineer before being pushed to production.

Sensitivity to Workload and Hardware. Wayfinder specializes a kernel configuration for a particular application running on a particular platform and processing a particular workload. Similarly to most performance evaluation works, a change in workload or hardware requires rerunning the evaluation to obtain accurate results for the software and hardware considered. Wayfinder could be extended to predict performance for hardware/workloads that are different from those evaluated, using cross-workload [96] and cross-platform [92] performance estimation methods e.g., from the heterogeneous computing literature [61, 68]. We scope out that objective as future work.

4 Evaluation

This evaluation answers the following questions:

- Given the vast configuration space offered by OSES, how quickly can Wayfinder converge on a specialized configuration for a given application? For a given iterations/time budget, what is the best configuration that can be found by Wayfinder? How do these results compare to baselines and competitors? (§4.1)
- When Wayfinder pre-trains a model on an application, what are the benefits of applying transfer learning and using that model to speed up the search of a specialized configuration for other applications? (§4.2)
- How good is Wayfinder at predicting crashes? (§4.3)
- Beyond Linux and performance, can Wayfinder be applied to different OSES and other metrics? (§4.4)

We select a set of popular applications: the Nginx [64] web server and the Redis [74] key-value store (both network-intensive), the SQLite [85] database management system (storage-intensive), and the NAS Parallel Benchmarks [11] (NPB, CPU- and memory-intensive). Nginx is benchmarked with `wrk` [99], Redis with `redis-benchmark` [75], and in both cases we optimize to maximize throughput. SQLite is benchmarked with LevelDB’s SQLite3 benchmark [50], in which a high number of SQL INSERT operations are issued, and we aim to minimize the average execution time of each operation. We use the OpenMP version of NPB and select a mix of CPU- and memory-intensive programs: FT, MG, CG, IS (running the entire suite is too long) with size classes S, W, A, and B. In one run we execute all programs for each size class and aggregate the number of operations per second reported by each benchmark, which is the metric we aim to maximize. We run all experiments on a dual-socket server with 2 Intel Xeon E5-2697 v2 (2x24 cores at 2.70 GHz, 128 GB of RAM), running Debian 10, and configured for high and stable performance: `isolcpus` on core 0-1, `hyperthreading` and `ASLR` disabled, performance CPU governor. Redis and SQLite run on 1 core because of their single-threaded nature, while Nginx and NPB run on 16 cores. The server used exposes 2

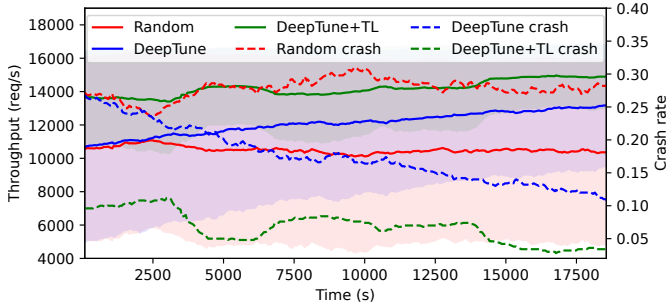
NUMA nodes, but we restrict the experiments to a single one to avoid NUMA skewing performance measurements. To avoid any disturbance due to experiment co-location, all test configurations are benchmarked one after the other and there is no experiment co-location.

Unless otherwise stated (§4.2), with Wayfinder transfer learning is disabled and each search process starts from scratch with a blank model. As a baseline we select random search [100], an approach well-known to give good results on large exploration spaces, such as the ones we target. This method explores the design space by continuously generating unique configurations with random values for each parameter. We omit comparing with grid search, which is well-known to be inferior to random search on the large configuration spaces we target. We also show that Wayfinder performs better than Bayesian optimization [79], a common technique used to optimize long-running black-box functions, which uses a probabilistic model to balance trying out previously unexplored parameters and focusing on parameters known to impact performance. We also demonstrate Wayfinder’s superiority to Unicorn [38], a closely related work focusing on the optimization of OS and application configurations using causal inference. As we demonstrate, Bayesian optimization and Unicorn do not scale to the large configuration spaces targeted by Wayfinder (e.g. Linux configuration), hence we compare to them on smaller exploration spaces. We also present how Wayfinder synergizes with compile-time, dynamic analysis based optimizers, like Cozart [43], a related work that leverages dynamic analysis to significantly reduce the number of Linux kernel configuration options, resulting in a much smaller footprint and exploration space. We show that configurations generated by Cozart are veritable baselines for Wayfinder to optimize upon through run-time options. Finally, for all experiments the initial OS configuration used to kickstart each search process is defined randomly.

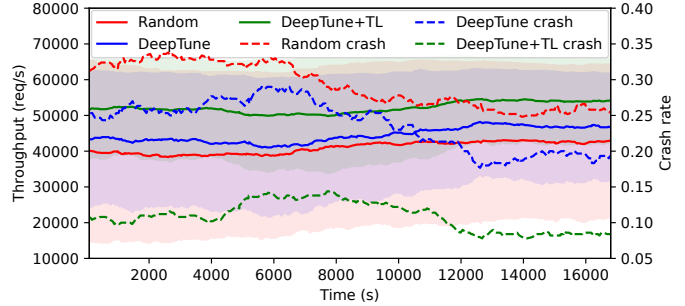
4.1 Performance of the Configuration-Space Search

OS Configuration Specialization. We run Wayfinder to specialize the Linux kernel towards performance, for each target application. We use Debian 10’s kernel v4.19 (a long-term support version). For these performance-based experiments, we configure Wayfinder to favor exploration of run-time parameters. We run both Wayfinder and random search for 250 iterations (250 parameters explored, which takes 3.5 to 5.2 hours depending on the application), and compare the performance of the configurations found.

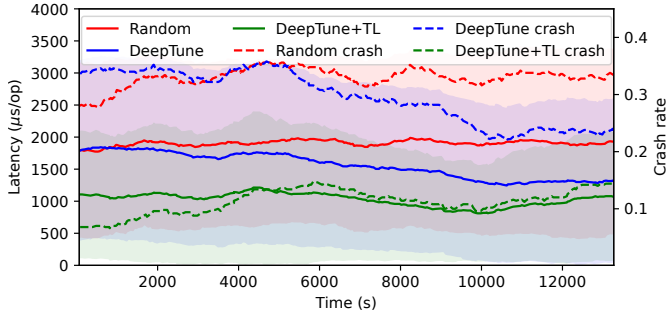
Table 2 shows the performance of the best configurations found by Wayfinder, along with the average time taken to find them and the performance of the default Lupine Linux [44] configuration. Lupine is a Linux kernel specialized for general purpose performance, i.e., not for any particular application. The configuration found for Nginx is 24% faster, showing that Wayfinder can successfully find configurations



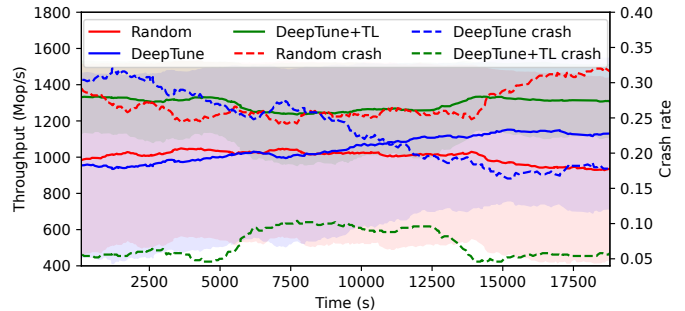
(a) Nginx (performance: throughput, **higher is better**).



(b) Redis (performance: throughput, **higher is better**).



(c) SQLite (performance: operation latency, **lower is better**).



(d) NPB (performance: Mop/s, **higher is better**).

Figure 6. Evolution of performance of configurations (solid lines, results of 5 runs smoothed for readability, with the area behind each curve representing the spread) found by a search session of 250 iterations for Nginx, Redis, SQLite, and NPB with Wayfinder, without and with transfer learning (TL, trained with Redis), and random search. Dashed lines show the crash rate.

that perform better than default. For NPB the improvement is only of 2%. Since the benchmark is mostly CPU- and memory-intensive and does not request any system functionality, the OS configuration has close to no impact on its performance. The best configuration found for SQLite also does not improve performance, which seems to indicate that the default configuration is already highly efficient for this scenario.

Figure 6 presents the performance of configurations found by each approach. Dashed lines represent the crash rate, with 1 corresponding to a crash every iteration. The performance of the configurations found by Wayfinder is at the beginning of the search process similar to that of random search. After a certain number of iterations, DEEPTUNE’s neural networks learn important parameters and how to use them efficiently, causing the configurations’ performance to outperform that of random. For example, for Nginx, after 250 iterations the smoothed throughput is more than 20% higher than that of random search. Unlike random’s relatively consistent crash behavior, Trailblazer’s crash rate decreases over time, as it learns to avoid configurations likely to crash: e.g., with Nginx the crash rate goes from 0.3 to 0.1 after 250 iterations.

Scaling to Large Configuration Spaces and Long Search Processes. Here we compare the scalability of Wayfinder with a competitor, Unicorn [38]. As Unicorn cannot scale to the size of Linux’s configuration, we create a synthetic

Table 2. Best-performing configurations found by Wayfinder applied to Linux v4.19 after 250 iterations.

App.	Lupine Linux	Wayfinder	Perf. unit	Relative Perf.	Avg. time to find	
					No TL	TL
Nginx	15731	19593	req/s	1.24x	415s	92s
Redis	58000	66118	req/s	1.14x	312s	69s
SQLite	284	284	μs/op	1x	248s	76s
NPB	1497	1522	Mop/s	1.02x	243s	76s

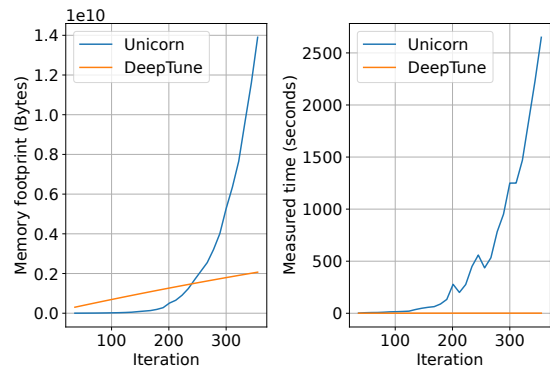


Figure 7. Evolution of the memory consumption (left) and algorithm execution time (right) of DEEPTUNE vs. Unicorn over a run of the search process.

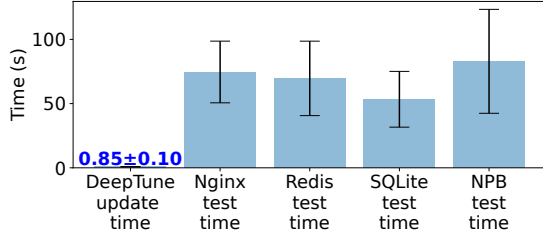


Figure 8. Average update time of DEEPTUNE vs. the average test time for different applications.

dataset with known local and global maximas to increase the likelihood of convergence. The dataset has a total number of parameters that match those used in the original Unicorn paper [38]. We measure the evolution of the execution time and peak memory consumption (using Python’s tracemalloc [72]) of each iteration for both algorithms. The results are presented on Figure 7. As one can observe, both the execution time and memory consumption of Unicorn increase exponentially while the algorithm iterate. These limitations come from Unicorn’s design choice of the causal analysis algorithm, which typically has $O(n^4)$ complexity, making it a poor fit for the large search spaces of modern OS configurations. In contrast, the complexity of DEEPTUNE grows linearly $O(n)$ in both time and memory consumption.

Due to these scalability issues, we are unable to compare to Unicorn in the other experiments presented in this evaluation. Indeed, these target design spaces which are several orders of magnitude larger than what Unicorn can handle.

Search Loop Execution Time. We measure the average execution time for evaluating a single configuration to understand Wayfinder’s performance. Figure 8 presents the execution time, broken down into the time spent running DEEPTUNE to decide what configuration to evaluate, and the time spent evaluating the configuration, i.e. booting the kernel, launching the target application, running the experiment, etc. Evaluating a configuration dominates the search process: it takes on average 60-80 s depending on the application, and varies widely due to the performance impact of the configurations evaluated. Conversely, the execution time of an iteration of DEEPTUNE takes less than a second, showing that the bottleneck is in the evaluation of configurations and not in the search algorithm.

High-Impact Configuration Parameters. We queried the models learned by DEEPTUNE to assess Wayfinder’s ability to identify parameters with the high impact on performance, including some that have been previously identified by experts. Here we focus on Nginx for space reasons, and because it is the target of many performance tuning guides.

Regarding top parameters impacting performance *positively*, while Trailblazer identifies parameters that are well-documented in tuning guides, it also uncovers other important configuration options that are not mentioned. For example, it identifies options such as the maximum number of connections that can be queued in the TCP/IP stack backlog per socket (`net.core.somaxconn`), the default size of sockets receive buffer (`net.core.rmem_default`), and the TCP keepalive time (`net.ipv4.tcp_keepalive_time`). These have been documented in Nginx/network performance tuning guides as high-impact parameters [37, 63, 78, 102]. However, Wayfinder also identifies parameters impacting performance in a less intuitive way, e.g. the frequency at which memory statistics such as those reported by `vmstat` are computed (`vm.stat_interval`). As mentioned in tuning guides [23], reducing it helps in latency sensitive scenarios. Hence, Wayfinder can automatically pinpoint and optimize for non-obvious parameters that have been identified by experts as impacting performance. Regarding top parameters *negatively* impacting performance, Wayfinder identifies several options leading to significant performance degradation, for example setting high levels of kernel verbosity (`printk`), delaying kernel logs (`printk_delay`) and enabling block I/O debugging (`vm.block_dump`). Logging and debugging are well-known to impact Nginx’s performance [39], and once again this demonstrates Wayfinder’s ability to identify documented bottlenecks.

4.2 Transfer Learning Efficiency

To assess the efficiency of transfer learning, we trained a model with DEEPTUNE on Redis for 250 iterations (taking 4.6 hours), and evaluated that model’s efficacy at finding specialized configurations for the other 3 applications considered in this evaluation.

The results are presented on Figure 6, with the curves labeled “TL” in the legend showing the performance of the configuration found and the crash rate of the transferred model. Using transfer learning, the configuration performance is consistently higher compared to starting the exploration process with an untrained model: for example, the first configuration found at the beginning of the process for Nginx has 1.33× higher performance with transfer learning than without transfer learning or with random search. The crash rate is also generally much lower with transfer learning, being below 10% in most cases. Table 2 takes another glance at the efficiency of transfer learning. It presents, in its last two columns, the time taken to reach a specialized configuration with and without transfer learning. The time savings brought by the technique are important: the search is sped up by a factor between 4.5× (Nginx) and 3.2× (NPB). Please note that, as the curves presented on Figure 6 are averages of multiple runs, it is unsurprising that the curve for DEEPTUNE + TL does not start exactly where the curve for DEEPTUNE without transfer learning ended in Figure 6b.

Table 3. Base prediction accuracy (1 being 100% accuracy) of DEEPTUNE. *Failure acc.* and *Run acc.* shows the prediction accuracy for a failure and an application running (non-failure) events. MAE is the normalized mean absolute error.

Application	Failure accuracy	Run accuracy	Performance prediction normalized MAE
Nginx	0.796	0.397	0.273
Redis	0.789	0.310	0.361
SQLite	0.742	0.456	0.112
NPB	0.755	0.455	0.359

4.3 Crash Prediction

Using random exploration, a high number of configurations (about one third) fail at runtime, which is a significant waste of time. Wayfinder also faces failures, however its ability to learn the configuration parameters that are likely to trigger crashes means that it can avoid many of them.

Table 3 presents the accuracy of DEEPTUNE when predicting that a given configuration will fail (failure accuracy) and when predicting that a given configuration will execute successfully (run accuracy). It also presents the normalized mean absolute error when predicting the performance of a configuration. As one can observe, the failure accuracy is high, being between 75% and 80%, allowing Wayfinder to avoid a large number of crashes. The run accuracy is low and ranges between 0% to 36%, hence we rely on failure accuracy to estimate the probability of a configuration failing, to determine if it is worth or not to evaluate that configuration.

4.4 Varying OSES and Target Metrics

Application to Other OSES: Unikraft. To demonstrate Wayfinder’s application to OSES other than Linux, we now apply Wayfinder to the Unikraft [42] library OS. We compile an Nginx Unikraft image and optimize its configuration to maximize request throughput using Wayfinder. Unikraft exposes a configuration space that is much smaller than that of Linux, making it possible to compare with Bayesian optimization: we explore 33 configuration parameters (10 Nginx application-level parameters, and 23 Unikraft OS parameters), yielding a search space of 3.7×10^{13} permutations.

We launch the exploration process with a time budget of 3 hours. The performance of the configurations found is presented in Figure 9. As one can observe, Wayfinder quickly converges on a specialized configuration, reached after 100 minutes. Bayesian optimization takes more than 160 minutes to reach configurations that perform similarly. With that time budget, we also observe that random search is not able to find high-performance configurations. Figure 9 clearly displays three phases in the behavior of DEEPTUNE. DEEPTUNE first slowly uncovers configurations with better performance. After about 25 minutes it picks up an impactful set of parameters and enters an exploitation phase where it focuses on the parameters to rapidly increase performance.

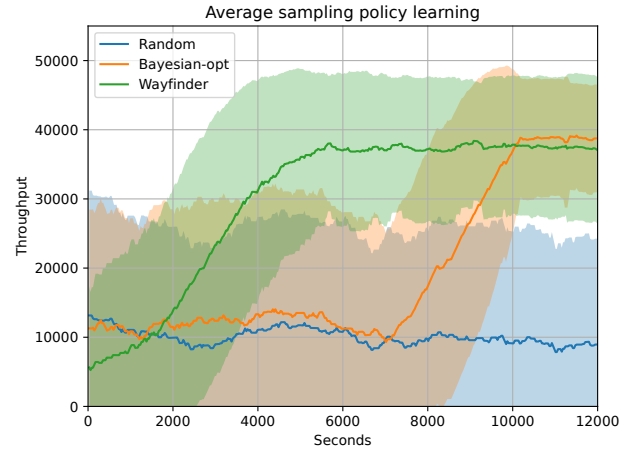


Figure 9. Evolution of the performance of configurations found by Wayfinder, random search, and Bayesian optimization for Nginx running on the Unikraft unikernel. Results of 5 runs, smoothed for readability.

This phase ends after about 100 minutes, at which points DEEPTUNE goes back to exploration.

Note that the performance improvement brought by top configurations is significantly higher than that observed for Linux (see Figure 6). This can be explained by the fact that Unikraft is a unikernel [54] offering low-latency user/kernel transitions which, under the right configuration, speed up latency-sensitive system intensive workloads significantly [42].

Application to Other Metrics: Memory Footprint. To assess Wayfinder’s suitability to optimize beyond performance, we measure its efficiency to find Linux kernel configurations with small memory footprint. This metric has been the target of several past works, since it is crucial in certain domains, such as lightweight virtualization [44, 73, 80] or embedded systems [1, 40], and also affect aspects such as security [45, 89]. To optimize for memory consumption we configure Wayfinder to build images of RISC-V Linux, an ISA that is highly popular in embedded systems. The memory consumption is measured by booting the images in a QEMU emulated setup (although emulation affects performance, it does not impact memory consumption). In line with past studies on that topic [1, 40, 44, 45, 89], in this experiment we configure Wayfinder to favor varying compile-time options (over run-time options as was the case in §4.1). We launch the exploration process with Wayfinder and random search, for a time budget of 3 hours.

The evolution of the memory footprint of the configurations found is presented on Figure 10. The default configuration has a 210 MB memory footprint. After 3 hours, Wayfinder finds a configuration having a memory footprint of 192 MB (smoothed), which is an 8.5% reduction with respect to the default one. At the end of the exploration process, the memory footprint of the configuration found by random

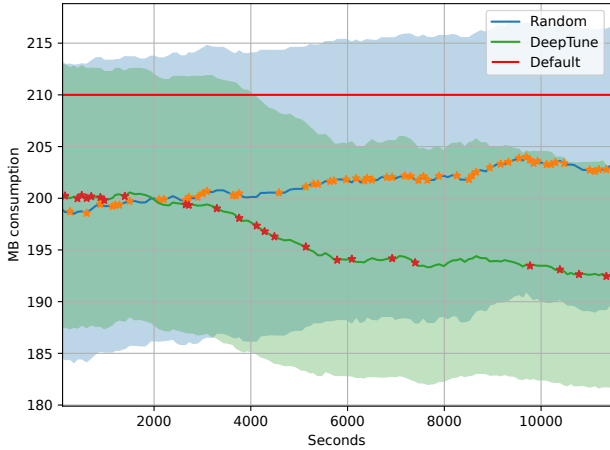


Figure 10. Memory footprint of Linux images built from configurations found during a 3-hour search session by Wayfinder and random search. Results of 5 runs, smoothed for readability. Crashes are indicated with stars.

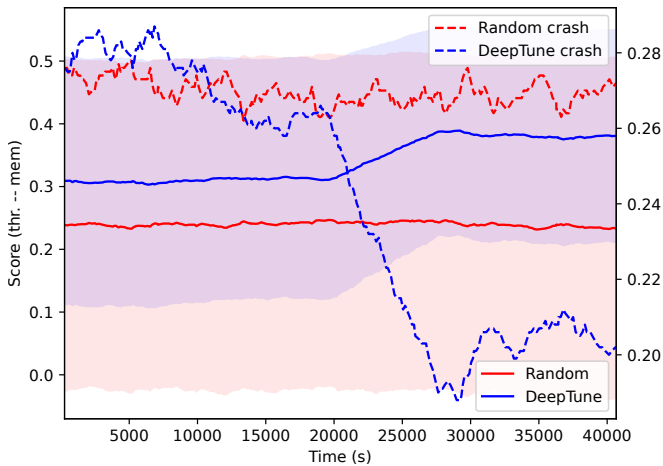


Figure 11. Evolution of a learning policy that co-optimizes throughput and memory consumption on top of Cozart. Higher is better. Application under test is Nginx. Smoothed for readability. The crash rate indicated on a separate axis.

search is 203 MB (smoothed), i.e. a reduction of 5.5%. Further, one can notice that the number of failing configurations is much smaller with Wayfinder than with random search. This is due to the system’s capacity to predict crashes, which is particularly efficient for that experiment: only four crashes happen in the last 100 minutes.

Synergy with Compile-Time Optimizers. Using the methods discussed earlier, Wayfinder can also examine compile-time parameters. However, this process can be inefficient due to how the Linux kernel manages build dependencies and how Wayfinder explores configurations. Initially, Wayfinder

generates nearly random configurations, which often requires complete rebuilds of the Linux kernel. Even a single incorrect parameter can cause the build to fail. To address this issue, we added an initial optimization step using Cozart [43], which uses dynamic analysis to significantly reduce the number of unused components in the Linux kernel. This reduction results in a much smaller configuration space, allowing Wayfinder to focus on the unexplored runtime parameters of Linux while also providing a performance boost. We observed a 31% increase in throughput compared to the baseline, along with a slight decrease in memory usage, similar to what was reported in the Cozart evaluation.

Next, we applied Wayfinder optimizations on top of the Cozart baseline. We define an optimization score s as follows:

$$s = \text{mXNorm}(t) - \text{mXNorm}(m), \quad (4)$$

Here, t represents throughput, m represents memory consumption, and $\text{mXNorm}(x)$ is the min-max normalization function, which brings throughput and memory numbers to a common scale. A higher score thus indicates better throughput and less memory usage.

Figure 11 shows that Wayfinder can effectively learn and create a policy that performs better than random search when run on top of a Cozart baseline. Similarly to Figure 9, this figure shows the exploration vs. exploitation strategies of DEEPTUNE. After about 300 minutes, DEEPTUNE finds a set of parameters that allows it to increase the throughput-memory score. It then exploits it for about 100 minutes. This manifests in a significantly lower crash rate since DEEPTUNE focuses on a more stable part of the space. DEEPTUNE switches back to exploration after this phase. This results in a higher crash rate as it explores more unknown configurations.

Table 4 presents the top five scores obtained during the joint exploration of throughput and memory usage. The first and second permutations, despite having significantly lower throughput, utilize less memory, leading to the highest overall scores. Conversely, the third permutation achieves much higher throughput with only a slight increase in memory usage. Compared to the Cozart baseline, these permutations consistently deliver higher throughput while maintaining lower memory consumption.

In conclusion, we demonstrate that by combining the Cozart compilation optimizations with the runtime optimizations from Wayfinder, the operating system can be adjusted to improve performance across various metrics.

5 Related Works

Configuration-based OS Specialization. Several works specialize Linux through its configuration to lower resource usage [1, 40, 44, 73, 77, 80] or attack surface [45, 89]. Some approaches are manual [44, 73, 80] and cannot scale to many applications, while others are automated. Undertaker [45, 89] uses dynamic analysis to trace kernel code executed by a workload, and correlates that code to compile-time options

Table 4. Top-5 best results found for the Throughput-Memory experiment (Figure 11) on top of Cozart. Note that these numbers are obtained with the same setup as the Cozart paper’s [43], and thus cannot be compared with Table 2 (different kernel versions, four CPU cores instead of one).

Rank	Score	Memory (mB)	Throughput (req./s)
1	0.84	327.72	47002
2	0.82	328.57	47215
3	0.81	329.67	49375
4	0.79	329.75	48606
5	0.78	330.46	49375
Cozart	–	331.77	46855

to compile out as much unneeded code as possible. Kernel tailoring [40] builds on top of Undertaker. Noting that a large part of the configurations explored by the tool fail at runtime, the work proposes a search method that tries to derive valid configurations from configurations known to execute successfully. These approaches target attack surface/memory footprint reduction, so they aim to determine, for a given workload, which compile-time configuration parameters can be switched off, and which must be on. These approaches cannot be directly applied to our problem due to our focus on performance (in addition to other metrics such as resource usage). Beyond turning only compile-time features on and off, Wayfinder considers the full set of kernel compile-time, boot-time, and run-time options, many of them taking arbitrary values, with undocumented ranges of validity. Cozart [43] also builds on top of Undertaker. With a similar approach that uses dynamic analysis to determine unused configuration options, Cozart reduces the footprint of the Linux kernel, with the added bonus of decreasing memory usage and increasing performance. These performance gains are not the main focus of Cozart though, but they do offer a better baseline for Wayfinder to start from. As mentioned previously, Wayfinder and Cozart do not have much in common, but they synergize very well when it comes to optimizing the Linux kernel both through compile-time options and through run-time ones. AutoS [20] specializes OS kernels (including Linux) for AIoT application performance, by expressing their configuration as a tree and passing it to a large language model driving the exploration with the help of prompts describing the environment and refining the search. This work is limited to compile-time options and requires expertise in order to write prompts.

Configuration-based Application Optimization. Unicorn [38] uses causal inference to automatically specialize configurations for embedded software. KML [3, 4] uses machine learning techniques to optimize the configuration of storage subsystems. Some past works have explored the performance impact of software configuration parameters using static and/or dynamic analysis [51, 94, 95]. Others [18]

focus on identifying the configuration parameters most relevant to performance for a certain workload, in this case in a storage subsystem. All these systems require expert knowledge to pre-identify sets of potentially important parameters, and target configuration spaces that are much smaller than Wayfinder’s. In particular, we have shown (Fig. 7) that Unicorn does not scale to such large configuration spaces.

In serverless computing, AWS Lambda Power Tuning [7, 19] is an automated platform to optimize the memory allocated to functions. Sizeless [24] proposes a regression model to estimate function performance. COSE [5] uses Bayesian optimization to find specialized configurations for functions. In databases, several works such as Otter-tune [93] also automatically optimize for performance. Such application-specific approaches present a configuration space that is orders of magnitude smaller than that of modern OSes. Hence, they rely on techniques such as Bayesian optimization that do not fit our problem.

Configuration-based optimization is used in other contexts including hardware design (accelerators tuning [62, 101], FPGA floor planning [59]), neural architecture search [97], and distributed computation placement [60]. These methods are tied to the application domain they target (e.g. they operate on a specific representation of the search space [97]) and are not directly applicable to OS specialization.

Other Optimization Works in the ML Literature. Parameter search/optimization has attracted significant attention in recent years [6, 27, 29, 34, 35, 38, 52, 71, 84]. Some approaches rely on Bayesian optimization. SMAC [52], for example, enables to easily apply Bayesian optimization to generic optimization problems. SMAC also supports random forests, which improve on scalability and performance prediction compared to a Gaussian process, but offer poor estimations of predicted posterior uncertainty (needed for exploration vs. exploitation decisions). Snoek et al. [84] contribute Bayesian Neural Networks (BNN) to replace Gaussian processes. BNNs are general approximators that can predict the posterior uncertainty, but their fitting and generalization capabilities are typically lower than conventional NNs. BNNs are also expensive to evaluate, since they require a Monte Carlo sampling strategy that requires to run them multiple times. Hase et al. [35] successfully apply this idea in chemistry, and extend the idea to multi-objective optimization [34]. Gal et al. [30] sample from dropout as an approximation of BNNs. Finally, Kendall et al. [41] study several kinds of uncertainties required for ML problems, and propose an uncertainty estimation mechanism for regression problems. Due to the nature of the conventionally used activation functions, they typically cannot provide a good uncertainty of outlier samples. DEEPTUNE proposes a new NN design that enables accurate prediction of system performance, failure, and accuracy in a single pass with high

precision. We also propose a new scoring function that creates a balanced permutation policy picking.

6 Conclusion

We proposed Wayfinder, a fully automated approach at OS specialization through configuration. Wayfinder ships with a benchmarking platform, and DEEPTUNE, a neural-network-based optimization algorithm which predicts the validity and performance of configurations. Wayfinder does not require expert knowledge, and scales to the vast configuration space of modern OSes such as Linux. After running Wayfinder to optimize for one application, the trained model can be reused to speed up the specialization for other similar applications. Wayfinder can identify specialized configurations for various metrics, with up to 24% application performance increase and 8.5% memory usage decrease vs. default configurations, converging faster on good solutions than competing approaches using random search and Bayesian optimization.

Acknowledgments

We thank the anonymous reviewers for their comments and insights. Special thanks go to Margo Seltzer for her precious comments that significantly improved the paper's quality. This work was funded in part by Innovate UK grant 10164504 (MicROS), and the UK Engineering and Physical Sciences Research Council grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), and EP/X015610/1 (FlexCap). We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Nous remercions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de son soutien.

A Artifact Appendix

A.1 Abstract

This artifact contains the source code of Wayfinder, the operating system specialization framework. It also contains scripts to reproduce the results found in the paper, and/or the dataset used for generating the figures/tables in the paper. The goal is to allow the reader to reproduce our experiments, and to give insight into the capabilities and usage of Wayfinder.

A.2 Description & Requirements

A.2.1 How to access. The latest version of the artifact can be found on GitHub². Alternatively, individual releases can be downloaded from our Zenodo archive³. Note that the artifact evaluation (AE) GitHub repository only contains part of the artifact, namely scripts to reproduce this paper's experiments. The Wayfinder framework code, tools, and other examples are available in the Wayfinder⁴ repository.

²<https://github.com/unikraft/wayfinder-eurosys26-ae>

³<https://doi.org/10.5281/zenodo.18598079>

⁴<https://github.com/unikraft/wayfinder> and <https://doi.org/10.5281/zenodo.18592520>

In order to precisely reproduce this paper's measurements, we gave EuroSys'26 AEC reviewers access to our server, an Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz with 315 GB RAM, Ubuntu 24.04, and Linux version 6.8.0-53-generic. Access to this particular setup is not required to run the artifact. Hardware and software dependencies are detailed below.

A.2.2 Hardware dependencies. An Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60 GHz with at least 64 GB RAM, typically any processor which has a dual-socket mount. The processor must have at least 8 cores. 64 GB of RAM are necessary to run the experiments corresponding to Figure 7, as the experiment tests RAM usage. Note that this amount of cores/RAM is required to reproduce this paper's results, not to run Wayfinder. At least 100 GB of disk space is needed.

A.2.3 Software dependencies. This artifact has been tested with Ubuntu 24.04 (Noble Numbat), with Linux kernel version 6.8.0-53-generic (KVM enabled), Docker version 28.1.1 (or any recent version), QEMU 5.2, Libvirt 7.4. Other dependencies use containers which are already versioned by Wayfinder and the artifact scripts.

A.2.4 Benchmarks. All data sets are included in the artifact, either generated automatically by scripts, or pre-generated.

A.3 Set-up

Before running any experiment, prepare your host with the recommendations detailed above in A.2.3. Once the system is set up, clone the AE repository:

```
$ git clone \  
https://github.com/unikraft/wayfinder-eurosys26-ae
```

Then follow the *README.md* step-by-step instructions to download, compile, and configure Wayfinder from its repository:

```
$ git clone https://github.com/unikraft/wayfinder
```

Finally, after everything is configured, start Wayfinder and its helper containers:

```
$ docker-compose up -d registry influxdb postgres  
redis  
$ sudo ./scripts/wayfinder.sh
```

A.4 Evaluation workflow⁵

Experiments should be run sequentially. Every experiment uses a different builder image to generate kernels to test. These images need to be built before running the experiment, for example:

⁵Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at <https://sysartifacts.github.io/eurosys2026/>.

```
$ docker build -t
  localhost:5000/linux-nginx:latest -f
  Dockerfile.jessie .
$ docker push localhost:5000/linux-nginx:latest
```

The preparation of each experiment is a mandatory step that consists in building the kernels that will be tested by the platform. Each kernel build operation only takes minutes and should be done while no other experiment is running on the system. Once one of the experiments has been prepared, it can be run with a syntax similar to this one:

```
$ ~/wayfinder/dist/wfctl create ./job.yaml
$ ~/wayfinder/dist/wfctl start --isol-level full
  --isol-split both -l 1 -s random "$ID"
```

Running all experiments takes on average days on our setup. This is because we collect thousands of permutations to showcase the evolution. For the experiments that interact with Wayfinder, consider decreasing the number of permutations and repetitions. For the rest, we provide pre-generated data-sets that can be used to quickly generate the figure.

A.4.1 Major Claims.

- (C1): *Wayfinder fully automatically identifies specialized configurations with up to 24% application performance improvement and 8.5% memory usage reduction compared to default configurations. This is proven by the experiments described in Sections 4.1 and 4.4 whose results are illustrated in Figure 6 and Figure 10.*
- (C2): *We highlight the benefits of our neural network, reaching good solutions faster than competing approaches (random and Bayesian), and successfully transferring knowledge between related applications. This is proven by the experiments described in Section 4.4 whose results are illustrated in Figure 9 and Figure 11.*

A.4.2 Experiments. Experiments presented and structured in the artifact repository mentioned in Section A.3 each have a *README.md* file and are structured suggestively in directories of the formats *figure-** and *table-**. These files briefly describe the experiments and the relevant preparation and execution steps: it is strongly recommended to go through them at least once. Experiments which do not have a time estimate take minutes. The total time to run the experiments in the repository should be around 10-12 hours, but can take up to 32 hours if repetitions are done. Out of this, in both cases, only 2-3 hours represent manual work. The Wayfinder framework takes care to run jobs in the background.

Reproducing experiments on the same machine should produce results similar as in the paper. On other machines, we expect different absolute numbers, but similar trends and ordering. Because of the non-deterministic nature of the exploration process, repeated measurements are subject to

some variation, but the general trends and averages of multiple executions should be consistent with what is presented in the paper. Figures 1, 2 and Table 1, are reproducible manually. Due to the very long exploration time (days) required to produce the data behind them, Figures 5, 6, 7, 8, 9, 10, 11 are reproducible from existing datasets. Finally, the measurements in the data use the following metrics: *requests/s*, *ms/op*, *Mop/s*, *seconds*, *MB*.

A.5 Notes on Reusability

Reviewers may use the examples in the Wayfinder repository to create their own job based on a custom application and its accompanying benchmarking tool. Instructions to build the Wayfinder developer Docker image, run generic jobs, and build custom images are available in the *README.md* file at the root of our AE repository⁶.

A.6 General Notes

All experiments have a *README.md* file detailing the manual steps that are needed to obtain the corresponding figures. The folders for Figure 1, Table 1, and Figure 2 contain the step-by-step guide to generate them from scratch. The other experiments contain the data sets obtained and scripts to generate the Figures/Tables in question. Most experiments took at least one day to run each, so it is recommended to inspect and adapt the scripts provided to lower the number of iterations. We strongly recommend carefully reading the instructions in each experiment before starting to reproduce experiments.

References

- [1] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Olivier Barais. 2019. *Learning very large configuration spaces: What matters for Linux kernel sizes*. Ph.D. Dissertation. Inria Rennes-Bretagne Atlantique.
- [2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Pivonka, and Diana-Maria Popa. 2020. *Firecracker: Lightweight Virtualization for Serverless Applications*. In *NSDI*, Vol. 20. 419–434.
- [3] Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. 2023. *Improving Storage Systems Using Machine Learning*. *ACM Transactions on Storage* 19, 1 (2023), 1–30.
- [4] Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. 2021. *A machine learning framework to improve storage system performance*. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*. 94–102.
- [5] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. *Cose: Configuring serverless functions using statistical learning*. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 129–138.
- [6] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. *Optuna: A next-generation hyperparameter*

⁶<https://github.com/unikraft/wayfinder-eurosys26-ae/blob/main/README.md>

- optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2623–2631.
- [7] Amazon Web Services. 2026. Profiling functions with AWS Lambda Power Tuning. <https://docs.aws.amazon.com/lambda/latest/operatorguide/profile-functions.html> [Accessed Feb. 11, 2026].
- [8] Thomas E Anderson. 1993. *The case for application-specific operating systems*. University of California, Berkeley, Computer Science Division.
- [9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’keeffe, Mark Stillwell, et al. 2016. Scone: Secure linux containers with Intel SGX. In *OSDI*, Vol. 16. 689–703.
- [10] Risan Bagja Pradana. 2017. Optimized Nginx Configuration. <https://github.com/risa/nginx-config> [Accessed Feb. 11, 2026].
- [11] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0*. Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.
- [12] Harry G. Barrow, Jay M. Tenenbaum, Robert C. Bolles, and Helen C. Wolf. 1977. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI’77, Vol. 2)*.
- [13] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanian. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP’09)*. ACM, 29–44. doi:10.1145/1629575.1629579
- [14] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R Lorch, Barry Bond, Reuben Olinsky, and Galen C Hunt. 2013. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 239–252.
- [15] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI’14)*. USENIX, 49–65.
- [16] Vincent Bernat. 2011. Tuning Linux IPv4 route cache. <https://vincent.bernat.ch/en/blog/2011-ipv4-route-cache-linux> [Accessed Feb. 11, 2026].
- [17] Leo Breiman. 2001. Random Forests. *Machine Learning* (2001).
- [18] Zhen Cao, Geoff Kuenning, and Erez Zadok. 2020. Carver: Finding important parameters for storage system tuning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 43–57.
- [19] Alex Casalboni. 2023. AWS Lambda Power Tuning Github Repository. <https://github.com/alexcasalboni/aws-lambda-power-tuning> [Accessed Feb. 11, 2026].
- [20] Huilai Chen, Yuanbo Wen, Limin Cheng, Shouxu Kuang, Yumeng Liu, Weijia Li, Ling Li, Rui Zhang, Xinkai Song, Wei Li, et al. 2024. (Poster) AutoOS: Make Your OS More Powerful by Exploiting Large Language Models. In *Forty-first International Conference on Machine Learning*.
- [21] Bob Cromwell. 2023. Performance Tuning on Linux — TCP. <https://cromwell-intl.com/open-source/performance-tuning/tcp.html> [Accessed Feb. 11, 2026].
- [22] Abhishek Dubey. 2019. Redis Best Practices and Performance Tuning. <https://iamabhishek-dubey.medium.com/redis-best-practices-and-performance-tuning-c48611388bbc> [Accessed Feb. 11, 2026].
- [23] Jeremy Eder. 2015. Low Latency Performance Tuning for Red Hat Enterprise Linux 7. <https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v1.1.pdf> [Accessed Feb. 11, 2026].
- [24] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*. 248–259.
- [25] Dawson R. Engler, M. Frans Kaashoek, and James O’Toole Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP’95)*. ACM, 251–266. doi:10.1145/224056.224076
- [26] Haoqiang Fan, Hao Su, and Leonidas J. Guibas. 2017. A Point Set Generation Network for 3D Object Reconstruction From a Single Image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [27] Matthias Feurer, Aaron Klein, Katharina Eggenberger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. *Advances in neural information processing systems* 28 (2015).
- [28] Mike Freemon. 2022. Optimizing TCP for high WAN throughput while preserving low latency. <https://blog.cloudflare.com/optimizing-tcp-for-high-throughput-and-low-latency/> [Accessed Feb. 11, 2026].
- [29] Pascal Friederich, Florian Häse, Jonny Proppe, and Alán Aspuru-Guzik. 2021. Machine-learned potentials for next-generation matter simulations. *Nature Materials* (2021).
- [30] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*.
- [31] Eduardo C. Garrido-Merchán and Daniel Hernández-Lobato. 2020. Dealing with categorical and integer-valued variables in Bayesian Optimization with Gaussian processes. *Neurocomputing* 380 (2020), 20–35. doi:10.1016/j.neucom.2019.11.004
- [32] Google. 2026. Google Syszcaller: Syscall description language. https://github.com/google/syszcaller/blob/master/docs/syscall_descriptions_syntax.md [Accessed Feb. 11, 2026].
- [33] Per Brinch Hansen. 1970. The nucleus of a multiprogramming system. *Commun. ACM* 13, 4 (1970), 238–241.
- [34] Florian Häse, Loïc M Roch, and Alán Aspuru-Guzik. 2018. Chimera: enabling hierarchy based multi-objective optimization for self-driving laboratories. *Chemical science* (2018).
- [35] Florian Häse, Loïc M. Roch, Christoph Kreisbeck, and Alán Aspuru-Guzik. 2018. Phoencis: A Bayesian Optimizer for Chemistry. *ACS Central Science* (2018).
- [36] Asmaul Hosna, Ethel Merry, Jigmey Gyalmo, Zulfikar Alom, Zeyar Aung, and Mohammad Abdul Azim. 2022. Transfer learning: a friendly introduction. *Journal of Big Data* 9, 1 (2022), 102.
- [37] IBM. 2021. Tuning your Linux system for more efficient parallel job performance. <https://www.ibm.com/docs/en/smpi/10.2?topic=mpi-tuning-your-linux-system> [Accessed Feb. 11, 2026].
- [38] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2022. Unicorn: Reasoning about Configurable System Performance through the Lens of Causality. In *Proceedings of the Seventeenth European Conference on Computer Systems (Rennes, France) (EuroSys ’22)*. Association for Computing Machinery, New York, NY, USA, 199–217. doi:10.1145/3492321.3519575
- [39] Alexander Jung, Hugo Lefeuvre, Charalampos Rotsos, Pierre Olivier, Daniel Oñoro-Rubio, Felipe Huici, and Mathias Niepert. 2021. Wayfinder: towards automatically deriving optimal OS configurations. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. 115–122.
- [40] Junghwan Kang. 2017. A practical approach of tailoring Linux kernel. *The Linux Foundation Open Source Summit North America, Los Angeles, CA* (2017). https://static.sched.com/hosted_files/ossna2017/97/Final_OSS_NA17_A_Practical_approach_of_tailoring_Linux_kernel.pdf [Accessed Feb. 11, 2026].

- [41] Alex Kendall and Yarin Gal. 2017. What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/2650d6089a6d640c5e85b2b88265dc2b-Paper.pdf
- [42] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gauthier Gain, Cyril Soldani, Costin Lupu, Ștefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. *Sixteenth European Conference on Computer Systems*. doi:10.1145/3447786.3456248
- [43] Hsuan-Chi Kuo, Jianyan Chen, Sibir Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 1, Article 03 (May 2020), 27 pages. doi:10.1145/3379469
- [44] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibir Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the 15th European Conference on Computer Systems* (Heraklion, Greece) (*EuroSys '20*). Association for Computing Machinery, New York, NY, USA, 15 pages. doi:10.1145/3342195.3387526
- [45] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *NDSS*.
- [46] Butler W Lampson and Robert F Sproull. 1979. An open operating system for a single-user machine. In *Proceedings of the seventh ACM symposium on Operating systems principles*. 98–105.
- [47] Chien-Cheng Lee, Pau-Choo Chung, Jea-Rong Tsai, and Chein-I Chang. 1999. Robust radial basis function neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 29, 6 (1999), 674–685. doi:10.1109/3477.809023
- [48] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Ștefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. 2021. Flexos: Making os isolation flexible. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 79–87.
- [49] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Ștefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) (*ASPLOS '22*). Association for Computing Machinery, New York, NY, USA, 467–482. doi:10.1145/3503222.3507759
- [50] LevelDB Contributors. 2020. LevelDB's SQLite3 Benchmark. https://github.com/google/leveldb/blob/main/benchmarks/db_bench_sqlite3.cc [Accessed Feb. 11, 2026].
- [51] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
- [52] Marius Lindauer, Katharina Eggenberger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research* (2022).
- [53] Linux Contributors. 2023. The kernel's command-line parameters. <https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html>.
- [54] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 461–472.
- [55] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 218–233.
- [56] Anqi Mao, Mehryar Mohri, and Yutao Zhong. 2023. Cross-entropy loss functions: theoretical analysis and applications. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) (*ICML '23*). JMLR.org, Article 992, 26 pages.
- [57] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. *SIGCOMM Comput. Commun. Rev.* 44, 4 (aug 2014), 175–186. doi:10.1145/2740070.2626311
- [58] Patrick Meenan. 2018. Optimizing HTTP/2 prioritization with BBR and tcp_notsent_lowat. <https://blog.cloudflare.com/http-2-prioritization-with-nginx/> [Accessed Feb. 11, 2026].
- [59] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. 2021. A graph placement methodology for fast chip design. *Nature* 594, 7862 (2021), 207–212.
- [60] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *International conference on machine learning*. PMLR, 2430–2439.
- [61] Sparsh Mittal. 2016. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–38.
- [62] Luigi Nardi, David Koepf, and Kunle Olukotun. 2019. Practical design space exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 347–358.
- [63] Rick Nelson. 2014. Tuning NGINX for Performance. <https://www.f5.com/company/blog/nginx/tuning-nginx> [Accessed Feb. 11, 2026].
- [64] Nginx Contributors. 2023. Nginx Website. <https://nginx.org/> [Accessed Feb. 11, 2026].
- [65] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. 2020. LibretOS: A dynamically adaptable multiserver-library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 114–128.
- [66] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE) (VEE'19)*. ACM, 59–73.
- [67] Pierre Olivier, Hugo Lefeuvre, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2021. A syscall-level binary-compatible unikernel. *IEEE Trans. Comput.* 71, 9 (2021), 2116–2127.
- [68] Pierre Olivier, A. K. M. Fazla Mehrab, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, and Binoy Ravindran. 2019. HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost, Low-Power Embedded Systems. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) (*HPDC '19*). Association for Computing Machinery, New York, NY, USA, 85–96. doi:10.1145/3307681.3325408
- [69] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*. 238–253.
- [70] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Transactions on Computer Systems* 33, 4 (November 2015), 1–30. doi:10.1145/2812806
- [71] Robert Pollice, Gabriel dos Passos Gomes, Matteo Aldeghi, Riley J Hickman, Mario Krenn, Cyrille Lavigne, Michael Lindner-D'Addario,

- AkshatKumar Nigam, Cher Tian Ser, Zhenpeng Yao, et al. 2021. Data-driven strategies for accelerated materials design. *Accounts of Chemical Research* (2021).
- [72] Python Contributors. 2023. tracemalloc - Trace memory allocations. <https://docs.python.org/3/library/tracemalloc.html> [Accessed Feb. 11, 2026].
- [73] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristol De Oliveira, Larry Woodman, Renato Mancuso, et al. 2023. Unikernel Linux (UKL). In *Proceedings of the Eighteenth European Conference on Computer Systems*. 590–605.
- [74] Redis Contributors. 2023. Redis Website. <https://redis.io/> [Accessed Feb. 11, 2026].
- [75] Redis Contributors. 2025. Redis benchmark utility. <https://github.com/redis/redis/blob/unstable/src/redis-benchmark.c> [Accessed Feb. 11, 2026].
- [76] Mark Richards. 2021. Extreme HTTP Performance Tuning: 1.2M API req/s on a 4 vCPU EC2 Instance. <https://talawah.io/blog/extreme-http-performance-tuning-one-point-two-million/> [Accessed Feb. 11, 2026].
- [77] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. 2014. Automatic feature selection in large-scale system-software product lines. *ACM SIGPLAN Notices* 50, 3 (2014), 39–48.
- [78] Emily Serverwise. 2022. Maximizing NGINX Performance: A Comprehensive Guide to Tuning the Backlog and net.core.somaxconn Parameters. <https://www.getpagespeed.com/server-setup/nginx/maximizing-nginx-performance-a-comprehensive-guide-to-tuning-the-backlog-and-net-core-somaxconn-parameters> [Accessed Feb. 11, 2026].
- [79] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.
- [80] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 121–135.
- [81] Mark Silberstein. 2017. OmniX: an accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 69–75.
- [82] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a file system with GPUs. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*. 485–498.
- [83] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. 2016. GPUnet: Networking abstractions for GPU programs. *ACM Transactions on Computer Systems (TOCS)* 34, 3 (2016), 1–31.
- [84] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, and Ryan P. Adams. 2015. Scalable Bayesian Optimization Using Deep Neural Networks. In *ICML*.
- [85] SQLite Contributors. 2023. SQLite Website. <https://www.sqlite.org/index.html> [Accessed Feb. 11, 2026].
- [86] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* (2014).
- [87] Timo Stark. 2022. Avoiding the Top 10 NGINX Configuration Mistakes. <https://www.f5.com/company/blog/nginx/avoiding-top-10-nginx-configuration-mistakes> [Accessed Feb. 11, 2026].
- [88] Jacob Swanson, Myra B. Cohen, Matthew B. Dwyer, Brady J. Garvin, and Justin Firestone. 2014. Beyond the Rainbow: Self-Adaptive Failure Avoidance in Configurable Systems. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) (FSE 2014). Association for Computing Machinery, New York, NY, USA, 377–388. doi:10.1145/2635868.2635915
- [89] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability. In *Eighth Workshop on Hot Topics in System Dependability (HotDep 12)*.
- [90] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. 2014. Cooperation and security isolation of library Oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.
- [91] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference*. 645–658.
- [92] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 39–50.
- [93] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.
- [94] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020. Configcrusher: Towards white-box performance analysis for configurable systems. *Automated Software Engineering* 27 (2020), 265–300.
- [95] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-box analysis over machine learning: Modeling performance of configurable systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1072–1084.
- [96] Yu Wang, Victor Lee, Gu-Yeon Wei, and David Brooks. 2019. Predicting New Workload or CPU Performance by Analyzing Public Datasets. *ACM Trans. Archit. Code Optim.* 15, 4, Article 53 (Jan. 2019), 21 pages. doi:10.1145/3284127
- [97] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadepta Dey, and Frank Hutter. 2023. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727* (2023).
- [98] Marcel Wienöbst, Max Bannach, and Maciej Liśkiewicz. 2021. Extendability of causal graphical models: Algorithms and computational complexity. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. PMLR.
- [99] wrk Contributors. 2021. wrk: Modern HTTP benchmarking tool. <https://github.com/wg/wrk> [Accessed Feb. 11, 2026].
- [100] Zelda B Zabinsky. 2009. Random search algorithms. *Department of Industrial and Systems Engineering, University of Washington, USA* (2009).
- [101] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 27–42.
- [102] Mike Zupan. 2014. Recommended TCP keepalive settings for a busy server. <https://serverfault.com/questions/641606/recommended-tcp-keepalive-settings-for-a-busy-server> [Accessed Feb. 11, 2026].