Interference-Aware Edge Runtime Prediction with Conformal Matrix Completion

Tianshu Huang [1] Arjun Ramesh [1] Emily Ruppel [2]
Nuno Pereira [3] Anthony Rowe [1 2] Carlee Joe-Wong [1]

Abstract

Accurately estimating workload runtime is a longstanding goal in computer systems, and plays a key role in efficient resource provisioning, latency minimization, and various other system management tasks. Runtime prediction is particularly important for managing increasingly complex distributed systems in which more sophisticated processing is pushed to the edge in search of better latency. Previous approaches for runtime prediction in edge systems suffer from poor data efficiency or require intensive instrumentation; these challenges are compounded in heterogeneous edge computing environments, where historical runtime data may be sparsely available and instrumentation is often challenging. Moreover, edge computing environments often feature multi-tenancy due to limited resources at the network edge, potentially leading to interference between workloads and further complicating the runtime prediction problem. Drawing from insights across machine learning and computer systems, we design a matrix factorization-inspired method that generates accurate interference-aware predictions with tight provably-guaranteed uncertainty bounds. We validate our method on a novel WebAssembly runtime dataset collected from 24 unique devices, achieving a prediction error of 5.2% – 2x better than a naive application of existing methods.

1 Introduction

Practitioners have always had to balance hardware capacity with expected software resource demands, for example, when specifying future hardware platforms or optimizing application placement across cloud datacenter servers based on predicted application runtimes (Amiri & Mohammad-Khanli, 2017). This task has only been made more difficult as improvements in compute and networking capabilities drive a paradigm shift pushing compute to *edge systems* closer to the physical world such as embedded computers and smart phones.

Edge systems present a unique set of challenges. Unlike cloud systems, edge systems are highly heterogeneous, involving compute platforms across a wide range of compute capabilities ranging from microcontrollers to edge servers. Edge systems are also highly resource constrained, and can suffer greatly from interference between workloads (Cavicchioli et al., 2017), yet benefit greatly from multi-tenancy since the alternative — cloud offloading — may result in unnacceptable network latency. In addition to these complexities, edge computing applications also often impose additional quality of service requirements such as deadlines and latency constraints, which must be balanced with the availability of compute, networking, or even power (Satyanarayanan, 1996; Zambonelli & Mamei, 2004).

The latency-sensitive nature of many edge workloads makes it useful, and sometimes crucial, to anticipate workload runtimes in deployment to ensure that they meet their quality-of-service needs. For example, an industrial controller on a manufacturing line may need to complete within a given timeframe with high probability to forestall interruptions to the manufacturing plant, or a smartphone might need to decide which model to load for a particular inference task. Indeed, such runtime performance measures are crucial for edge orchestration frameworks that aim to ensure workload performance by placing them on different available platforms. However, the heterogeneous, resource-limited nature of edge systems also limits the practicality of exhaustively benchmarking all possible deployments: even if each workload can be benchmarked on each platform, all combinations of potentially interfering workloads cannot.

In the absence of benchmark data, orchestrators must instead *predict* workload performance, of which predicting the execution time, or *runtime*, is a critical component, especially with latency-sensitive workloads. As a further complica-

[1]Department of Electrical and Computer Engineering, Carnegie Mellon University [2]Robert Bosch GmbH [3]School of Engineering (ISEP/IPP) and INESC TEC, Polytechnic Institute of Porto. Correspondence to: Tianshu Huang <tianshu2@andrew.cmu.edu>.

Proceedings of the 6th MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

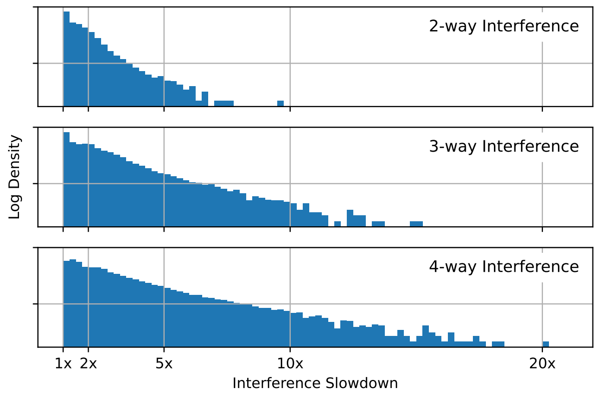


Figure 1. Log-histogram of interference effects in our dataset, sorted by the number of interfering workloads; we observe up to a 20× slowdown in randomly sampled benchmark combinations.

tion, accurate runtime estimates may not always be possible: accuracy may be constrained by insufficient data, or, since edge devices are harder to maintain than cloud datacenters, even unpredictable variations in the platforms themselves. Thus, we focus on three key challenges: maximizing data efficiency for a *matrix completion* formulation, *predicting interference* between concurrent workloads, and *quantifying the uncertainty* of our predictions.

Matrix Completion Predicting workload runtime across heterogeneous platforms must predict the impact of complex effects such as differences in compiler optimizations (Hoste & Eeckhout, 2008), operating system performance (Chen & Bershad, 1993), and hardware architecture (Zheng et al., 2015). Generalizing runtime predictions from a limited number of observations to multiple heterogeneous platforms thus requires us to intelligently combine observations of how different workloads perform on various platforms. While matrix completion is not new to performance analysis, we are the first to use it for explicit runtime prediction.

Interference Modeling Workloads running simultaneously (Fig. 1) can cause a range of interfering effects due to shared resource contention (Chai et al., 2007; Cavicchioli et al., 2017). This is of particular concern in edge systems, where co-locating interfering workloads may be preferable to the latency of offloading to a remote server.

Many prior works capture set interference types using microbenchmarks (Delimitrou & Kozyrakis, 2013; 2014); this strategy is well-suited for cloud systems with well-defined resources such as CPU, memory, and networking. However, edge systems can suffer from difficult-to-quantify types of resource contention such as scheduling and alignment, power and clock speed effects, or even contention for application-specific communication channels such as Bluetooth or CAN. We instead focus on an *observation-based* interference-aware prediction method that learns interference patterns by simply observing workload behavior, using a novel extension of matrix factorization.

Uncertainty Quantification Not all workloads and platforms have performance characteristics which are equally predictable: some may be more susceptible to indeterminism, have more complex behavior, or simply have less training data available. Thus, while a "mean" runtime prediction can be useful, an estimate which captures the *uncertainty* of a model's prediction can better guide design and deployment decisions — whether for an engineer or for an orchestration algorithm (Alipourfard et al., 2017).

We formalize this requirement by predicting *runtime bounds*: for a given tail ε, what runtime budget will be sufficient for a workload to complete with probability at least $1 - \varepsilon$? Applying conformal prediction (Shafer & Vovk, 2008), we develop a quantile regression-based procedure (Romano et al., 2019) for predicting tight bounds in probability.

Contributions Our runtime prediction method, Pitot[1], integrates a collection of novel contributions that draw insight across machine learning and systems theory, and provides up to a 2x improvement over a naive combination of existing methods. To summarize our contributions:

(1) **Runtime Prediction for the Edge**: We make several novel contributions to the matrix completion method (Sec. 3) that address challenges found in edge systems, including a *log-residual objective* to handle high heterogeneity, a *matrix factorization with side information* formulation to exploit workload and platform features, an *interference prediction* term to predict workload interference, and *uncertainty quantification* to probabilistically bound runtime.

(2) **Runtime Dataset**: In order to develop and evaluate our method, we collect a novel dataset with 410,970 unique data points from 249 benchmarks, 10 different runtime configurations, and 24 devices. Our dataset uses WebAssembly (Haas et al., 2017), a lightweight virtualization framework which allows us to run common benchmarks on platforms ranging from x86 servers to microcontrollers. WebAssembly has also seen increasing adoption from the cloud (Fastly, 2022; Cloudflare, 2018; Shopify, 2020) to edge applications such as IoT (Li et al., 2022), augmented reality (Pereira et al., 2021), automotive (Danzeisen, 2023; Scheidl, 2020), and industrial automation (Woods & Chhokra, 2022).

[1]Our code and dataset are open source, and can be found at `https://github.com/wiseLabCMU/pitot`; an archival copy is also available at `https://zenodo.org/records/14977004`.

(3) **Evaluation**: We run extensive experiments demonstrating the impact of each of our methodological contributions and verifying the efficacy of our method. In particular, Pitot achieves as low as 5.2% error, and can generate tightly-bounded prediction intervals (Sec. 5).

2 Related Work

Worst-Case Execution Time Predicting the worst-case execution time (WCET) — the longest time a program takes to finish, considering all possible inputs — has strong roots in real-time and embedded systems (Wilhelm et al., 2008). WCET algorithms bound runtime execution using pessimistic assumptions on aspects of program execution such as memory access (Ferdinand et al., 2007), loop counts (Puschner & Schedl, 1997), thread interaction (Dietrich et al., 2017), and execution paths (Ermedahl & Gustafsson, 1997). However, WCET estimates of ordinary programs can be extremely large or even unbounded, leading to the need for more data-driven runtime prediction approaches.

Platform Modeling Learning a performance model for each platform/device is popular for architecture-specific instruction throughput modeling (Mendis et al., 2019; Nemirovsky et al., 2017), or for relating runtime to different input data for a given program using historical examples on the same platform (Huang et al., 2010). If a reference platform (or CPU simulator (Zheng et al., 2015)) is available, performance features from dynamic profiling can be used also to learn the relation to performance on another platform (Zheng et al., 2016; 2017; Saeed et al., 2019).

Matrix Completion Modeling workload and platform characteristics jointly using a matrix completion approach such as matrix factorization can reduce the need for intrusive profiling or instrumentation. For example, Quasar and Paragon (Delimitrou & Kozyrakis, 2014; 2013) both use matrix factorization to predict QoS (quality-of-service) metrics for cloud orchestration, though neither use platform or application features. Other works have also explored matrix completion with side information for execution time prediction (Pham et al., 2017) using black-box methods.

Interference Modeling The difficulty of analytically modeling program interference has motivated several learning-based approaches, e.g., by constructing micro-benchmarks which are used as input features (Delimitrou & Kozyrakis, 2014; 2013) for predicting interference in the cloud. Previous edge-focused work also includes models targeting memory interference on embedded processors (Saeed et al., 2021), OpenCL memory interference (Lee & Wu, 2017), Simultaneous Multithreading contention (Moseley et al., 2005), and virtual machine interference (Koh et al., 2007). These works train a device-specific model, while our method generalizes across heterogeneous devices, and is the first to integrate interference modeling into matrix factorization.

Uncertainty Quantification Previous works using uncertainty quantification for workload performance prediction rely on Bayesian Optimization, for example for cloud workload configurations (Alipourfard et al., 2017) and time series workload demand forecasting (Zhou et al., 2022). We instead base our approach on conformal prediction (Shafer & Vovk, 2008; Romano et al., 2019), which is distribution-free and provides a mathematical guarantee of validity.

3 Method

Figure 2 illustrates our method, Pitot. As workloads are run on different platforms, possibly with other interfering workloads, we record the runtime of each (platform, workload, interference) tuple. We break down the key steps as follows:

(1) Pitot uses a **log-residual objective**, which has a normalizing effect and allows us to handle heterogeneity across several orders of magnitude (Sec. 3.2).

(2) A "two-tower" neural network-based **matrix factorization** model learns workload and platform embeddings from side information (Sec. 3.3).

(3) Pitot adds a novel **interference prediction** term which models arbitrary interference patterns while also accounting for interference in the training data (Sec. 3.4).

(4) If runtime bounds are required instead of a "mean" prediction, we *conformalize* the outputs of our model (Sec. 3.5) to provide **uncertainty bounds**.

Each step draws from insights across computer systems and machine learning literature and makes significant contributions to the efficacy of our method (Sec. 5.2), which together add up to over a 2x improvement over a naive application of existing methods (Sec. 5.3).

3.1 Problem Formulation

Assumptions Since predicting the runtime of unknown workloads consisting of arbitrary programs (and potentially infinite resource usage) is impossible in a general sense, we begin by making the following assumption:

- **Workloads are uniquely identifiable,** and if the nature of a workload changes, this can be identified externally. Thus, if a workload undergoes a phase shift, we assume that the shift can be identified, and the new phase treated as a new workload.
- **Each workload is observed at least once,** and we are never asked to predict the runtime of a workload using only features which are statically available.
- **Each platform is observed at least once,** and we are

Prediction (MLSys '24 Version)

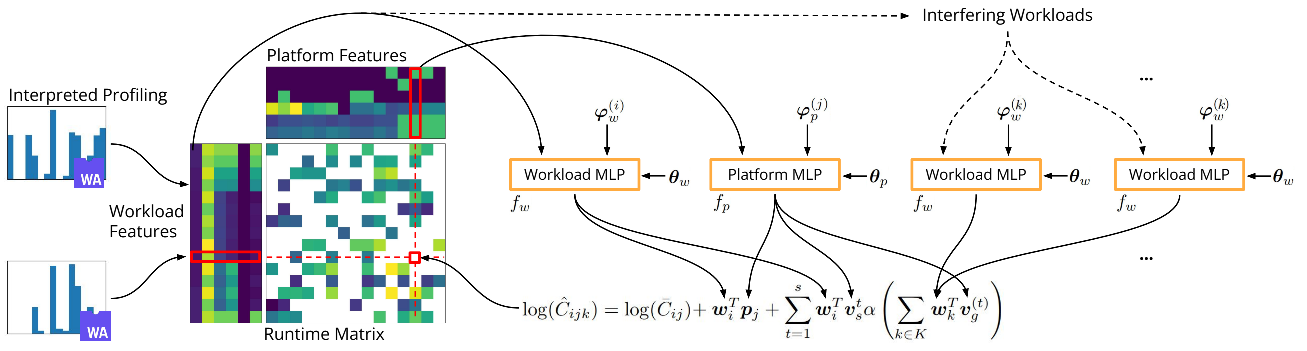


Figure 2. Illustration of Pitot's interpreted profiling, matrix factorization, and interference model. Workload and platform embeddings $\boldsymbol{w}_i$, $\boldsymbol{p}_j$ are first computed by embedding networks f_w, f_p from input features $\boldsymbol{x}_w^{(i)}, \boldsymbol{x}_p^{(j)}$ concatenated to learned features $\boldsymbol{\varphi}_w^{(i)}, \boldsymbol{\varphi}_p^{(j)}$. Then, for each (workload, platform) pair, Pitot adds the inner product $\boldsymbol{w}_i^T\boldsymbol{p}_i$ to the baseline $\bar{C}_{ij}$. If interfering modules are present, Pitot also computes the interference susceptibility $\boldsymbol{w}_i^T\boldsymbol{v}_s^t$ and magnitude $\boldsymbol{w}_k^T\boldsymbol{v}_g^{(t)}$ for each, and adds an interference term (Eq. 9). The resulting prediction is then compared to the observed runtime to train our model weights $\{\boldsymbol{\theta}_w, \boldsymbol{\theta}_p, \boldsymbol{\varphi}_w, \boldsymbol{\varphi}_p\}$.

never asked to predict runtime on a platform using only a platform description.

- **Observations are exchangeable.** In other words, the order in which workloads are run does not matter, and the characteristics of platforms are constant over time.

Formulation From these assumptions, we formulate our matrix completion problem as follows:

- Workloads are indexed by $i = 1, 2, \dots N_w$, and each have side information $\boldsymbol{x}_w^{(i)}$ (e.g. opcode count).
- Platforms are indexed by $j = 1, 2, \dots N_p$, and are associated with side information $\boldsymbol{x}_p^{(j)}$ (e.g. CPU, runtime environment, memory information).
- We observe runtimes C^*_{ijK} of workload i running on platform j for some subset $\mathcal{A}$ of (platform, workload, interference) tuples, where the interference K is an arbitrary set of simultaneously running workloads.

We then try to predict runtimes $\hat{C}_{ijK}$ for unobserved i, j, K.

3.2 Log-Residual Objective

Execution time and other resource usage in computer systems are known to be extremely long-tailed (Tirmazi et al., 2020), with the fastest and slowest programs, the fastest and slowest WebAssembly runtimes (ahead-of-time-compilers vs interpreters), and the strongest and weakest devices (modern x86 vs low-power embedded ARM) varying by several orders of magnitude in speed. When using an "absolute" objective such as l_2 error, this causes the total loss to be dominated by data with the largest magnitudes – the slowest programs running on the slowest platforms – where the same relative error accounts for disproportionate l_2 losses.

Log Runtime Using geometric instead of arithmetic averages is a well-established practice in benchmarking computer systems. This serves to prevent authors from arbitrarily weighting benchmarks by modifying each benchmark's total runtime to advantage methods of their choice[2].

We find that this insight is not restricted to the fairness of benchmarking: modeling performance multiplicatively can also "normalize" the distribution of the data, which can be seen by applying the Central Limit Theorem (CLT). Modeling workloads as a collection of largely independent tasks, the CLT implies that the runtime should be well-behaved in distribution (and roughly normal). However, from a performance analysis perspective, while these tasks – think a single instruction or system call – may be largely independent *conditioned on the platform*, they are highly correlated with the platform itself.

This implies that we should instead view workloads as a collection of performance characteristics which are *multiplied* to find the runtime – or characteristics which are added to find the log runtime, to which we can apply the CLT. As such, instead of simply normalizing workload runtimes by dividing by a metric (Delimitrou & Kozyrakis, 2013), we minimize the l_2 loss in *log* space:

$$\mathcal{L} = \sum_{(i,j,k)\in\mathcal{A}} ||\log(C^*_{ijk}) - \log(\hat{C}_{ijk})||_2^2. \qquad (1)$$

Residual Objective Under a log-objective, we can further normalize each workload and platform using a simple optimization procedure. Modeling workloads and platforms with a "difficulty" (i.e. instruction count) and "speed" (throughput), which, assuming linear performance scaling,

[2]This practice has led to arithmetic averaging being labeled a "benchmarking crime" (van der Kouwe et al., 2018), which we are careful not to commit.

are multiplied to find the runtime, we can learn a simple (interference-blind) model

$$\log(\bar{C}_{ij}) = \bar{w}_i + \bar{p}_j \tag{2}$$

for workload log "difficulty" $\bar{w}_i$ and platform log "speed" $\bar{p}_j$, which can be efficiently learned by alternating minimization (App. B.1). Instead of directly predicting the runtime $\hat{C}_{ijk}$, we construct the rest of our model to predict the residual of the baseline $y_{ijk} = \log(C^*_{ijk}) - \log(\bar{C}_{ij})$.

This objective has the key advantage of being preserved under simple scaling. All else being equal, a job i' consisting of γ repetitions of job i (i.e. $\hat{C}_{i'jk} = \gamma\hat{C}_{ijk}$ and $\bar{C}_{i'jk} = \bar{C}_{ijk}$) will satisfy $y_{ijk} = y_{i'jk}$. This provides significant advantages for prediction and interpretation, and can be seen by expanding $y_{i'jk}$:

$$\begin{aligned} y_{i'jk} &= \log(\gamma\hat{C}_{ijk}) - \log(\gamma\bar{C}_{ijk}) \\ &= \log(\hat{C}_{ijk}) - \log(\bar{C}_{ijk}) = y_{ijk}. \end{aligned} \tag{3}$$

3.3 Matrix Factorization Model

Matrix Factorization techniques decompose a partially observed low-rank target matrix into the product of multiple matrices. When decomposing a matrix into two matrices $\boldsymbol{C} = \boldsymbol{W}\boldsymbol{P}^{\boldsymbol{T}}$, each element C_{ij} (the runtime of workload i on platform j) is represented as the inner product $\boldsymbol{w}_i^T\boldsymbol{p}_j$ of the corresponding rows $\boldsymbol{w}_i$ in $\boldsymbol{W}$ and $\boldsymbol{p}_j$ in $\boldsymbol{P}$. This approach learns a common "embedding space": workloads and platforms with similar execution behavior should have similar $\boldsymbol{w}_i$ and $\boldsymbol{p}_j$ embeddings.

We predict the residual using matrix factorization (Fig. 2), with embeddings derived from the workload and module side information. Instead of analytical solutions such as (Chiang et al., 2015), we use the "two-tower" neural network architecture popular in recommender systems (Covington et al., 2016) to handle nonlinearity with respect to our side information $\boldsymbol{x}_w^{(i)}$ and $\boldsymbol{x}_p^{(j)}$. Specifically, the workload and platform embeddings $\boldsymbol{w}_i, \boldsymbol{p}_j \in \mathbb{R}^r$ for workload i and platform j are output by multi-layer perceptrons f_w and f_p:

$$\begin{aligned} \boldsymbol{w}_i &= f_w(\boldsymbol{x}_w^{(i)}, \boldsymbol{\varphi}_w^{(i)}; \boldsymbol{\theta}_w) \\ \boldsymbol{p}_j &= f_p(\boldsymbol{x}_p^{(j)}, \boldsymbol{\varphi}_p^{(j)}; \boldsymbol{\theta}_p). \end{aligned} \tag{4}$$

Here f_w, f_p have weights $\boldsymbol{\theta}_w, \boldsymbol{\theta}_p$; $\boldsymbol{\varphi}_w^{(i)}, \boldsymbol{\varphi}_p^{(j)} \in \mathbb{R}^q$ are additional parameters[3] associated with each workload and platform that are appended to side information $\boldsymbol{x}_w^{(i)}$ and $\boldsymbol{x}_p^{(j)}$. The matrix factorization term is then added to Eq. 2 to obtain the interference-blind prediction

$$\log(\hat{C}_{ij}) = \log(\bar{C}_{ij}) + \boldsymbol{w}_i^T\boldsymbol{p}_j. \tag{5}$$

Model Architecture In our experiments, the workload and platform embedding networks f_w, f_p each have 2 hidden layers of 128 units, and GELU activation on all hidden layers. When used with our dataset (Section 4), Pitot has 111,200 parameters split roughly equally between workload and platform embedding networks $\boldsymbol{\theta}_w, \boldsymbol{\theta}_p$, with a negligible number making up the additional learnable features $\boldsymbol{\varphi}_w^{(i)}, \boldsymbol{\varphi}_p^{(j)}$.

3.4 Interference-Aware Prediction

Inspired by the general principle of representation learning applied by matrix factorization – learning a common embedding space for all workloads and platform, we add a term to our model that models interference in this embedding space. We begin by modeling the interference caused by a single workload using a low-rank "interference matrix" $\boldsymbol{F}_j$ for each platform j:

$$\log(\hat{C}_{ijk}) = \log(\bar{C}_{ij}) + \boldsymbol{w}_i^T\boldsymbol{p}_j + \boldsymbol{w}_i^T\boldsymbol{F}_j\boldsymbol{w}_k. \tag{6}$$

This term, $\boldsymbol{w}_i^T\boldsymbol{F}_j\boldsymbol{w}_k$, models the log-performance penalty caused by workload $\boldsymbol{w_k}$ running alongside workload $\boldsymbol{w}_i$ on a platform with an interference matrix $\boldsymbol{F}_j$. Unlike previous works (Delimitrou & Kozyrakis, 2014; 2013), this allows us to capture the key asymptotic benefits of matrix factorization: the ability to learn interactions between workloads and platforms simply by observing them, and without needing to explicitly model or benchmark these interactions.

Interpreting $\boldsymbol{F}_j$ This interference term can be interpreted as an extension of matrix factorization that models interference in the common embedding space. Consider the SVD (singular value decomposition)-like decomposition $\boldsymbol{F}_j = \sum_{t=1}^{s} \boldsymbol{v}_s^{(t)}\boldsymbol{v}_g^{(t)T}$ for a rank constraint s. Expanding our runtime prediction with this decomposition,

$$\boldsymbol{w}_i^T\boldsymbol{p}_j + \boldsymbol{w}_i^T\boldsymbol{F}_j\boldsymbol{w}_k = \boldsymbol{w}_i^T\boldsymbol{p}_j + \sum_{t=1}^{s} \boldsymbol{w}_i^T\boldsymbol{v}_s^{(t)}\boldsymbol{w}_k^T\boldsymbol{v}_g^{(t)} \tag{7}$$

$$= \boldsymbol{w}_i^T\left(\boldsymbol{p}_j + \sum_{t=1}^{s} \boldsymbol{v}_s^{(t)}\boldsymbol{w}_k^T\boldsymbol{v}_g^{(t)}\right). \tag{8}$$

The model can be viewed (Eq. 7) as the sum of s types of interference, where the interfering module $\boldsymbol{w}_k$ causes a "magnitude" $\boldsymbol{w}_k^T\boldsymbol{v}_g^{(t)}$ of type t interference, and the workload $\boldsymbol{w}_i$ has a "susceptibility" $\boldsymbol{w}_i^T\boldsymbol{v}_s^{(t)}$ to type t interference. Factoring $\boldsymbol{w}_i$ out, this can also be interpreted (Eq. 8) as moving the platform embedding $\boldsymbol{p}_j$ in the direction $\boldsymbol{v}_s^{(t)}$ with magnitude $\boldsymbol{w}_k^T\boldsymbol{v}_g^{(t)}$ for each type t of interference.

[3]This captures information that cannot be expressed as a function of the input features, for example if two workloads or platforms differ in hard-to-measure ways such as memory access patterns or memory latency, and is essential for the model to have sufficient representational capacity D.2.

Multiple Interfering Workloads If multiple interfering workloads are present, we could extend our model by adding the interference magnitude $\boldsymbol{w}_k^T \boldsymbol{v}_g^{(t)}$ for each workload $k \in K$ using the susceptibility-magnitude representation (Eq. 7). However, this assumes that workloads only cause multiplicative (i.e. log-additive) interference effects. We instead apply an activation function α to the total magnitude for each type:

$$\begin{aligned}\log(\hat{C}_{ijk}) &= \log(\bar{C}_{ij}) \\ &+ \boldsymbol{w}_i^T \boldsymbol{p}_j + \sum_{t=1}^{s} \boldsymbol{w}_i^T \boldsymbol{v}_s^{(t)} \alpha \left(\sum_{k \in K} \boldsymbol{w}_k^T \boldsymbol{v}_g^{(t)} \right). \end{aligned} \tag{9}$$

This allows us to model the case that little interference is observed until a certain threshold, while also allowing any relationship between interference magnitude and runtime slowdown to be approximated with a sufficiently large s.

Model Architecture We learn $\boldsymbol{v}_s$ and $\boldsymbol{v}_g$ for each platform j by adding additional output heads to f_p. We then compute our interference term using the factorization given in Eq. 9.

For the interference activation α, we use a leaky ReLU activation with negative slope 0.1, since ordinary ReLU activation functions often lead to "dead" interference types (i.e. becoming extremely negative) due to poor initialization.

3.5 Uncertainty Quantification using Conformalized Quantile Regression

When applying runtime prediction to resource allocation or system design, practitioners are often interested in not just the expected resource usage, but some (over-provisioned) bound which will be sufficient with high probability. Stated formally, for a workload distribution C^*, we would like to predict a regression output $\tilde{C}^{(\varepsilon)}$ such that

$$\Pr(C^* < \tilde{C}^{(\varepsilon)}) < \varepsilon \tag{10}$$

for a one-sided *target miscoverage rate* $\varepsilon > 0$ that represents the probability that the predicted runtime is insufficient for the target workload. Given the miscoverage rate as a constraint, our objective is to make our bound as "tight as possible," minimizing the *overprovisioning margin*

$$m = \mathbb{E}[\max(\tilde{C}^{(\varepsilon)} - C^*, 0)/C^*], \tag{11}$$

which measures the relative resource overprovisioning that would occur using this prediction bound.

This motivates the usage of *split conformal regression* (Shafer & Vovk, 2008), which uses a hold-out calibration set to provide statistically guaranteed bounds in probability (under the assumption of exchangeability). Split conformal regression can be used on any regression algorithm, and works by "calibrating" the regression output by adding a constant offset γ to the output predictions $\hat{C}$:

$$\tilde{C}^{(\varepsilon)} = \hat{C} + \gamma : \quad \Pr(C^* < \hat{C} + \gamma) = \varepsilon. \tag{12}$$

However, when applied directly to a least-squares regression output, split conformal regression is not adaptive, and can only quantify the uncertainty of the model as a whole.

Conformalized Quantile Regression Quantile regression (Koenker & Bassett Jr, 1978) uses a "pinball" loss which estimates a *target quantile* ξ when minimized:

$$\mathcal{L} = \begin{cases} \xi(\hat{C} - C^*) & C^* > \hat{C} \\ (1-\xi)(\hat{C} - C^*) & C^* \le \hat{C} \end{cases}. \tag{13}$$

While this is not guaranteed to result in predictions which capture the target quantile, and in practice can be far from the target quantile, quantile regression does provide an *adaptive* measure of uncertainty. By applying conformal regression to the output of quantile regression, we can obtain adaptive, yet calibrated, predictions; this is known as *Conformalized Quantile Regression* (CQR) (Romano et al., 2019).

Optimal Quantile Choice Common practice in CQR[4] is to set the target quantile as the miscalibration rate (i.e., $\xi = \varepsilon$); we refer to this as a "naive" application of CQR. No theoretical results justify this practice, and we find that in some cases, the quantile which results in the tightest bounds once calibrated (i.e. minimizing Eq. 11) can vary greatly from the target miscoverage rate (App. B.2).

Instead, we train Pitot on a spread of different ξ; then, for a target ε at test time, we calibrate each output for ε. Finally, we calculate the tightness (Eq. 11) of each resulting predictor on the validation set, and select the best one. In addition to providing a modest boost to bound tightness, this also allows us to calibrate Pitot for an arbitrary ε without needing to retrain the model, while only increasing the training time by $\approx 5\%$.

Calibration Pools Conformalized quantile regression (and conformal prediction more broadly) only requires the assumption of *exchangeability*: the ability to swap an observation in the calibration set with one in the test set without modifying the joint distribution of the calibration set. Notably, there is no requirement to have only *one* calibration set: if sufficient data is available, we can partition the data based on some discrete random variable I into multiple calibration and test set pairs while preserving exchangeability conditioned on I.

More homogeneous calibration sets are known to lead to smaller prediction intervals (Sousa et al., 2022). Since we

[4]This corresponds to $\xi = \varepsilon/2$ for the more commonly used two-sided CQR (Sousa et al., 2022; Romano et al., 2019).

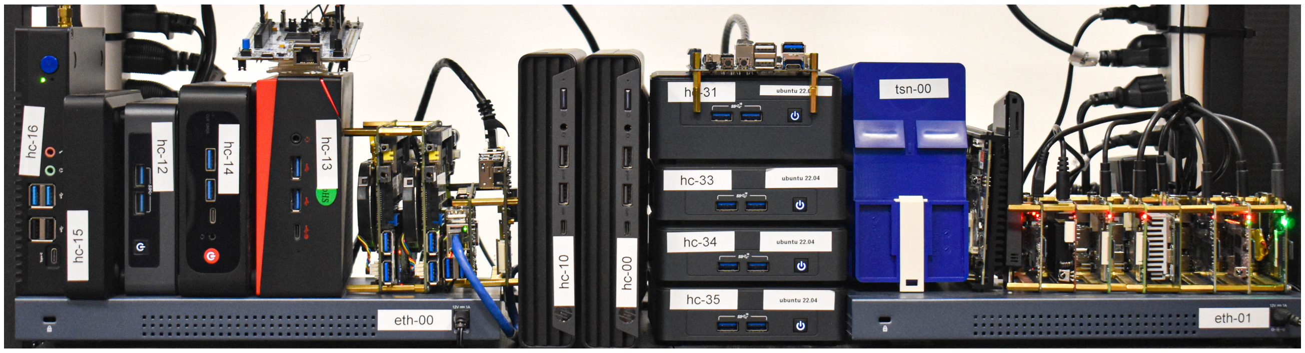


Figure 3. Heterogeneous cluster test bench used to collect our dataset; our cluster includes Intel and AMD-based x86 computers, ARM A-class single board computers, as well as a RISC-V SBC and an ARM M-class microcontroller.

observe that program runtime is much more unpredictable when more interfering workloads are running, we split our data into different calibration pools depending on how many workloads were running simultaneously. As an added benefit, conditioning on the number of simultaneously-running workloads as I allows Pitot to maintain *conditional* exchangeability even under distribution shift of I.

Model Architecture Neural network-based quantile regression methods typically add multiple output heads, and train them jointly using a weighted objective. However, in embedding-based models such as matrix factorization, the output is a vector (or in our case, a collection of vectors), which would significantly increase training and inference time if naively duplicated for each target quantile ξ.

To limit this impact, we exploit the fact that only one of the embedding networks needs to have multiple outputs. Furthermore, Pitot's embeddings are not "balanced": the workload embedding consists of a single vector, while the platform embedding also must learn interference terms $\boldsymbol{v}_s$ and $\boldsymbol{v}_g$. As such, we learn multiple workload embeddings, and reuse the same platform embedding for each ξ.

3.6 Training and Implementation

Pitot is trained using stochastic gradient descent, and is extremely lightweight, with a single inference call taking $\approx$400Kflops, and training taking only 12.1 seconds (using a RTX 4090 on our dataset), including validation and checkpointing. For a detailed description of Pitot's hyperparameters, training procedure, and implementation, see Appendix B.3.

4 Dataset

In order to develop and evaluate *Pitot*, we assembled a heterogeneous compute cluster (Fig. 3) running a WebAssembly-based edge orchestration framework, and collected a dataset of 53,637 observations of different (workload, platform) pairs, and an additional 357,333 observations of workloads running on different platforms with up to 3 interfering workloads in the background. In total, this dataset represents approximately 80 hours of continuous data collection on our cluster. We provide a full description of our dataset and data collection methodology in Appendix C.

Workloads Our datatet uses 249 workloads drawn from the following benchmark suites:

- Polybench (Pouchet, 2015): numerical floating-point-heavy kernels.
- MiBench (Guthaus et al., 2001): a diverse collection of miscellaneous benchmarks.
- UCSD Cortex Suite (including the San Diego Vision Benchmark Suite) (Thomas et al., 2014): computer vision and machine learning benchmarks.
- Libsodium benchmark suite: cryptography benchmarks from the Libsodium test and benchmark suite.
- Python: 12 benchmarks written for CPython, run on a WebAssembly-compiled CPython using WASI.

While some benchmarks required light modifications to compile to WebAssembly and run on our platforms, continual efforts to improve software support such as extensions to the WebAssembly Standard Interface or by exposing linux system calls (Ramesh, 2023) raise the possibility of bringing more legacy applications to WebAssembly in the future.

Platforms We assembled a cluster (Fig. 3) of 24 different devices ranging from a microcontroller[5] and various single-board computers to x86 desktop computers, and ran each benchmark using a suite of different WebAssembly runtimes on each device (App. C.1).

[5]Notably, the microcontroller executes some of the smallest benchmarks faster than many linux-based platforms due to the absence of operating system overhead.

Input Features Bytecode-based virtualization allows a system to "inspect" program execution in a cross-platform way instead of treating programs as "black box" binaries. To take advantage of this, we count the number of times each opcode was executed using an instrumented interpreter, which we collect as workload features $\boldsymbol{x}_w$ (similar to (Kuperberg & Becker, 2007)). We also collect data about each platform such as CPU and WebAssembly host information, which we provide as platform features $\boldsymbol{x}_p$. For a detailed specification of these features, see Appendix C.2.

Limitations Our dataset is limited to a set of publicly available benchmarks that we were able to run on our test cluster; these benchmarks are primarily compute-bound, though some do feature a significant amount of filesystem I/O. In our dataset and formulation, we also assume that workloads either have a constant input data distribution, or perform computations that do not depend on the input data (e.g., control algorithms); workloads with phase changes due to alternate input data (e.g., data analysis) are treated as separate workloads similar to other works (Zheng et al., 2015; Delimitrou & Kozyrakis, 2014; 2013). As such, different data inputs or data sizes for the same binary or source code in our benchmarks (e.g., different python scripts for the same Python binary) are treated as separate workloads.

5 Results

To evaluate our contributions, we ran ablations showing how each aspect of Pitot contributes to its superior performance (Sec. 5.2) compared to existing approaches (Sec. 5.3). We also show how Pitot learns interpretable embeddings that can potentially be used for other tasks (Sec. 5.4).

5.1 Experiment Setup

Evaluation To measure data efficiency, we evaluated each method in training data splits of $10\%, 20\%, \ldots 90\%$ of the data. We trained 5 replicates for each configuration (method and train split size), with each replicate independently receiving a train and test set. Within the training set, 80% was used for actual method training, while the remaining 20% was used for validation and calibration (where applicable).

Since predicting the runtime of workloads running with interference is intrinsically harder than for workloads running in isolation (and the weighting between the two objectives is arbitrary), we also show the performance on test data with and without interfering workloads separately.

Error To evaluate the accuracy of our average runtime predictions, we report the Mean Absolute Percent Error (MAPE) on the holdout test set $\mathcal{A}^c$.

Tightness To evaluate the tightness of our predicted uncertainty bounds, we compute the overprovisioning margin (Eq. 11) for miscoverage rates $\varepsilon = 0.1, 0.09, \ldots 0.01$, which measures the average excess of the predicted bound compared to the actual runtime.

Note that while models trained to predict the expected runtime (i.e. squared loss minimization) can be calibrated and used to predict (conformalized) bounds, it is not appropriate to evaluate models trained to predict a target quantile using an error metric. As such, we evaluate error on a version of Pitot trained with a squared loss (Eq. 1) and evaluate tightness on a version trained with quantile regression (Eq. 13).

5.2 Method Ablations

Pitot draws its accurate predictions and tight bounds not from a single key insight, but from the collection of improvements we highlight in this paper. To demonstrate this, we ran ablations on the contribution of each improvement.

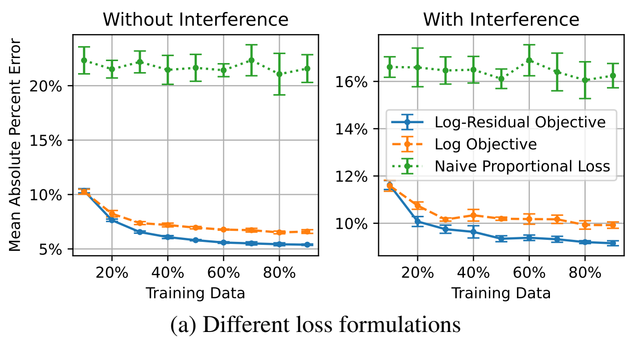
Hyperparameter Ablations We provide ablations for key hyperparameters of Pitot in Appendix D.2; as long as sufficient representational capacity is provided, Pitot is not sensitive to the choice of hyperparameters.

Log-residual Objective Figure 4a shows the impact of our log-residual objective compared to an ordinary log objective and a naive proportional loss on the prediction error. Due to the large variation in runtime magnitudes, a naive proportional loss cannot achieve reasonable error. Adding a residual objective also significantly improves accuracy, especially as more of the dataset is observed.

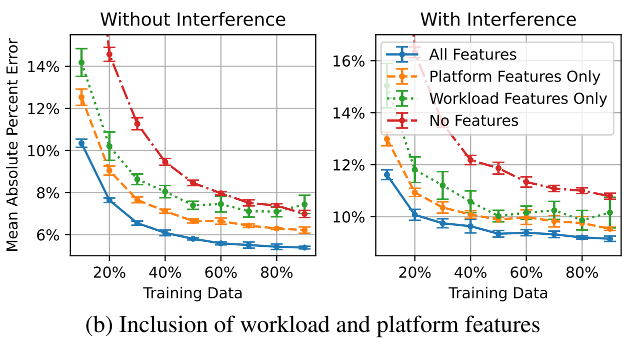
Side Information Adding side information in the form of workload and platform features can significantly boost accuracy, especially when only a small amount of data is observed. In our experiments, both platform and workload features significantly decrease error, especially when used together (Fig. 4b). Platform features do have a significantly higher marginal impact than workload features; this is likely due to the presence of similar platforms in our dataset (e.g. different Cortex-A53 processors) which the model can pick up on. In larger datasets where many similar workloads are present (e.g. different versions of the same workload), this relationship would likely be reversed.

Interference Prediction We compare our interference-aware method (Eq. 6) to two alternate strategies: *discard*, where we discard any observations with interfering workloads, and *ignore*, where all observations are treated the same, regardless of any interference.

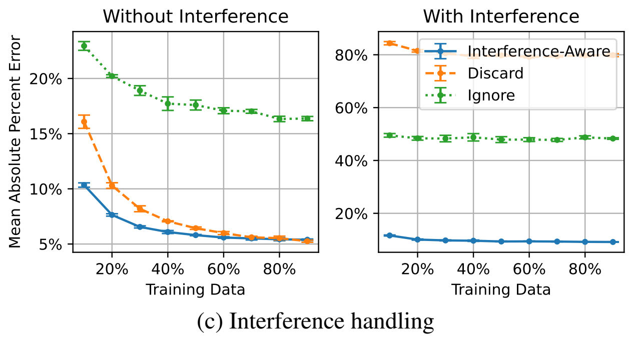
Discarding data leads to a low error floor when predicting data without interference, though the predictor cannot cap-



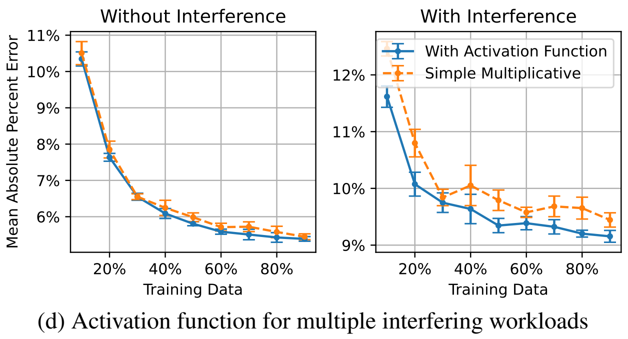
(a) Different loss formulations



(b) Inclusion of workload and platform features



(c) Interference handling



(d) Activation function for multiple interfering workloads

Figure 4. Ablations of key aspects of our Pitot. Each figure shows the mean absolute percent error (± 2 standard errors) for varying amounts of training data; error for test data with and without interference are shown separately.

ture interference. On the other hand, ignoring the effects of interference leads to a predictor with much higher error, since the effects of interference end up "averaged in" to all predictions. By modeling interference, Pitot can also use observations with interference to improve overall accuracy: when less data is observed, the interference-aware model has a significantly lower error than an interference-blind model only trained on "clean" data (Fig. 4c).

Multiple Interfering Workloads In order to model interference relationships beyond a simple multiplicative model, Pitot includes an activation function applied to the learned interference "magnitude" (Eq. 9). Figure 4d shows the modest but significant impact of including this activation function, indicating that "interference thresholds" have a small but significant effect.

Uncertainty Quantification Figure 5 shows the impact of our CQR-based uncertainty quantification method compared to a naive application of CQR and calibrating an ordinary predictor not trained using quantile regression. CQR produces significantly tighter bounds especially for small miscoverage rates, while our quantile choice method improves CQR significantly at larger miscoverage rates.

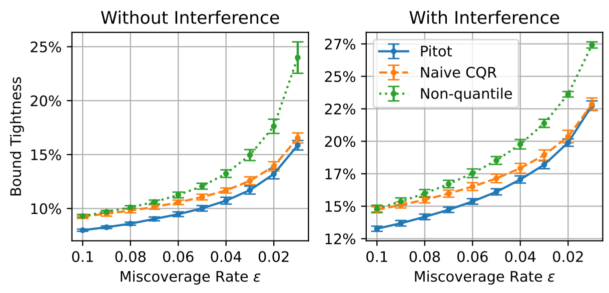
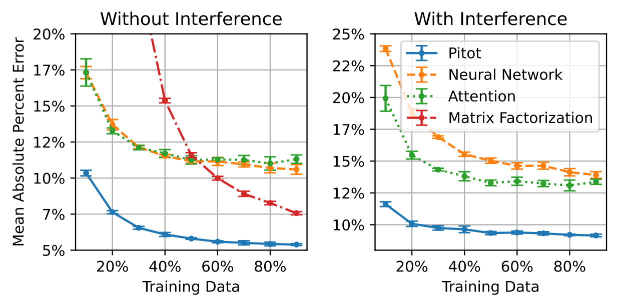


Figure 5. Bound tightness (± 2 standard errors) of our conformalized quantile regression algorithm compared to naive approaches for varying miscoverage rates when trained on 50% of the dataset.

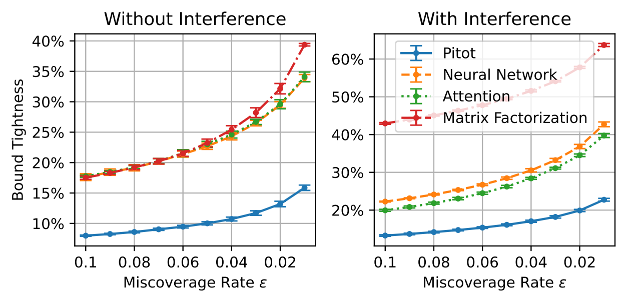
5.3 Baselines

While no prior works tackle runtime prediction in the same setting (interference-aware runtime prediction with interference in a heterogeneous environment), we compare Pitot with several baselines designed from elements drawn from state-of-the-art prediction algorithms. For additional details on our baselines, see Appendix B.4.

Matrix Factorization As our first baseline, we use a matrix factorization model similar to (Delimitrou & Kozyrakis, 2014; 2013) which predicts the (log) runtime by learning a feature vector for each workload and platform. This model does use workload or platform features, and is not



(a) Error for different amounts of training data. Matrix Factorization is not visible due to exceeding 75% error in all cases; see App. D.1 for an uncropped version.



(b) Bound tightness for varying miscoverage rates when trained on 50% of data; see Appendix D.3 for other data splits.

Figure 6. Comparisons against baselines; Pitot has significantly less error and tighter bounds in all settings that we evaluated. *Matrix factorization* performs better relative to other baselines when more data is observed, but lacks data efficiency. Also, while *attention* is less accurate than our interference model, it does perform better than *neural network*, supporting the need for better interference modeling.

interference-aware (and discards any observations with interference) since matrix factorization cannot be easily extended to explicitly predict the impact of interference on runtime[6].

Neural Network For a stronger baseline, we use a neural network-based model which takes our workload and platform features as inputs similar to (Pham et al., 2017) to generate a base, interference-blind predictions. To handle interference, we add a second neural network which predicts an interference multiplier (Saeed et al., 2021) for each interfering workload.

Attention For our final baseline, we replace the simple multiplicative interference model in our neural network baseline with an attention mechanism followed by an output head which predicts a single interference multiplier which is applied to the base prediction. This attention mechanism is somewhat similar to Pitot's interference model, which can be thought of as a simple attention mechanism with a theory-informed output function instead of a neural network.

Comparison with Baselines Pitot has both significantly better accuracy (Fig. 6a) and tighter bounds (Fig. 6b) compared to each baselines for all evaluation settings (App. D.3). Unlike the "pure" *matrix factorization* baseline without workload or platform features, Pitot is highly data efficient, and has lower error and tight bounds even when only a small amount of data is observed. Calibrating the *neural network* and *attention* baselines using split conformal regression to predict uncertainty bounds, Pitot also has much better accuracy and tighter bounds, demonstrating the advantage of our principled approach compared to generic neural networks.

Overall, Pitot is an extremely accurate predictor which can have up to 48% less error and 58% tighter bounds (and an average of 36% error and 44% tighter) compared to the next best baseline across each evaluation setting.

5.4 Model Interpretation

Unlike black-box methods, Pitot's matrix factorization-based approach learns embeddings $\boldsymbol{w}_i, \boldsymbol{p}_j$ for each workload and platform. These embeddings map similar workloads and platforms to nearby points in the embedding space, thereby quantifying their performance characteristics, and could be used for downstream tasks such as clustering or anomaly detection. Figure 7 shows a t-distributed Stochastic Neighbor Embedding (t-SNE) of our learned workload embeddings. We can observe a clear clustering of workloads by benchmark suite[7], demonstrating the interpretability of the platform embeddings. For additional visualizations of platform and interference embeddings, see Appendix D.4.

6 Conclusion

We formulate workload runtime prediction as an interference-aware matrix completion problem, and present our solution, Pitot, which combines several novel contributions including a novel "log-residual" training objective, interference-aware matrix factorization, and uncertainty quantification using conformalized quantile regression to make predictions which are far more accurate and tightly bounded than combining existing methods.

[6]While (platform, workload, interference) tensor completion is possible, the size increases exponentially with each additional interfering workload, quickly leading to unworkable sparsity. While (Delimitrou & Kozyrakis, 2013) does include a matrix factorization-based interference model, it relies on a microbenchmarking approach for cloud applications, and cannot predict the runtime time impact of an arbitrary interfering workload — only a relative measure of interference impact and susceptibility.

[7]Especially for relatively homogenous benchmark suites such as Polybench and Libsodium.

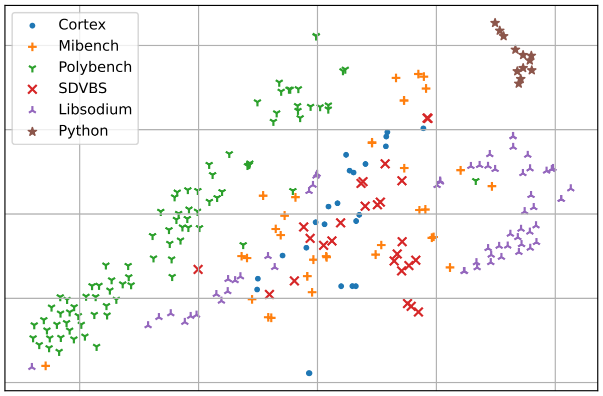


Figure 7. 2-dimensional t-SNE of workload embeddings by benchmark suite; the axes shown do not haves any particular meaning.

Possible applications of Pitot such as hardware-software co-design and edge orchestration could also benefit from future extensions that build on our work, including efficient online learning or statistical bounding techniques for miscoverage rates beyond what is possible using distribution-free approaches such as conformal prediction. We hope that our dataset – which we intend to be a "living" project that will be updated as we add new platforms to our cluster, new benchmarks to our collection, and new measurement types – will enable other to tackle these problems.

Limitations While our dataset is substantial, it cannot fully represent distributed systems deployed at scale. Edge orchestration frameworks for cyber-physical systems (Ruppel, 2023) have not yet seen widespread deployment, leading to a lack of real-world applications which can be used to test performance prediction algorithms; as such, Pitot is a methodological proof of concept, not a trained and deployable model. However, with sufficient buy-in from a large organization or deployment framework, we believe that our method could be applied at scale and provide substantial utility to practitioners and system managers alike.

Machine learning workloads present another challenge: although predicting the performance of machine workloads is of great relevance in edge computing, the lack of software portability and standardized workload/platform interfaces makes cross-platform runtime analysis impractical. Thus, as the edge-ML ecosystem matures, it may enable future work to bring Pitot – or a similar technique – to machine learning workloads.

References

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

Amiri, M. and Mohammad-Khanli, L. Survey on prediction models of applications for resources provisioning in cloud. *Journal of Network and Computer Applications*, 82:93–113, 2017.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Cavicchioli, R., Capodieci, N., and Bertogna, M. Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–10. IEEE, 2017.

Chai, L., Gao, Q., and Panda, D. K. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *Seventh IEEE international symposium on cluster computing and the grid (CCGrid'07)*, pp. 471–478. IEEE, 2007.

Chen, J. B. and Bershad, B. N. The impact of operating system structure on memory system performance. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pp. 120–133, 1993.

Chiang, K.-Y., Hsieh, C.-J., and Dhillon, I. S. Matrix completion with noisy side information. *Advances in neural information processing systems*, 28, 2015.

Cloudflare. Webassembly on cloudflare workers, 2018. URL `https://blog.cloudflare.com/webassembly-on-cloudflare-workers/`.

Covington, P., Adams, J., and Sargin, E. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pp. 191–198, 2016.

Danzeisen, M. Truly portable vehicle applications using webassembly & wasi. COVESA All Member Meeting, April 2023. URL `https://wiki.covesa.global/display/WIK4/COVESA+All+Member+Meeting+~+April+25-27%2C+2023`.

Delimitrou, C. and Kozyrakis, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

Delimitrou, C. and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

Dietrich, C., Wägemann, P., Ulbrich, P., and Lohmann, D. Syswcet: Whole-system response-time analysis for fixed-priority real-time systems (outstanding paper). In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 37–48, April 2017. doi: 10.1109/RTAS.2017.37.

Ermedahl, A. and Gustafsson, J. Deriving annotations for tight calculation of execution time. In *Euro-Par'97 Parallel Processing: Third International Euro-Par Conference Passau, Germany, August 26–29, 1997 Proceedings 3*, pp. 1298–1307. Springer, 1997.

Fastly. Compute@Edge, 2022. URL https://docs.fastly.com/products/compute-at-edge.

Ferdinand, C., Heckmann, R., and Franzen, B. Static memory and timing analysis of embedded systems code. In *Proceedings of VVSS2007-3rd European Symposium on Verification and Validation of Software Systems, 23rd of March*, pp. 07–04, 2007.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pp. 3–14. IEEE, 2001.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 185–200, 2017.

Hoste, K. and Eeckhout, L. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 165–174, 2008.

Huang, L., Jia, J., Yu, B., Chun, B.-G., Maniatis, P., and Naik, M. Predicting execution time of computer programs using sparse polynomial regression. *Advances in neural information processing systems*, 23, 2010.

Koenker, R. and Bassett Jr, G. Regression quantiles. *Econometrica: journal of the Econometric Society*, pp. 33–50, 1978.

Koh, Y., Knauerhase, R., Brett, P., Bowman, M., Wen, Z., and Pu, C. An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pp. 200–209. IEEE, 2007.

Kuperberg, M. and Becker, S. Predicting software component performance: On the relevance of parameters for benchmarking bytecode and apis. *Universität Karlsruhe*, pp. 30, 2007.

Lee, S.-Y. and Wu, C.-J. Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference. In *2017 IEEE international symposium on workload characterization (IISWC)*, pp. 43–53. IEEE, 2017.

Li, B., Fan, H., Gao, Y., and Dong, W. Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pp. 261–272, 2022.

Mendis, C., Renda, A., Amarasinghe, S., and Carbin, M. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*, pp. 4505–4515. PMLR, 2019.

Moseley, T., Kihm, J. L., Connors, D. A., and Grunwald, D. Methods for modeling resource contention on simultaneous multithreading processors. In *2005 International Conference on Computer Design*, pp. 373–380. IEEE, 2005.

Nemirovsky, D., Arkose, T., Markovic, N., Nemirovsky, M., Unsal, O., and Cristal, A. A machine learning approach for performance prediction and scheduling on heterogeneous cpus. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 121–128. IEEE, 2017.

Pereira, N., Rowe, A., Farb, M. W., Liang, I., Lu, E., and Riebling, E. Arena: The augmented reality edge networking architecture. In *2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 479–488. IEEE, 2021.

Pham, T.-P., Durillo, J. J., and Fahringer, T. Predicting workflow task execution time in the cloud using a two-stage machine learning approach. *IEEE Transactions on Cloud Computing*, 8(1):256–268, 2017.

Pouchet, L.-N. Polybench/c: the polyhedral benchmark suite, 2015. URL http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

Puschner, P. P. and Schedl, A. V. Computing maximum task execution times-a graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

Ramesh, A. Webassembly linux interface, 2023. URL https://github.com/arjunr2/wali.

Romano, Y., Patterson, E., and Candes, E. Conformalized quantile regression. *Advances in neural information processing systems*, 32, 2019.

Ruppel, E. Keynote: When wasm meets cyber-physical systems: A discussion of webassembly in real-time, safety-critical systems, 9 2023. URL `https://wasmcon2023.sched.com/`.

Saeed, A., Wurst, F., Pressler, M., Hamann, A., and Dasari, D. Machine learning based cross-platform runtime prediction. In *10th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2019.

Saeed, A., Mueller-Gritschneder, D., Rehm, F., Hamann, A., Ziegenbein, D., Schlichtmann, U., and Gerstlauer, A. Learning based memory interference prediction for co-running applications on multi-cores. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pp. 1–6. IEEE, 2021.

Satyanarayanan, M. Fundamental challenges in mobile computing. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pp. 1–7, 1996.

Scheidl, F. Webassembly: Paving the way towards a unified and distributed intra-vehicle computing-and data-acquisition-platform? In *2020 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*, pp. 1–6. IEEE, 2020.

Shafer, G. and Vovk, V. A tutorial on conformal prediction. *Journal of Machine Learning Research*, 9(3), 2008.

Shopify. How shopify uses webassembly outside of the browser, 2020. URL `https://shopify.engineering/shopify-webassembly`.

Sousa, M., Tomé, A. M., and Moreira, J. Improved conformalized quantile regression. *arXiv preprint arXiv:2207.02808*, 2022.

Thomas, S., Gohkale, C., Tanuwidjaja, E., Chong, T., Lau, D., Garcia, S., and Taylor, M. B. Cortexsuite: A synthetic brain benchmark suite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 76–79. IEEE, 2014.

Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., Harchol-Balter, M., and Wilkes, J. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pp. 1–14, 2020.

van der Kouwe, E., Andriesse, D., Bos, H., Giuffrida, C., and Heiser, G. Benchmarking crimes: an emerging threat in systems security. *arXiv preprint arXiv:1801.02381*, 2018.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

Woods, C. and Chhokra, A. An end-to-end toolchain for evaluating webassembly runtimes for cps-iot use cases. WebAssembly Research Day 2022, October 2022. URL `https://www.cs.cmu.edu/~wasm/wasm-research-day-2022.html`.

Xu, J., He, L., Wang, X., Huang, W., and Wang, N. A fast webassembly interpreter design in wasm-micro-runtime, 2021.

Zambonelli, F. and Mamei, M. Spatial computing: An emerging paradigm for autonomic computing and communication. In *Workshop on Autonomic Communication*, pp. 44–57. Springer, 2004.

Zheng, X., Ravikumar, P., John, L. K., and Gerstlauer, A. Learning-based analytical cross-platform performance prediction. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 52–59. Citeseer, 2015.

Zheng, X., John, L. K., and Gerstlauer, A. Accurate phase-level cross-platform power and performance estimation. In *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6, 2016.

Zheng, X., Vikalo, H., Song, S., John, L. K., and Gerstlauer, A. Sampling-based binary-level cross-platform performance estimation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1709–1714. IEEE, 2017.

Zhou, Z., Zhang, Y., and Delimitrou, C. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 1–14, 2022.

A Glossary of Notation

We provide a glossary of the symbols used in this paper for convenience in Table 1.

B Additional Method Details

In this section, we provide details regarding the architecture, training, calibration, and implementation of Pitot. Our code and dataset are also open source, and can be found at `https://github.com/wiseLabCMU/pitot`; an archival copy is also available at `https://zenodo.org/records/14977004`.

B.1 Linear Scaling Baseline

In this section, we provide a brief sketch of how we learn the parameters of the baseline model. We refer to this model as the *Linear Scaling Baseline* since it corresponds to common benchmarking practice where (geometric) mean benchmarking scores are used to estimate a linear relationship between platforms. Note that the linear scaling baseline is only learned from data collected with no interfering workloads running in the background.

Proposition 1. *Since the log-loss (Eq. 1) is convex for $\bar{m}_i$ and $\bar{p}_j$, we can efficiently learn the linear scaling model $\log(\bar{C}_{ij}) = \bar{w}_i + \bar{p}_j$ from C^*_{ij} by alternating minimization over $\bar{w}_i$ and $\bar{p}_j$ using the update rule*

$$\bar{m}_i = \frac{\sum_{i,j\in\mathcal{A}} \log(C^*_{ij}) - \bar{p}_j}{\sum_{i,j\in\mathcal{A}} 1}, \tag{14}$$

with a similar rule applying for $\bar{p}_j$.

Convexity can easily be verified by noting that the loss (Eq. 1) is the sum of convex quadratics (with respect to $\bar{m}_i$ and $\bar{p}_j$ individually). The update rule (Eq. 14) then follows by differentiating and solving for $\partial\mathcal{L}/\partial\bar{m}_i = 0$, with the update rule for $\bar{p}_j$ being symmetric to $\bar{m}_i$.

B.2 Quantile Selection

In (one-sided) conformalized quantile regression, using the same target quantile as the desired miscoverage ratio (i.e. $\xi = \varepsilon$) can be significantly less than optimal. Figure 8 shows an illustrative example with replicates trained on 50% of the dataset for prediction without interference, with a miscoverage ratio of $\varepsilon = 0.05$. For each replicate, the optimal quantile regression target quantile which results in the narrowest overprovisioning margin after calibration is between 80% and 90%.

In our experiments, we also observe that small changes in ξ have a larger impact on the resulting overprovisioning margin closer to $\xi = 100\%$. As such, we train target quantiles of $\{50\%, 60\%, 70\%, 80\%, 90\%, 95\%, 98\%, 99\%\}$, with more target quantiles close to 100%.

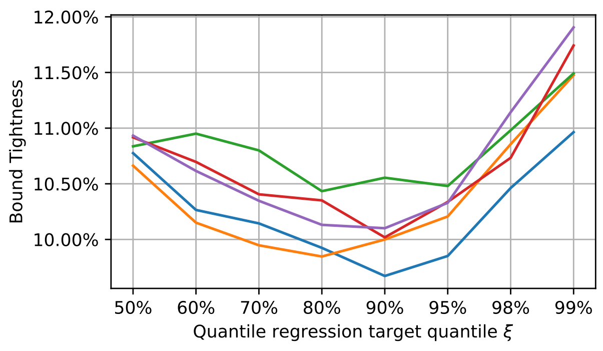


Figure 8. Bound tightness (overprovisioning margin) resulting from different quantile regression target quantiles ξ for 5 different replicates. The optimal target quantile is between 80% and 90%, compared to the calibration target miscoverage ratio of 95%.

B.3 Model Training

Multi-objective Optimization Pitot uses several different optimization objectives:

- Interference mode: in order to balance the influence of prediction with and without interference and better utilize GPU acceleration (Appendix B.3), each interference mode (without interference and with 2, 3, and 4 simultaneously running workloads) is treated as a different objective.
- Quantile regression: for each interference mode, each target quantile is also a different optimization objective (Section 3.5).

In order to define a single optimization objective for gradient descent, we assign a weight to each objective:

- To account for the increased difficulty and randomness of interference, and thus the "higher quality" of data collected without interfering workloads, we assign a higher weight to prediction without interference (Appendix D.2).
- Each quantile regression output is given equal weight.

Training Details Pitot (and all of our baselines) were trained using the AdaMax optimizer (i.e. the l_∞ variant of Adam) with default hyperparameters (learning rate = 0.001, $\beta_1 = 0.9, \beta_2 = 0.999$) and a batch size of 2048 (split equally across non-interference, 2, 3, and 4-way interference objectives).

Each model was trained for 20,000 steps, which we found was enough for convergence in all cases. During training, we evaluated each model every 200 steps, and returned the checkpoint which had the lowest validation loss for testing.

Implementation Our algorithm is implemented in JAX (Bradbury et al., 2018). While our dataset contains many

Table 1. Glossary of notation used in this paper.

Symbol	Description	Notes
N_w, N_p	Unique workloads, platforms	$N_w = 249, N_p = 231$ in our dataset.
i, j	Workload, platform index	$1 \leq i \leq N_w$ and $1 \leq j \leq N_p$.
k	Set of interfering workloads	$\forall l \in k : 1 \leq l \leq N_w$. We sometimes abuse notation and use k as an index when k is a singleton.
$\mathcal{A}$	Dataset	Contains all observed (workload, platform, interference) tuples.
C^*_{ijk}	Actual runtime	
$\hat{C}_{ijk}$	Predicted runtime	Non-interference-aware predictions are abbreviated $\hat{C}_{ij}$.
$\bar{C}_{ij}$	Baseline prediction	Linear scaling baseline predicted runtime.
$\bar{w}_i, \bar{p}_j$	Baseline parameters	Log workload "difficulty" and platform "speed"
$\boldsymbol{x}_w$	Workload features	Log opcode counts.
$\boldsymbol{x}_p$	Platform features	CPU architecture, WebAssembly runtime information.
$\boldsymbol{w}_i, \boldsymbol{p}_j$	Learned embeddings	Dimensionality $r = 128$ workload, platform embeddings.
f_w, f_p	Embedding networks	Generates workload and platform embeddings
$\boldsymbol{\theta}_w, \boldsymbol{\theta}_p$	Embedding network weights	
$\boldsymbol{\varphi}^{(i)}_w, \boldsymbol{\varphi}^{(j)}_p$	Extra learned features	Has dimension $q = 1$.
$\boldsymbol{F}_j$	Interference matrix	We never *explicitly* compute $\boldsymbol{F}_j$.
$\boldsymbol{v}^{(t)}_s$	Interference susceptibility	Associated with a platform j and interference type t ($1 \leq t \leq s = 2$).
$\boldsymbol{v}^{(t)}_g$	Interference magnitude	Associated with a platform j and interference type t ($1 \leq t \leq s = 2$).
α	Interference activation	Activation function (Leaky ReLU) for multiple interfering workloads
ε	Target miscoverage rate	For conformal regression.
ξ	Target quantile	For quantile regression.

($N = 410970$) data points, each data point uses a small amount of memory, consisting only of platform, workload, and interfering workload indices, which point to shared platform ($N_p = 231$) and workload ($N_w = 249$) features. As such, we make a number of optimizations in our implementation which target this data regime:

- All data is stored in GPU memory at all times.
- Since our batch size (2048) is relatively large compared to our matrix (231 platforms, 249 workloads), we always compute all module and device embeddings $\boldsymbol{w}_i$ and $\boldsymbol{p}_j$, and index the ones that we need.

As an additional optimization, when training on data with interference, each additional source of interference adds additional nodes to the compute graph that are only used when interference is present. As such, we separately sample fixed-sized batches of 512 samples from each degree of interference instead of randomly drawing a batch of 2048 data points from the entire dataset at once in order to maximize GPU parallelism (i.e. allowing all operations to have a fixed dimension across batches) while avoiding wasted compute (i.e. if the results of unused computations are ignored).

With these optimizations, our method is very cheap to train, and has a median per-replicate training time of 11.5 seconds (or 12.1 seconds for the multi-objective quantile regression version) on a RTX 4090 GPU across 45 different runs.

B.4 Baseline Details

Common settings To make our baselines more competitive, each baseline was also trained to predict runtime in the log domain. The baselines were also trained in the same way as Pitot (20,000 steps with batch size 2048, etc).

Matrix Factorization Our matrix factorization baseline uses the same number of features ($r = 32$) as we found to be optimal for Pitot, and can be thought of as Pitot without our log-residual objective, workload or platform features, interference modeling, and uncertainty quantification method.

Neural Network The neural network baseline uses two neural networks with two hidden layers of 256 units and the GELU activation (twice as large as Pitot):

(1) The "base" network concatenates the workload and platform features of each data point as input, and predicts a single interference-blind runtime which is used on workloads running in isolation.

(2) The "interference" network concatenates two sets of workload features (current workload and interfering workload) and one set of platform features as an input, and predicts an interference multiplier.

The interference network computes an interference multiplier for each interfering workload; the base prediction is multiplied by each interference multiplier to generate the final interference-aware runtime prediction.

Attention The attention network uses a (single-headed) attention mechanism to predict the interference generated by a set of interfering workloads instead of assuming a simple multiplicative relationship between pairs of workloads. Like the neural network baseline, a neural network

Table 2. Cluster devices with the CPU vendor, model, and microarchitecture.

Model	CPU		Architecture
NUC 8	Intel	i7-8650U	Skylake
NUC 4	Intel	i3-4010U	Haswell
Generic ITX	Intel	i7-4770TE	Haswell
Compute Stick	Intel	x5-Z8330	Silvermont
NUC 11	Intel	i5-1145G7	Tiger Lake
NUC 11	Intel	i7-1165G7	Tiger Lake
Mini PC	Intel	N4020	Goldmont Plus
EliteDesk 805 G8	AMD	R5-5650G	Zen 3
Mini PC	AMD	R5-4500U	Zen 2
Mini PC	AMD	R3-3200U	Zen 1
Mini PC	AMD	A6-1450	Jaguar
RPi 4 Rev 1.2	Broadcom	BCM2711	Cortex-A72
RPi 3B+ Rev 1.3	Broadcom	BCM2837B0	Cortex-A53
Banana Pi M5	Amlogic	S905X3	Cortex-A55
Le Potato	Amlogic	S905X	Cortex-A53
Odroid C4	Amlogic	S905X3	Cortex-A55
RockPro64	RockChip	RK3399	Cortex-A72
Rock Pi 4b	RockChip	RK3399	Cortex-A72
Renegade	RockChip	RK3328	Cortex-A53
Orange Pi 3	Allwinner	H6	Cortex-A53
Starfive VF2	SiFive	U74	RISC-V
Nucleo-F767ZI	STMicro	STM23F767ZI	Cortex-M7

with two hidden layers of 256 units and GELU activation is used to generate a "base" prediction. To add an attention mechanism, this network also generates a query vector.

To model interference, a second embedding network (also with two hidden layers of 256 units and GELU activation) generates key and value vectors. The query vector is used to index the weight of the value vector across each interfering workload according $\langle \text{key}, \text{query} \rangle$ product, and an output network with a single hidden layer produces the final interference multiplier. We tuned the key/query vector dimension and output network hidden layer size, arriving at a vector dimension of 8 and an output hidden layer of 32.

C Dataset

Using our heterogeneous cluster (Figure 3), we collected a large dataset which includes a range of different workloads, compute platforms, and varying levels of interference. In this section, we describe the workloads, compute platforms, data collection procedures, and collected data.

C.1 Platforms

Each platform in our dataset consists of a (device, runtime) tuple. While datasets could conceivably include additional platform dimensions such as the operating system, scheduler, and CPU frequency governor, we chose to study hardware devices and WebAssembly runtimes since these are most relevant to the WebAssembly community.

Devices Our cluster (shown in Figure 3) includes 24 devices from 9 different vendors (Intel, AMD, SiFive, Broadcom, NXP, Amlogic, RockChip, Allwinner, STMicroelectronics) across 14 different microarchitectures (Table 2). Notable devices include the RISC-V-based Starfive VF2 and the Cortex-M7-based Nucleo-F767ZI.

Runtimes For each device, we ran 5 different WebAssembly runtimes with a total of 10 different configurations, including interpreted, ahead-of-time compiled (AOT), and just-in-time compiled (JIT) runtimes (Table 3). Each runtime was run on each device except where not supported: only AOT WAMR runs on the cortex M7, and only WAMR and wasm3 run on the RISC-V device. Ahead-of-time-compiled WAMR was also excluded from Cortex A-72-based platforms due to a code generation bug which can randomly cause illegal instruction errors.

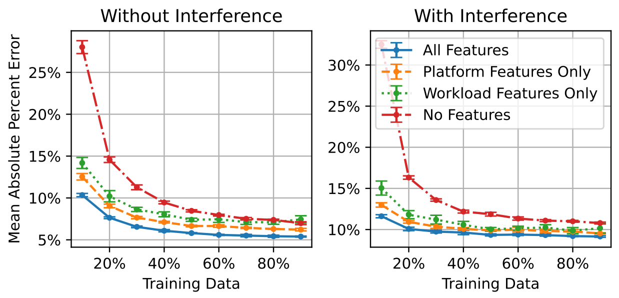
Table 3. WebAssembly runtimes used. WAMR (the *WebAssembly Micro Runtime*) is also commonly referred to as "iwasm".

Runtime	Runtime Type
Wasm3	Interpreter
WAMR	Interpreter, LLVM AOT
WasmEdge	Interpreter
Wasmtime	Cranelift AOT, Cranelift JIT
Wasmer	Singlepass JIT, Cranelift JIT, Cranelift AOT, LLVM AOT

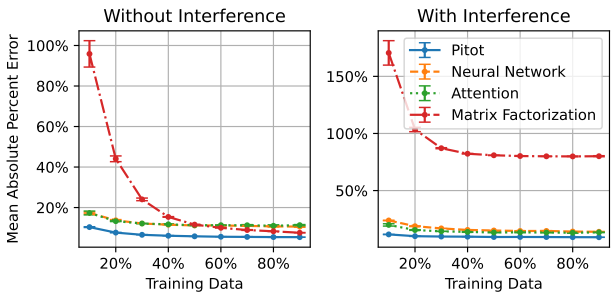
C.2 Side Information

Workload Features In order to collect the "opcode count" (the number of times each opcode was executed) for each workload, we instrumented the WebAssembly Micro Runtime (WAMR) fast interpreter (Xu et al., 2021) to increment an opcode counter table each time each instruction was executed. Due to several order-of-magnitude differences in opcode counts between short and long benchmarks as well as rare and common instructions, we transform the opcode counts by the log-frequency $f(n) = \log(n + 1)$ (so that $f(0) = 0$). We also exclude opcodes which are not used by any of the workloads from the dataset.

While it is possible to reduce this profiling overhead through an instrumentation-based opcode counting approach, profiling of any kind at this level of detail will be expensive relative to execution without any profiling. However, profiling does not need to be performed on the edge: opcode frequency does not depend on the underlying hardware and only needs to be performed once. As such, profiling can use a fast computer before a workload is to be deployed or



(a) Uncropped version of Figure 4b; removing both workload and platform features from Pitot leads to much higher error when only a small amount of data is observed.



(b) Uncropped version of Figure 6a; the Matrix Factorization baseline performs an order of magnitude worse with less training data and predicting interference (since it is not interference-aware).

Figure 9. Uncropped versions of figures where the y-axis was cropped for clarity.

is observed for the first time, and does not need to be run during deployment (in the case of edge orchestration) or on a highly-constrained candidate edge device (in the case of system design).

Platform Features In addition to a one-hot encoding of the WebAssembly runtime used, we recorded a number of features via linux `cpuinfo` and `meminfo`:

- CPU microarchitecture (e.g. `znver3`, `cortex-a72`, `tigerlake`), which is one-hot encoded.
- Nominal CPU Frequency (i.e. differently clocked CPUs with the same microarchitecture). Note that clock frequency governors (e.g. `ondemand`) may set the CPU frequency on-the-fly in a highly dynamic manner, which we cannot easily record.
- Memory architecture: L1d / L1i cache sizes, L2 size, L2 line size and associativity, L3 size, and main memory size. Cache sizes are passed as a log size, while line size and associativity are provided as one-hot features. Each cache feature is augmented with an indicator feature to account for cases where a given level in the memory hierarchy is not present (e.g. the ARM Cortex-A72 architecture does not have a L3 cache).

C.3 Collected Data

Benchmarks in Isolation We ran each benchmark on each of our (device, runtime) platforms where supported. In total, we collected 53,637 observations of valid (workload, platform) pairs, and recorded the wall clock execution time for each, averaged over up to 50 repeated executions over a maximum of 30 seconds. While we attempted to execute every possible (workload, platform) pair, some combinations resulted in errors or crashes, which we omit from the dataset. Notable omissions include some WebAssembly runtimes lacking full ARM and RISC-V support at present, interpreted runtimes struggling to complete large benchmarks before being timed out (especially on slower devices), and various implementation bugs on some combinations of runtimes, platforms, and benchmarks.

Interference Dataset To evaluate our interference model, we also ran up to 4 benchmarks simultaneously. Each benchmark was run continuously in a loop, resulting in random program alignments. In total, we collected 357,333 usable observations, which includes 98,957 observations with two simultaneously running workloads, 139,208 with three simultaneously running, and 119,168 with three simultaneously running.

During interference data collection, we ran 250 random sets of 2, 3, and 4 workloads on each platform (for a total of 750 sets). Each workload was run repeatedly for 30 seconds. If any of the workloads in a set crashed or otherwise terminated before the end of the 30-second period, that entire set was excluded. Workloads which timed out and failed to complete by the end of the 30-second period but did not crash were also excluded, though other simultaneously running workloads in that set were still included in the dataset since timed-out workloads still cause interference.

D Additional Results

D.1 Uncropped Figures

Figure 4b and figure 6a were cropped in the y-axis for clarity; we provide uncropped versions of these figures in Figure 9a and Figure 9b, respectively.

D.2 Hyperparameter Ablations

We conducted ablation studies on four key hyperparameters for our method, and ran an exponential spread of 5 different values for each (Figure 10). Our method is not sensitive to the parameters selected, and will perform close to optimally as long as the model embeddings have sufficient

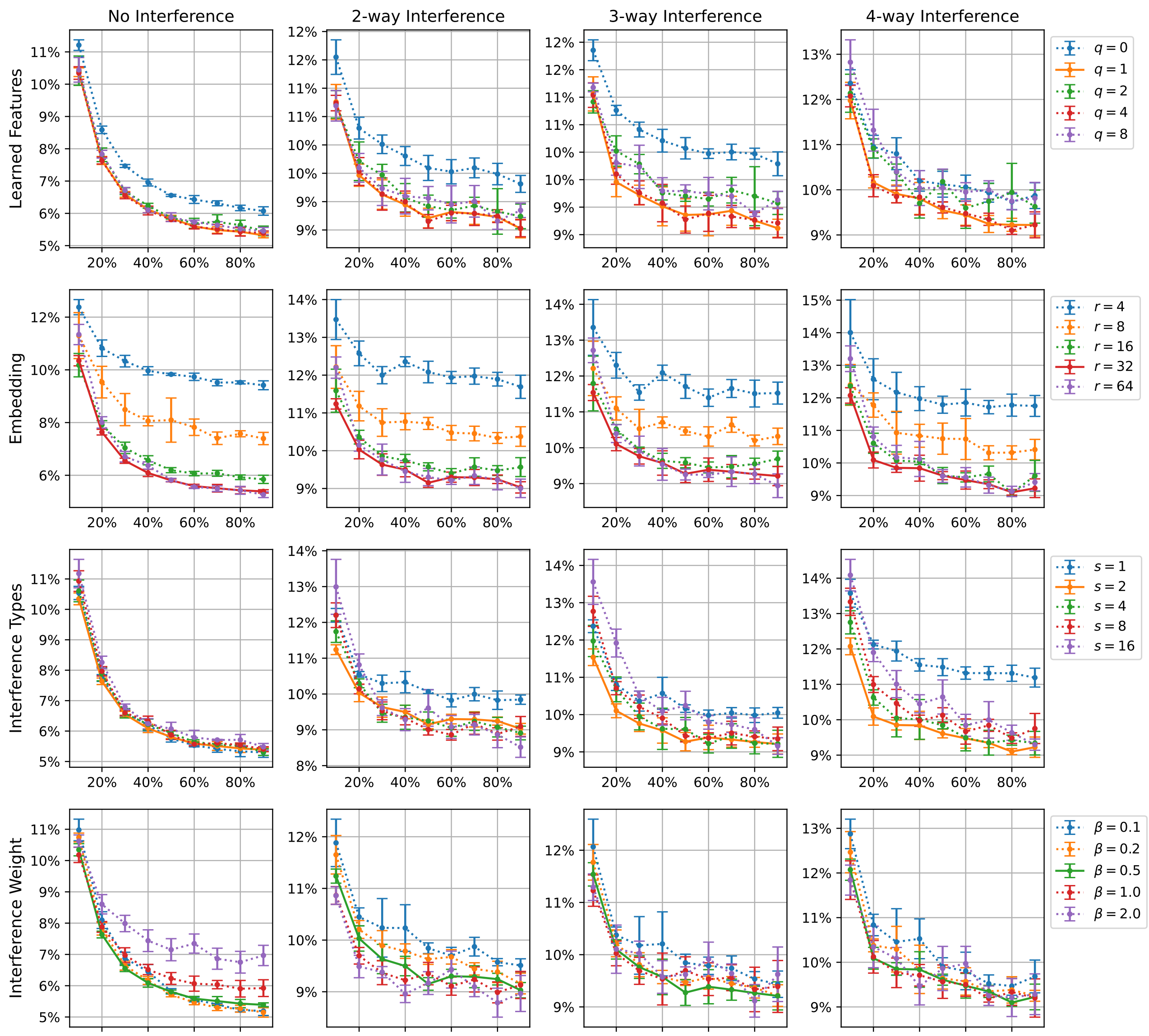
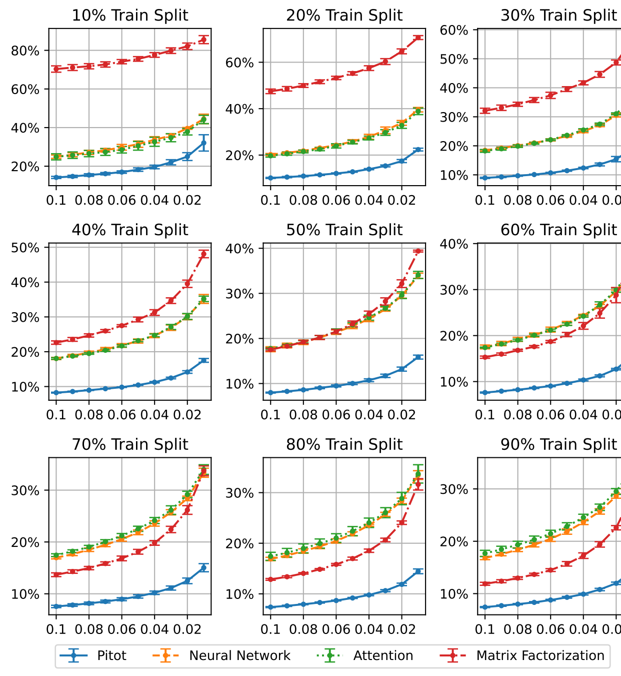


Figure 10. Hyperparameter ablations for the number of learned features, embedding dimension, interference types, and interference objective weight, with mean absolute percent error on the y-axis, and the proportion of observed data on the x-axis. We split our results in each column depending on the number of simultaneously running workloads due to the increased prediction error (and intrinsic problem difficulty) associated with more interfering workloads. In each plot, the solid line indicates the selected hyperparameter value; error bars indicate ± 2 standard errors.


(a) Bounds for prediction without interference

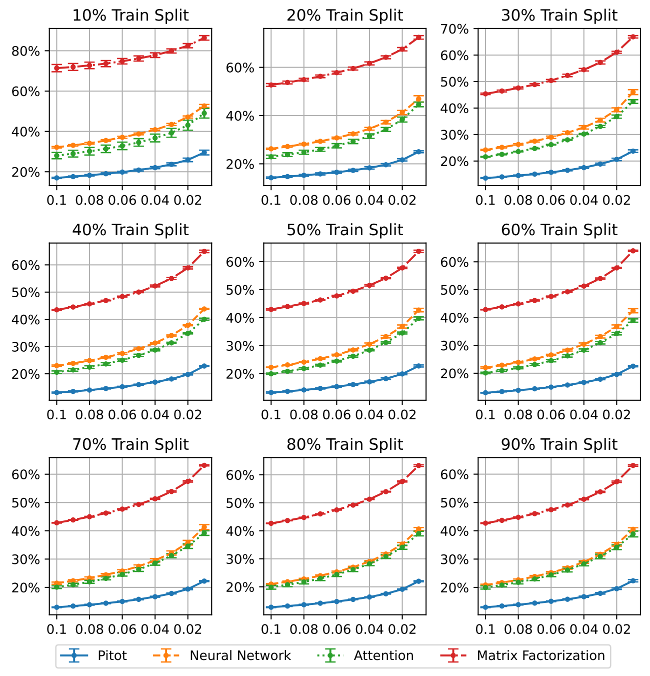

(b) Bounds for interference prediction

Figure 11. Full bound tightness comparison between Pitot and baselines for the conformal prediction task across varying amounts of training data; each plot shows the bound tightness (with ± 2 standard errors) for a given training split size and varying miscoverage rates.

dimensionality and thus representational power.

Number of Learned Features q Learned features in Pitot are feature vectors associated with each platform and workload, which are jointly trained with the embedding network parameters using gradient descent. There is a significant decrease in error in all categories after introducing just one additional learned feature, indicating the necessity of this aspect of Pitot. However, adding additional features does not make a significant impact on model performance. We select $q = 1$ for our experiments; in general, any small value of q should be sufficient, though higher q may be beneficial for larger datasets.

Embedding Dimension r The embedding dimension is the output dimensionality of Pitot's workload and platform embedding networks, and acts as the rank constraint for matrix factorization. In our ablations, we can see a significant improvement in error as the dimensionality increases up to 32 dimensions, after which the error no longer improves. We select $r = 32$; in general, r only needs to be sufficiently large, with no significant prediction error downside to an overly large r.

Interference Types s We find that using $s = 2$ interference types is sufficient to obtain optimal performance. Our model is slightly sensitive to s, with a slight increase in error as s increases for some evaluation settings.

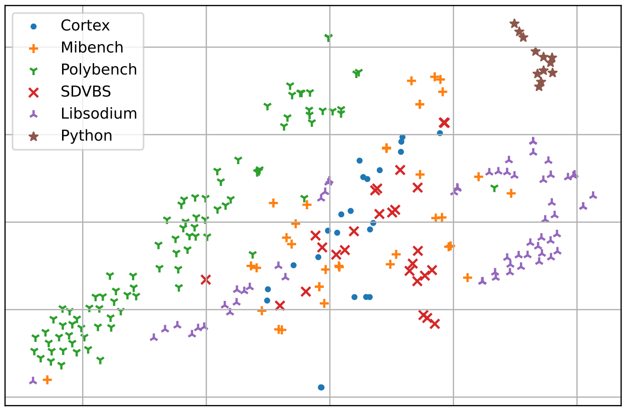
Note that the choice of s does not impact the error of Pitot when predicting the runtime of workloads without any background interference, which is expected, since the interference susceptibility and magnitude embeddings $\boldsymbol{v}_s, \boldsymbol{v}_g$ are ignored when no interference is present.

Interference Weight β Since Pitot solves a multi-objective optimization problem (even before considering quantile regression), the weight of each objective can impact its error. We assign a constant weight of 1.0 to objectives predicting runtime without interfering workloads, and a weight of β to interference prediction, split equally across 2, 3, and 4-way interference.

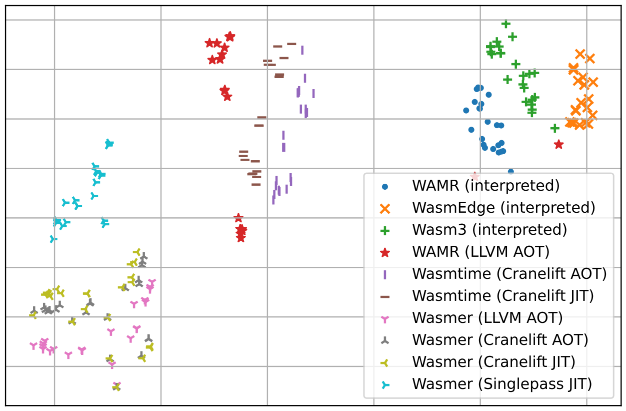
Increasing the interference objective weight β reduces interference prediction error at the cost of increasing error for prediction without interference, with a similar effect in reverse. We choose $\beta = 0.5$ as a compromise which does not significantly increase the prediction error for either objective.

D.3 Bound Tightness Comparisons

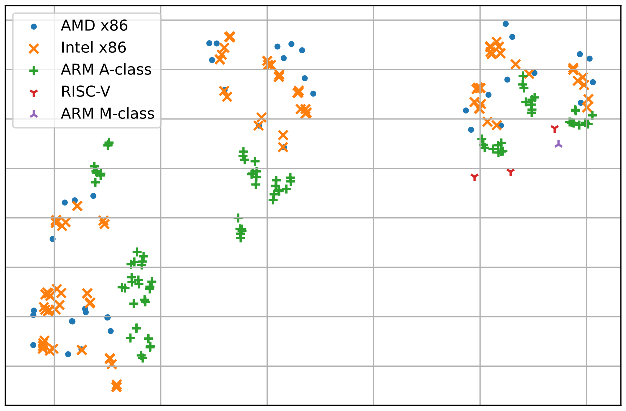
Figure 11 provides an expanded version of Figure 6b showing miscoverage rate-bound tightness curves for each training split size. Pitot performs far better than all of our base-



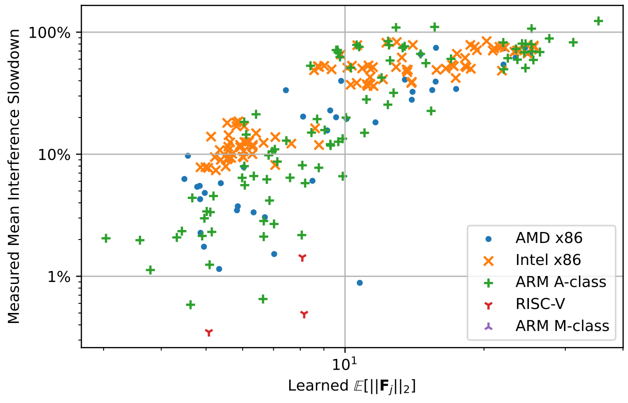
(a) Workload features (t-SNE) sorted by benchmark suite



(b) Platform features (t-SNE) sorted by WebAssembly runtime



(c) Platform features (t-SNE) sorted by CPU microarchitecture



(d) Measured and learned interference

Figure 12. Full-size visualizations of Pitot's learned embeddings. Figure 12a-12c show t-SNE embeddings of the learned workload and platform features, while Figure 12d shows the l_2 norm of the learned interference matrix compared with the observed mean interference, sorted by CPU microarchitecture.

lines in each setting, while the attention baseline performs slightly better on predicting interference as the neural network baseline. The matrix factorization baseline performs far worse in most settings, except when predicting runtime without interference when a large proportion of the dataset is observed.

D.4 Embedding Visualizations

Unlike black-box models, Pitot learns embedding vectors for each workload and platform which can be interpreted, and potentially used as inputs for downstream tasks such as anomaly detection. To demonstrate the information value of these learned embeddings, we visualized them by projecting them to two dimensions (Fig. 12a-12c). We also analyze our learned interference representation as a sanity check on our interference model (Fig. 12d).

Workload Features To analyze embedding features (Fig. 12a), we project them to 2 dimensions using a t-distributed stochastic neighbor embedding (t-SNE), which maps similar workloads to nearby locations in a 2-dimensional scatter plot, though distances and units do not have any particular meaning. Relatively homogenous benchmark suites such as Polybench, Libsodium, and our Python benchmarks form clear clusters, while more diverse benchmark suites (Mibench, SDVBS, Cortex) are largely mixed.

Platform Features We also projected platform features using a 2-dimension t-SNE. Sorting platforms by WebAssembly runtime (Fig. 12b), we see that most runtimes form clear clusters. Notably, the three interpreted runtimes in our dataset (WAMR, WasmEdge, Wasm3) form nearbly clusters, while different configurations of Wasmtime and Wasmer are also respectively clustered together.

Alternatively, organizing platform embeddings by CPU microarchitecture (Fig. 12c), we also see clear clusters of each microarchitecture category within the larger clusters associated with each type of runtime.

Interference Matrix The interference matrix $\boldsymbol{F}_j$ allows us to gain insight into the interference characteristics of each platform. Specifically, consider the spectral norm $||\boldsymbol{F}_j||_2$,

$$||\boldsymbol{F}_j||_2^2 = \sup_{||w_i||_2=1, ||w_k||_2=1} \boldsymbol{w}_i^T \boldsymbol{F}_j \boldsymbol{w}_k. \tag{15}$$

This can be interpreted as the maximum possible interference between two workloads $\boldsymbol{w}_i, \boldsymbol{w}_k$. Figure 12d shows the spectral norm of F_j (trained on the 90% data split and average over the 5 replicates) plotted against mean interference on each platform. We observe a positive correlation between $||\boldsymbol{F}_j||_2$ and measured interference on each device, as we would expect from our interpretation.