A Artifact Appendix

Our artifact implements the interference-aware conformal prediction algorithm described in this paper, and can be found at https://github.com/wiseLabCMU/pitot*. The implementation is GPU-accelerated and parallelized using Jax, and includes a poetry lock file to install dependencies. The artifact provided includes code to (1) generate train/val/test splits, (2) train the experiments described in the paper, (3) summarize these experiments into key performance metrics, and (4) reproduce the figures shown in the paper. We also include our dataset, which is packaged with the repository. Finally, in addition to an archived copy of our repository, our splits, results, and summary metrics are also archived at* https://zenodo.org/records/14977004*.*

A.1 Artifact check-list (meta-information)

- **Algorithm:** matrix completion for program runtime prediction + extension for interference + extension for conformal prediction
- **Run-time environment:** linux, cuda, python/conda, poetry
- **Hardware:** Nvidia GPU, 24GB VRAM
- **How much disk space required (approximately):** <5GB and a conda environment
- **How much time is needed to prepare workflow (approximately)?:** $<$ 10 minutes
- **How much time is needed to complete experiments (approximately)?:** $\approx$ 6-12 hours using a RTX 4090
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT
- **Data licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** 10.5281/zenodo.14977004

A.2 Description

How to access. Our code, data, and installation instructions can be found at our project repository: https://github.com/wiseLabCMU/pitot. Minimal disk space is required for the repository ($<$ 5GB after generating all outputs), though considerably more may be consumed by the python environment after installing dependencies if they are not already present.

Hardware dependencies. While our implementation should run on any platform which is compatible with Jax, our implementation is tuned for (and tested on) 24GB Nvidia GPUs, e.g. the RTX 3090 or 4090.

Software dependencies. Our implementation assumes a linux operating system, a conda environment, and poetry for installing from the provided lockfile.

Datasets Our dataset consists of aggregated benchmarking data collected from our heterogeneous cluster; all benchmark data is included with the repository.

The benchmarks used to collect our dataset are also open source, and can be found at https://github.com/silverLineFramework/benchmarks.

A.3 Installation

All dependencies can be installed with `conda` and `poetry`; see the included README for details.

A.4 Experiment workflow

The experiment workflow consists of three stages: training (and split generation), evaluation, and analysis. The exact commands used for each phase can be found in the provided `Makefile`.

A.5 Evaluation and expected result

The output of each stage (splits, results, summary) can be found on Zenodo (https://zenodo.org/records/14977004); note that the results may not necessarily be numerically identical, for example due to floating point discrepancies.

B Glossary of Notation

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

C 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.

C.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*

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

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	$1 \le i \le N_w$ and $1 \le j \le N_p$.
k	Set of interfering workloads	$\forall l \in k : 1 \le l \le N_w$. 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	
$\boldsymbol{\varphi}_w^{(i)}, \boldsymbol{\varphi}_p^{(j)}$	Extra learned features	Has dimension $q = 1$.
$\boldsymbol{F}_j$	Interference matrix	We never *explicitly* compute $\boldsymbol{F}_j$.
$\boldsymbol{v}_s^{(t)}$	Interference susceptibility	Associated with a platform j and interference type t ($1 \le t \le s = 2$).
$\boldsymbol{v}_g^{(t)}$	Interference magnitude	Associated with a platform j and interference type t ($1 \le t \le s = 2$).
α	Interference activation	Activation function (Leaky ReLU) for multiple interfering workloads
ε	Target miscoverage rate	For conformal regression.
ξ	Target quantile	For quantile regression.

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 $\bar{m}_i$.

C.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%.

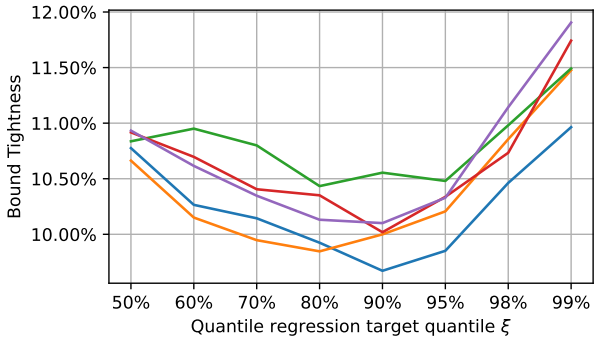


Figure 8. Bound tightness (overprovisioning margin) resulting from different quantile regression target quantiles ξ for 5 different replicates. The optimal target quantile is between 80% and 90%, compared to the calibration target miscoverage ratio of 95%.

C.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 C.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 (Ap-

pendix E.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 ($N = 410970$) 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.

C.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 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.

D 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.

D.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

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

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

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.

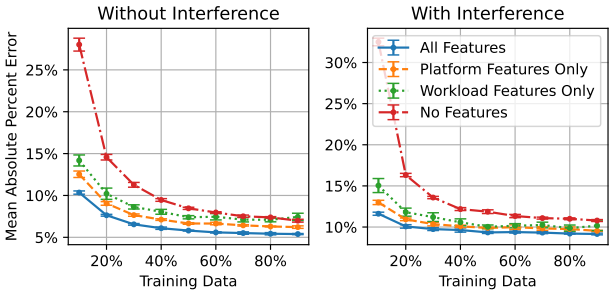
D.2 Side Information

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.

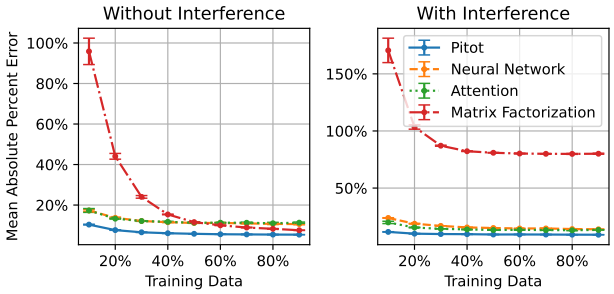
While it is possible to reduce this profiling overhead through an instrumentation-based opcode counting approach, profiling of any kind at this level of detail will be expensive relative to execution without any profiling. However, profiling does not need to be performed on the edge: opcode frequency 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 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).



(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.

D.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.

E Additional Results

E.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.

E.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 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.

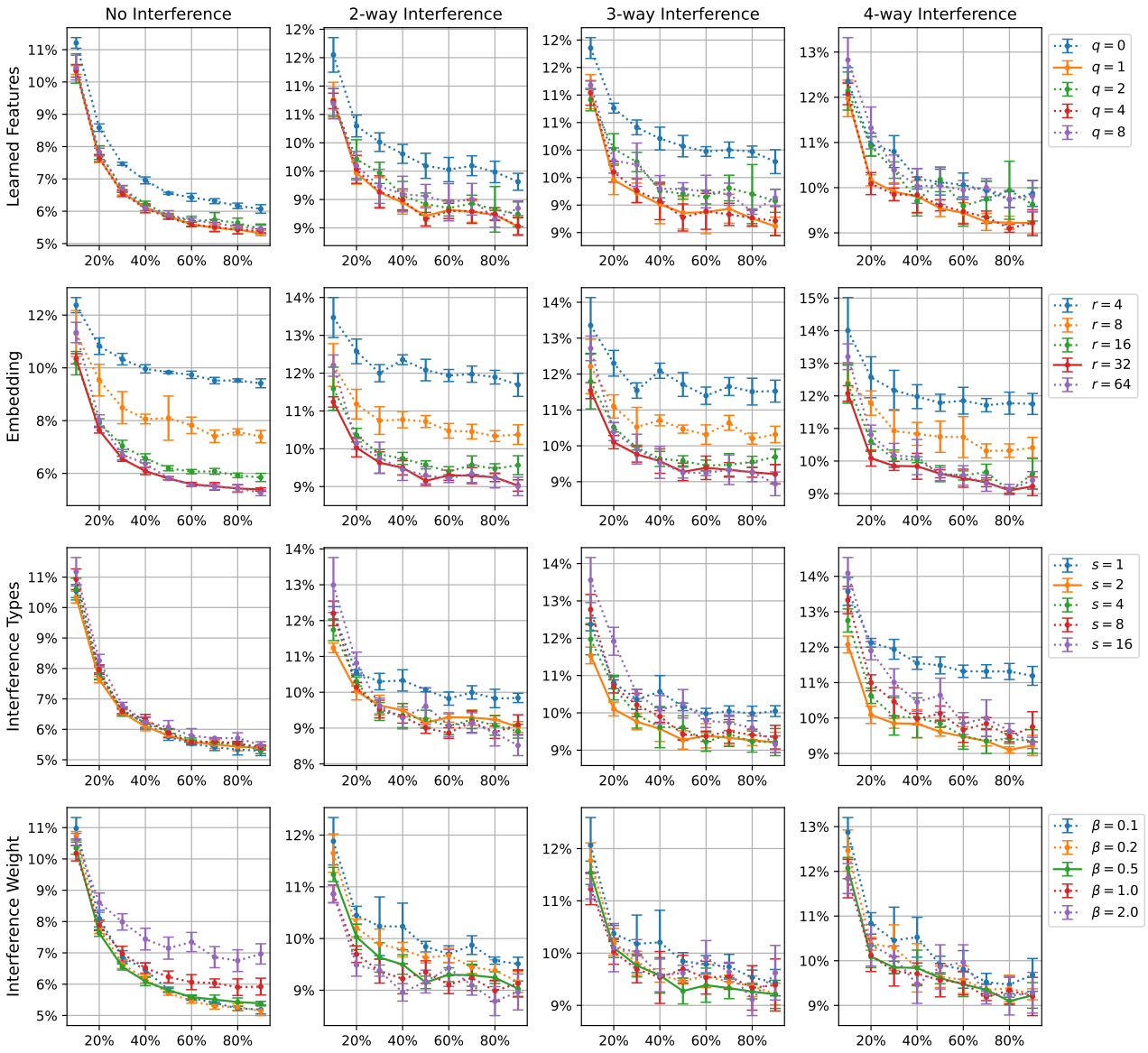
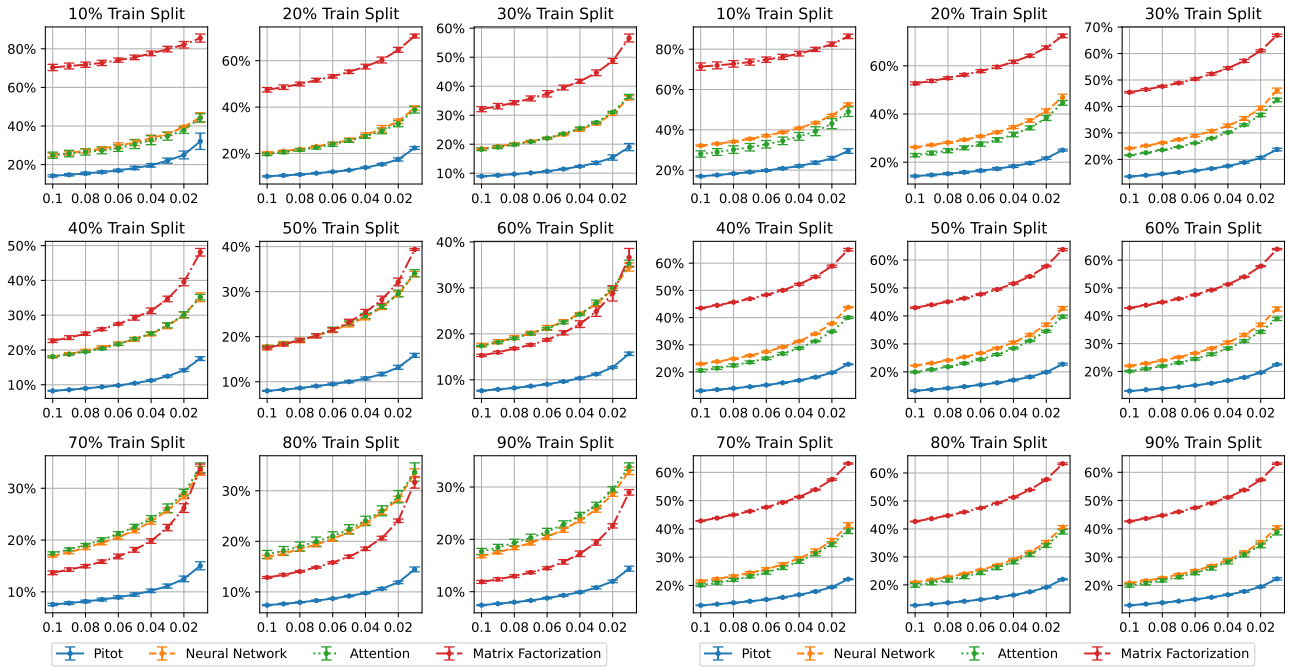


Figure 10. Hyperparameter ablations for the number of learned features, embedding dimension, interference types, and interference objective weight, with mean absolute percent error on the y-axis, and the proportion of observed data on the x-axis. We split our results in each column depending on the number 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.

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.

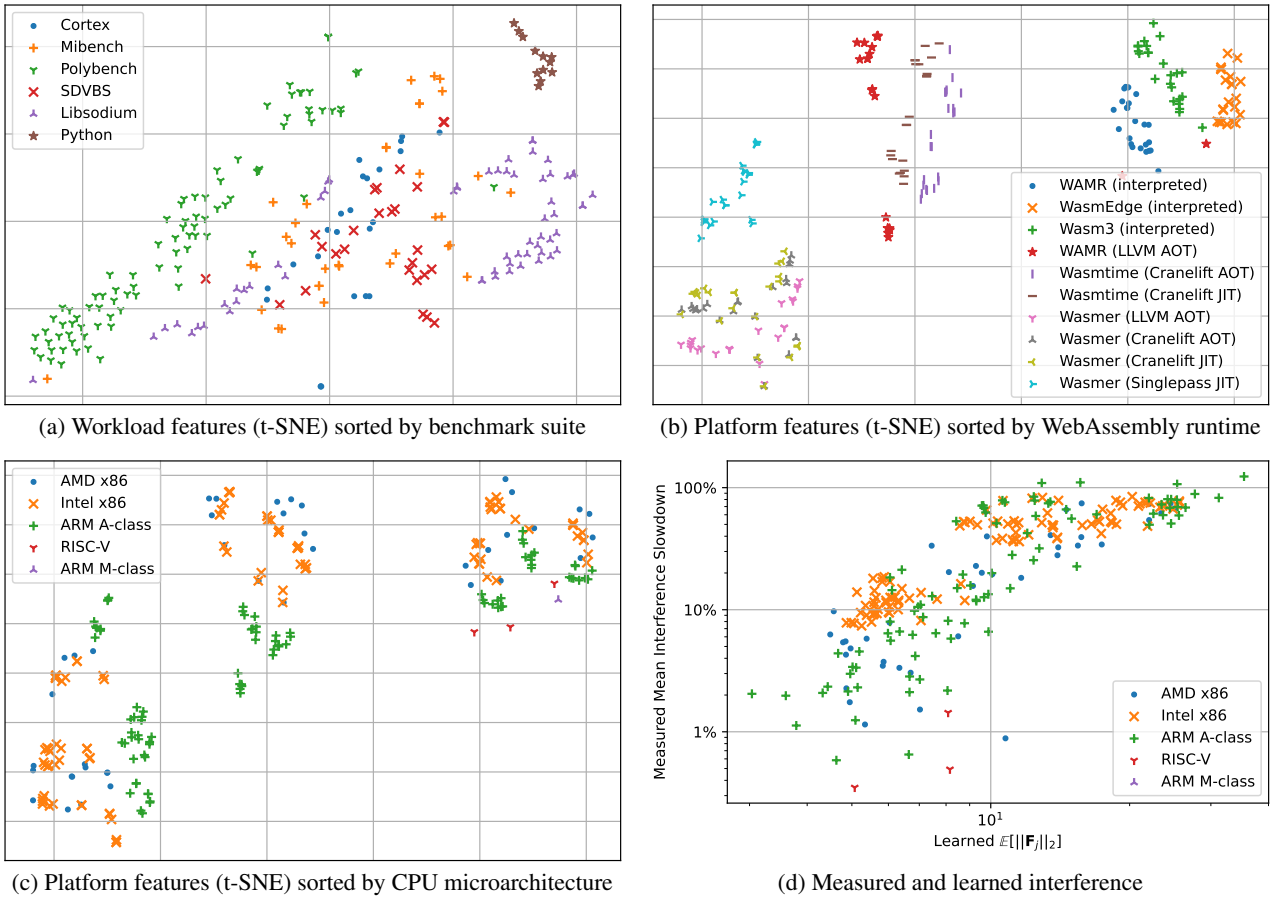
E.3 Bound Tightness Comparisons

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 baselines 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.

E.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



(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.

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.