Reasoning about Configurable System Performance through the lens of Causality

Shahriar Iqbal
University of South Carolina
miqbal@email.sc.edu

Rahul Krishna
Columbia University
rahul.krishna@columbia.edu

Mohammad Ali Javidian
Purdue University
mjavidia@purdue.edu

Baishakhi Ray
Columbia University
rayb@cs.columbia.edu

Pooyan Jamshidi
University of South Carolina
pjamshid@cse.sc.edu

Abstract

Modern computer systems are highly configurable. They often are composed of heterogeneous components—each component consists of numerous configurations, giving a total variability space sometimes larger than the number of atoms in the universe. The performance of a system configured with different configuration options can widely vary. Thus, given the vast configuration space, understanding and reasoning about the performance behavior of such systems become challenging. These configuration options interact with each other within and across the system stack, and such interactions typically vary in different deployment environments or workload conditions. So, it becomes almost impossible to track down configuration options that should be set to different values to improve performance if the system's performance shows wide variability during operation time. As a result, existing performance models that rely on predictive machine learning models suffer from (i) *high cost*: given a deployment environment, regression-based performance models require a large number of configuration samples for accurate predictions, and more importantly, (ii) *unreliable predictions*: even if they predict performance for the environment where configurations are measured, since they may infer correlations as causation, they typically do not transfer well for predicting system performance behavior in a new environment (e.g., change of hardware from the canary environment to production).

The main problem we address here is to understand *why* the performance degradation is happening and *reason* based on a reliable model to improve it. To this end, this paper proposes a new methodology, called Unicorn, which initially learns a Causal Performance Model to reliably *capture intricate interactions* between options across software-hardware stack by tracing system-level performance events across the stack (hardware, software, cache, and tracepoint). Then, it uses them to *explain* how such interactions impact the variation in performance objectives causally. Given a limited sampling budget, Unicorn iteratively updates the learned performance model by estimating the causal effects of configuration options to performance objectives, then selecting the highest-impact options to adjust in order to address performance issues by improving the performance objective of interest without deteriorating other objectives in debugging task or recommend a near-optimal configuration.

We evaluated Unicorn on six highly configurable systems, including three on-device machine learning systems, a video encoder, a database, and a data analytics pipeline. In addition, we compared the results with state-of-the-art configuration optimization and debugging methods. The experimental results indicate that Unicorn can find effective repairs for performance faults and find configurations with near-optimal performance. Furthermore, unlike the existing methods, the learned causal performance models in Unicorn reliably predict performance for new environments where it has not been used during the learning process.

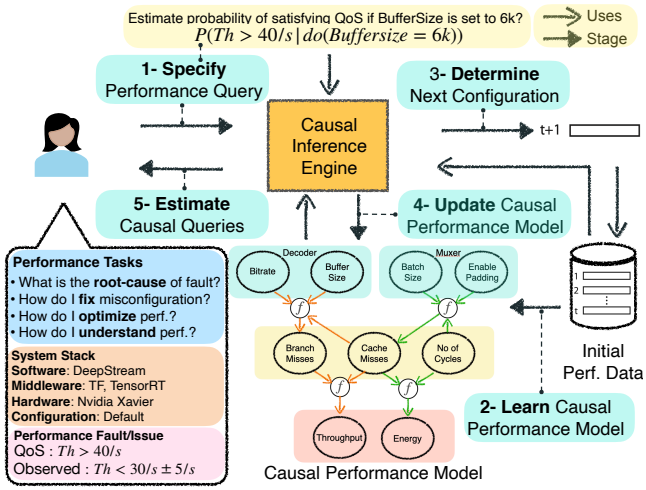


Figure 1. Overview of Unicorn.

1 Introduction

Modern computer systems are typically composed of multiple components, each component has many configuration options, and they can seamlessly be deployed on various hardware platforms. The configuration space of highly configurable systems is combinatorially large with 100s if not 1000s of software and hardware configuration options that

interact non-trivially with one another [37, 50, 101]. Developers of individual components typically have a very local, and thus limited, understanding regarding the performance behavior of such systems. Developers and users of the final system are often overwhelmed with the complexity of composing and configuring components, and thus, configuring these systems to achieve specific performance goals is challenging and error-prone.

Misconfigurations are typically caused by interactions between software and hardware, resulting in *non-functional faults*[1], i.e., faults in *non-functional* system properties such as latency and energy consumption. These non-functional faults—unlike regular software bugs—do not cause the system to crash or exhibit an obvious misbehavior [73, 82, 96]. Instead, misconfigured systems remain operational while being compromised, resulting in severe performance degradation, for example, in execution time or energy consumption [14, 68, 72, 83].

Misconfigurations are not only causing major issues for internet-scale cloud systems [1], but also, they exist for many types of systems, including embedded systems. For example, a developer on NVIDIA's developer forum complained that *"I have a complicated system composed of multiple components running on Nvidia Nano and using several sensors and I observed several performance issues. [4]."* In another instance, a more experienced developer asks *"I'm quite upset with CPU usage on Jetson TX2, while running TCP/IP upload test program" [5].* After struggling in fixing the issues over the span of several days, the developer conclude: *there is a lot of knowledge required to optimize network stack and measure CPU load correctly. I tried to play with every configuration option explained in the kernel documents. I was able to reduce CPU load (mpstat) up to 77-83% IDLE with full CPU / max clock during.*" In addition, they would like to *understand* the impact of configuration options and their interactions: *"What is the effect of swap memory on increasing throughput? [2]".*

Existing works. Understanding the performance behavior of configurable systems can enable (i) performance debugging [33, 89], (ii) performance tuning [41, 44, 45, 70, 71, 76, 94, 98, 103], and (iii) architecture adaptation [6, 24, 25, 29, 43, 52, 55, 58]. A common strategy to build performance influence models in which regression-based models, of the form $f(c) = \beta_0 + \Sigma_i \phi(o_i) + \Sigma_{i..j} \phi(o_i..o_j)$, are used to build a model that explains the influence of individual options and their interactions [35, 80, 89, 98]. These approaches are adept at inferring the correlations between certain configuration options and performance objectives, however, they suffer from several issues: (i) the models require many performance measurements to provide reliable predictions, (ii) the models are not transferable for predicting performance for different environment, and (iii) the models provide incorrect explanations regarding important options and interactions that 'causes' performance variations.

[1]we use *non-functional faults* and *performance faults* interchangeably for severe performance degradation (below or above a user-specified threshold) that are caused by certain type of misconfigurations, also known as specious configuration [46].

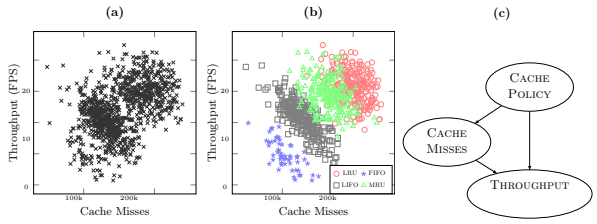


Figure 2. Simple example showing the effectiveness of causal reasoning in explaining the increase of Throughput with the decrease of Cache Misses. Observational data (Fig. 2a) (incorrectly) shows that as increase in Cache Misses leads to high throughput. Fig. 2c captures causal dependencies between different configuration option and system throughput. Thus, incorporating Cache Policy as a confounder correctly shows increase of Cache Misses correspond to decrease in throughput (Fig. 2b).

Our approach. Based on the intuition and several experimental evidence presented in the following section, this paper proposes Unicorn–a methodology that enables reasoning about configurable system performance with causal inference and counterfactual reasoning. As depicted in Fig. 1, Unicorn first recovers the underlying causal structure (a model that consists of variables related to system performance across stack) by measuring the system variables under different configurations. The causal performance model allows users to (a) identify the root causes of performance faults, (b) estimate the effects of various configurable parameters on the performance objectives, and (c) prescribe candidate configurations to fix the performance fault or optimize system performance.

Contributions. Our contributions are as follows.

- We propose Unicorn, a novel approach that instead of relying on correlations between system variables and performance objectives, constructs a causal performance model and enable reasoning about system performance in a reliable fashion using the mathematics of causality.
- We evaluate Unicorn in terms of its *effectiveness, transferability*, and *scalability* and compared with state-of-the-art performance debugging and optimization using six real-world highly configurable systems (three machine learning systems, a video analytics pipeline, a video encoder, and a database system) deployed on 3 architecturally different NVIDIA Jetson devices.
- We offer a manually curated performance fault dataset (called Jetson Faults) and accompanying code required to reproduce our findings at https://git.io/JtFNG.

2 Causal Performance Modeling and Analyses: Motivating Scenarios

In this section, we, first, point out reliability and stability issues with existing performance analyses practices via a simple as well as a large-scale real scenario. We then define a new abstraction that enables us to perform causal reasoning in systems. We, finally, show the effectiveness of causality vs correlation for performance tasks that address the issues.

A simple scenario: Fig. 2 shows a simple scenario where the observational data indicates that throughput is positively correlated with increased Cache Misses (as in Fig. 2 a). A simple ML model built on this data will predict with high confidence that larger Cache Misses leads to higher throughput—this is misleading as higher Cache Misses should, in theory, lower throughput. However, segregating the same data on Cache Policy (as in Fig. 2 b) reveals that within each group of Cache Misses, as Cache Misses increases, the throughput decreases. One would expect such behavior as the more Cache Misses the higher number of access to external memory and therefore, the throughput would be expected to decrease. The system resource manager may change the cache policy based on some criteria; this means that for the same number of cache misses, the throughput may be lower or higher, however, in all policies, the increases of cache misses result in a decrease in throughput. Thus, Cache Policy acts as a confounder that explains the relation between Cache Misses and throughout, which a correlation-based model will not be able to capture. In contrast, a causal performance model, as shown in Fig. 2 c, finds the relation between Cache misses, Cache Policy, and throughput and thus can reason about the observed behavior correctly.

In reality, performance analysis and debugging of heterogeneous multi-component systems is non-trivial, and often compared with finding the needle in the haystack [102]. In particular, there are two main challenges: (i) end-to-end performance analysis is not possible by reasoning about individual components in isolation, (ii) intra-component analysis may not be sufficient as severe performance bottlenecks may happen due to suboptimal interactions of configuration options across components. Next, we use a highly configurable multi-stack system to motivate why causal reasoning is a better choice for understanding the performance behavior of complex systems and then illustrate UNICORN with the example.

Scenario involving highly configurable multi-stack system: We deployed a data analytics pipeline, DeepStream [2], on NVIDIA Jetson Xavier hardware as shown in Fig. 3. DeepStream has many components, and each component has many configuration options, resulting in many variants of the same system. In particular, the variability comes from: (i) the configuration options of each software component in the pipeline, (ii) configurable low-level libraries that implement functionalities required by different components (e.g., the choice of tracking algorithm in the tracker or different neural architectures), (iii) the configuration options associated with each component's deployment stack (e.g., CPU frequency of NVIDIA Xavier). Further, there exist many configurable events that can be measured/observed at the OS level by the event tracing system. Such huge variabilities make performance analysis challenging. Moreover, configuration options

[2]https://developer.nvidia.com/deepstream-sdk

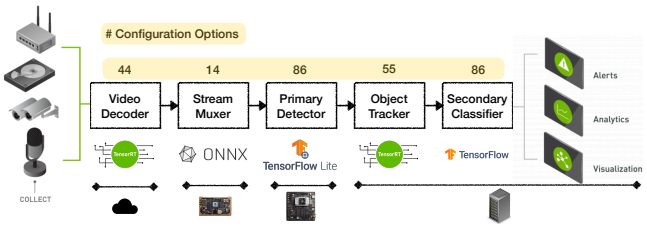


Figure 3. An example of a highly-configurable composed system, DeepStream [74]. It is a data analytics pipeline with several configurable components: a (i) Video Decoder; (ii) Stream Muxer, a plugin that accepts input streams and converts them to sequential batch frames; (iii) Primary Detector, which transforms the input frames based on input NN requirements and does model inference to detect objects; (iv) Object Tracker, which supports multi-object tracking; (v) Secondary Classifier, which improves performance by avoiding re-inferencing on the same objects in every frame.

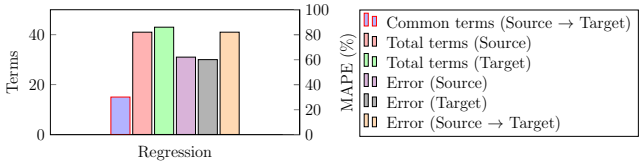


Figure 4. Regression models do not generalize well as the number of common terms, total terms and prediction error of the structural causal models change from source (XAVIER) to target (TX2). The rank correlation between source and target is 0.07 (p-value=0.73).

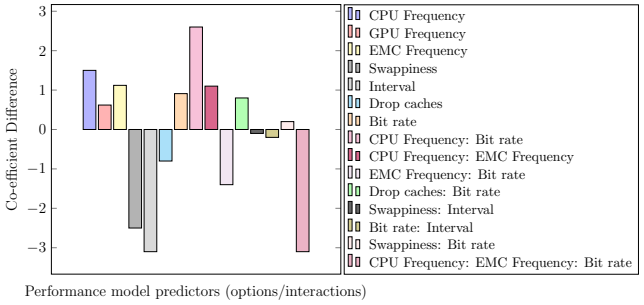
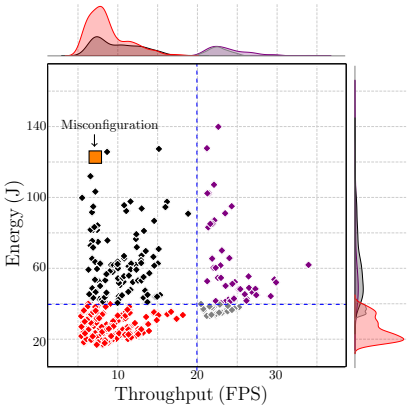


Figure 5. Visualizing co-efficient differences from the source (Xavier) regression model to the target (TX2) regression model for the common terms and interactions (shown by ":").



(a) Performance Distribution

Config. Option	Value
CPU Cores	2
CPU Frequency (GHz)	0.1
EMC Frequency (GHz)	1.6
GPU Frequency (GHz)	0.7
Scheduler Policy	NOOP
Scheduler Runtime (μs)	950k
Scheduler Child	1
Batch Timeout	1
Batch Size	1
Interval	1
Cache Size	5
DirtyBackground Ratio	5
Dirty Ratio	10
Drop Caches	0
Cache Pressure	500
Swappiness	60
GPU Memory Growth	def.
Logical Devices	50
Swap Memory (Gb)	1
Throughput (FPS)	6.8
Energy (J)	122

(b) Misconfiguration

Figure 6. (a) Multi-objective performance distribution when DeepStream is deployed on NVIDIA Jetson Xavier (b) Misconfiguration that caused the multi-objective non-functional fault (shown as the example fault in □ in the performance distribution.

among the components *interact* with each other, making the performance analysis tasks even harder.

In particular, we focus on two performance tasks: (i) *Performance Debugging*: It starts with an observed performance issue (e.g., slow execution), and the task is involved replacing the current configurations in the deployed environment with another configuration that fixes the observed performance issue; (ii) *Performance Optimization*: Here, no performance issue is observed; however, one wants to get a near-optimal performance by finding a configuration that enables the best tradeoff in the multi-objective space (e.g., throughput vs. energy consumption vs. accuracy in DeepStream). To better understand the potential of the proposed approach, we show the limitations of existing correlation-based approaches for these tasks in the context of DeepStream Framework.

In particular, we measured (i) application performance metrics including throughput and energy consumption by instrumenting the DeepStream code, and (ii) 288 system-wide performance events (hardware, software, cache, and tracepoint) using *perf* and measured the data for 2000 configurations of DeepStream. We measured the performance data in two different hardware environments, Jetson Xavier and TX2. More specifically, the configuration space of the system included (i) Software level (Decoder: 44, Stream Muxer: 14, Detector: 52), (ii) 206 Kernel level (e.g., Swappiness, Scheduler Policy, etc.), and (iii) 4 Hardware options (CPU Freq, active cores). We used 8 camera streams as the workload. We used x286 as the decoder, TrafficCamNet model that uses ResNet 18 architecture for the detector. This model is pre-trained in 4 classes on a dataset of 150k frames and has an accuracy of 83.5% for detecting and tracking cars from a traffic camera's viewpoint. The 4 classes are Vehicle, BiCycle, Person, and Roadsign. We use the Keras (Tensorflow backend) pre-trained model from TensorRT. We used NvDCF tracker, which uses a correlation filter-based online discriminative learning algorithm to a single object and uses a data association algorithm for multi-object tracking. As it is depicted in Fig. 6a, performance behavior of DeepStream, like other highly configurable systems, is non-linear, multi-modal, non-convex [51].

To show major shortcomings of existing state-of-the-art performance models, we built performance influence models that have extensively been used in the systems' literature [32, 33, 35, 53, 57, 62, 69, 90, 91] and it is the standard approach in industry [57, 62]. Specifically, we built non-linear regression models with forward and backward elimination using a step-wise training method on the DeepStream performance data. We then performed several sensitivity analyses and identified the following issues:

1. **Performance influence models could not reliably predict performance in unseen environments.** Performance behavior of configurable systems varies across environments, e.g., when we deploy a software on a new hardware with a different microarchitecture or when the workload changes [48, 53–55, 98]. When building a performance model, it is important to capture predictors (options and interactions that appear in the performance influence models of form $f(c) = \beta_0 + \Sigma_i \phi(o_i) + \Sigma_{i..j} \phi(o_i..o_j)$) that transfer well, i.e., remain *stable* across environmental changes. Such characteristic is expected from performance models since the models are learned based on one environment (e.g., staging) and are desirable to reliably predict performance in another environment (e.g., production). Therefore, if the predictors in a performance model become unstable, even if they produce accurate predictions in the current environment, there is no guarantee that it performs well in other environments, i.e., they become unreliable for performance predictions and performance optimizations due to large prediction errors. To investigate how transferable performance influence models are across environments, we performed a thorough analysis when learning a performance model for DeepStream deployed on two different hardware platforms that have two different microarchitectures. Note that such environmental changes are common, and it is known that performance behavior changes when in addition to a change of hardware resources (e.g., higher CPU frequency), we have major differences in terms of architectural constructs [20, 23], also supported by a thorough empirical study [53]. The results in Fig. 4 indicate that the number of stable predictors is too small with respect to the total number of predictors that appear in the learned regression models.
2. **Performance influence models could produce incorrect explanations.** In addition to performance predictions, where developers are interested to know the effect of configuration changes on performance objectives, they are also interested to estimate and explain the effect of a change in particular *configuration options* (e.g., changing *CachePolicy*) toward performance variations. It is therefore desirable

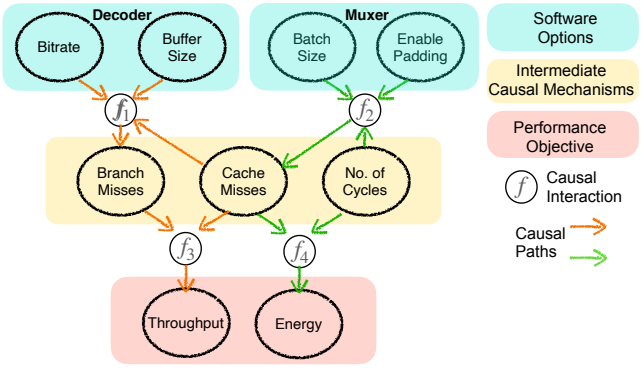


Figure 7. A Causal Performance Model for DeepStream.

that the strength of the predictors in performance models, determined by their coefficients, remain consistent across environments [23, 53]. In the context of our simple scenario in Fig. 2, the performance influence model that has been learned indicates that $0.16 \times CacheMisses$ is the most influential term that determines throughput, however, the (causal) model in Fig. 2 (c) show that there the interactions between configuration options *CachePolicy* and system event *CacheMisses* is a more reliable predictor of the throughput, indicating that the performance influence model, due to relying on superficial correlational statistics, incorrectly explains factors that influence performance behavior of the system. We performed a thorough analysis and found out that a very low Spearman rank correlation between predictors coefficients, indicating that a performance model based on regression could be highly unstable and thus would produce unreliable explanations as well as unreliable estimation of the effect of change in specific options for performance debugging purposes.

Causal Performance Models. We hypothesize that the reason behind the above-mentioned issues is the inability of correlation-based models to capture causally relevant predictors in the learned performance models. Hence, we introduce a new abstraction for performance modeling, called **Causal Performance Model (CPM)**, that gives us the leverage for performing causal reasoning in the systems performance domain. A CPM is an instantiation of Graphical Models [77] with new types and structural constraints to enable performance modeling. Formally, CPMs (cf., Fig. 7) are Directed Acyclic Graphs (DAGs) [77] with (i) performance variables, (ii) functional nodes that define functional dependencies between performance variables, and (iii) causal links that interconnect performance nodes with each other via functional nodes, and (iv) structural constraints to define assumptions we require in performance modeling. In particular, we defined three new variable types: (1) Software-level configuration options associated with a software component in the composed system (e.g., Bitrate in the decoder component of DeepStream), (2) intermediate performance variables relating the effect of configuration options to performance objectives including middleware traces (e.g., Context switches), performance events (e.g., CacheMisses), and hardware-level options (e.g., CPU frequency), and (3) end-to-end performance objectives (e.g., Throughout, Energy consumption). We also characterize the functional nodes with polynomial models to be explainable, although, in a generic form, they could be characterized with any functional forms, e.g., neural networks [84, 104]. We also defined two specific constraints over CPMs to characterize the assumptions in performance modeling: (i) defining variables that can be intervened (note that some performance variables can only be observed (e.g., CacheMisses) or in some cases where a variable can be intervened, the user may want to restrict the variability space, e.g., the cases where the user may want to use prior experience, restricting the variables that do not have a major impact to performance objectives); (ii) structural constraints, e.g., configuration options do not cause other options. Note that such constraints enable incorporating domain knowledge and enable further sparsity that facilitates learning with low sample sizes.

How causal reasoning can fix the reliability and explainability issues in current performance analyses practices. The causal performance models contain more detail than the joint distribution of all variables in the model. For example, the CPM in Fig. 7 encodes not only *BranchMisses* and *Throughpout* readings are dependent but also that lowering *CacheMisses* causes the *Throughpout* of DeepStream to increase and not the other way around. The arrows in causal performance models correspond to the assumed direction of causation, and the absence of an arrow represents the absence of direct causal influence between variables, including configuration options, system events, and performance objectives. The only way we can make predictions about how performance distribution changes for a system when deployed in another environment or when its workload changes are if we know how the variables are causally related. This information about causal relationships is not captured in non-causal models, such as regression-based models. Using the encoded information in CPMs, we can benefit from analyses that are only possible when we explicitly employ causal models, in particular, interventional and counterfactual analyses [78, 79]. For example, imagine that in a hardware platform, we deploy the DeepStream and observed that the system throughput is below 30/s and BufferSize as one of the configuration options was determined dynamically between 8k-20k. The system maintained may be interested in estimating the likelihood of fixing the performance issue in a counterfactual world where the BufferSize is set to a fixed value, 6k. The estimation of this counterfactual query is only possible if we have access to the underlying causal model because setting a specific option to a fixed value is an intervention as opposed to conditional observations that

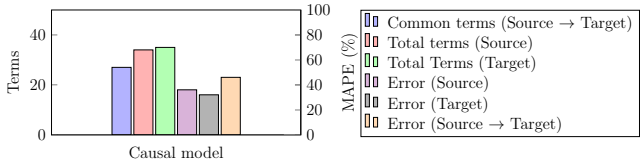


Figure 8. Causal models generalize better as the number of common terms, total terms and prediction error of the structural does not change much from source (Xavier) to target (TX2). The rank correlation between source and target is 0.49 (p-value=0.76).

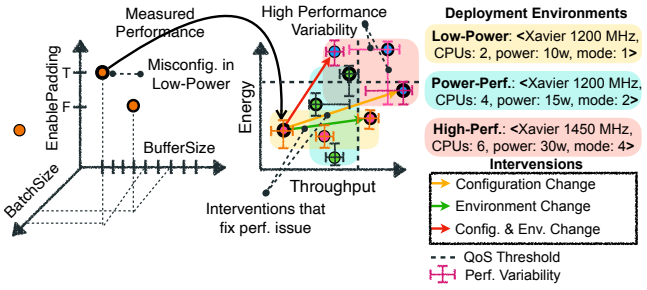


Figure 9. Mapping configuration space to multi-objective performance space

have been done in the traditional performance model for performance predictions.

CPMs are not only capable of predicting system performance in certain environments, they encode the causal structure of the underlying system performance behavior, i.e., the data-generating mechanism behind system performance. Therefore, the causal model can reliably transfer across environments [85]. To demonstrate this for CPMs as a particular characterization of causal models, we performed a similar sensitivity analysis to regression-based models and observed that CPMs can reliably predict performance in unseen environments (see Fig. 8). In addition, as opposed to PIMs that are only capable of performance predictions, CPMs can be used for several downstream heterogeneous performance tasks. For example, using a CPM, we can determine the *causal effects* of configuration options on performance objectives. Using the estimated causal effects, one can determine the effect of change in a particular set of options towards performance objectives and therefore can select the options with the highest effects to fix a performance issue, i.e., bring back the performance objective that has violated a specific quality of service constraint without sacrificing other objectives. CPMs are also capable of predicting performance behavior by calculating conditional expectation, $E(Y|X)$, where Y indicates performance objectives, e.g., throughput, and $X = x$ is the system configurations that have not been measured.

3 Unicorn

This section presents Unicorn–our methodology for performance analyses of highly configurable and composable systems with causal reasoning.

Overview. Unicorn works in five stages, implementing an active learning loop (cf. Fig. 1): (i) Users or developers of a highly-configurable system *specify*, in a human-readable language, the performance task at hand in terms of a query in the Inference Engine. For example, a DeepStream user may have experienced a throughput drop when they have deployed it on NVIDIA Xavier in low-power mode (cf. Fig. 9). Then, Unicorn's main process starts by (ii) collecting some predetermined number of samples and *learning a causal performance model*; Here, a sample contains a system configuration and its corresponding measurement—including low-level system events and end-to-end system performance. Given a certain budget, which in practice either translates to time [49] or several samples [51], Unicorn, at each iteration, (iii) *determines the next configuration(s)* and measures system performance when deployed with the determined configuration–i.e. new sample; accordingly, (iv) the *learned causal performance model is incrementally updated*, reflecting a model that captures the underlying causal structure of the system performance. Unicorn terminates if either budget is exhausted or the same configuration has been selected a certain number of times consecutively, otherwise, it continues from stage-iii. Finally, (v) to automatically derive the quantities which are needed to conduct the performance tasks, the specified performance queries are *translated* to formal causal queries, and they will be *estimated* based on the final causal model.

Stage-I: Formulate Performance Queries. Unicorn enables *developers* and *users* of highly-configurable systems to conduct performance tasks, including performance debugging, optimization, and tuning in the following scenarios. In particular, they need to answer several performance queries: (i) What configuration options *caused* the performance fault? (ii) What are *important options and their interactions* that influence performance? (iii) How to *optimize* one quality or navigate *tradeoffs* among multiple qualities in a reliable and explainable fashion? (iv) How can we *understand* what options and possible interactions are most responsible for the performance degradation in production?

At this stage, the performance queries are translated to formal causal queries using the interface of the causal inference engine in Unicorn. Note that in the current implementation of Unicorn, this translation is performed manually, however, this process could be made automatically by creating a grammar for specifying performance queries and the translations can be made between the performance query into the well-defined causal queries, note that such translation has been done in domains such as genomics [26].

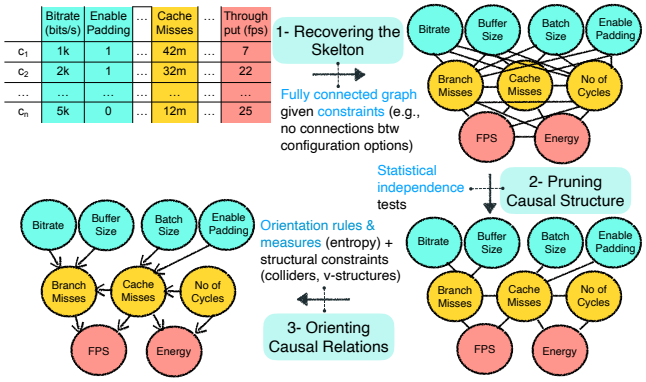


Figure 10. Causal model learning from performance data.

Stage-II: Learn Causal Performance Model In this stage, Unicorn learns a CPM (see Section 2) that explains the causal relations between configuration options, the intermediate causal mechanism, and performance objectives. Here, we use an existing structure learning algorithm called *Fast Causal Inference* (hereafter, FCI) [93]. We selected FCI because: (i) it accommodates for the existence of unobserved confounders [31, 75, 93], i.e., it operates even when there are latent common causes that have not been, or cannot be, measured. This is important because we do not assume absolute knowledge about configuration space, hence there could be certain configurations we could not modify or system events we have not observed. (ii) FCI, also, accommodates variables that belong to various data types such as nominal, ordinal, and categorical data common across the system stack (cf. Fig. 9). To build the CPM, we, first, gather a set of initial samples (cf. Table Fig. 10). To ensure reliability [20, 23], we measure each configuration multiple times, and we use their median for the causal model learning. As depicted in Fig. 10, Unicorn implements three steps for causal structure learning: (i) recovering the skeleton of the CPM by enforcing structural constraints; (ii) pruning the recovered structure using standard statistical tests of independence. In particular, we use mutual info for discrete variables and Fisher z-test for continuous variables. (iii) orienting undirected edges using entropy [17, 18, 31, 75, 93].

Stage-III: Iterative Sampling At this stage, Unicorn determines the next configuration to be measured. Unicorn first estimates the causal effects of configuration options towards performance objectives using the learned causal performance model. Then, Unicorn iteratively determines the next system configuration using the estimated causal effects as a heuristic. Specifically, Unicorn determines the value assignments for options with a probability that is determined proportionally based on their associated causal effects. The key intuition is that such changes in the options are more likely to have a larger effect on performance objectives, and therefore we can learn more about the performance behavior

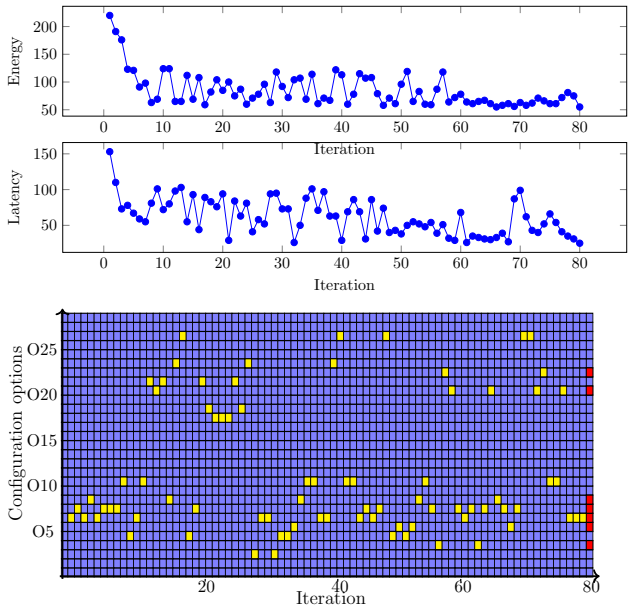


Figure 11. Incremental update of Latency and Energy using Unicorn for debugging a multi-objective fault (top two plots). Yellow-colored nodes indicate the configuration option values recommended by Unicorn (bottom plot). Red colored nodes indicate the recommended fixes from the fault (Iteration 1.)

of the system. Given the exponentially large configuration space and the fact that the span of performance variations is determined by a small percentage of configurations, if we had ignored such estimates for determining the change in configuration options, the next configurations would result in considerable variations in performance objectives comparing with the existing data. Therefore, measuring the next configuration would not provide additional information for the causal model.

Stage-IV: Update Causal Performance Model At each iteration, Unicorn measures the configuration that is determined by Phase-III and updates the causal performance model (CPM) incrementally. Since the causal model uses limited observational data, there may be a discrepancy between the underlying performance model and the learned CPM, note that this issue exist in all domains using data-driven models, including causal reasoning [78]. The more accurate the causal graph, the more accurate the proposed intervention will be [17, 18, 31, 75, 93]. Therefore, in case our repairs do not fix the faults, we update the observational data with this new configuration and repeat the process. Over time, the estimations of causal effects will become more accurate. We terminate the incremental learning once we achieve the desired performance.

The reason behind this initial sampling is to learn an initial causal model and given that our sampling budget is limited, we maximize the information in the causal model iteratively

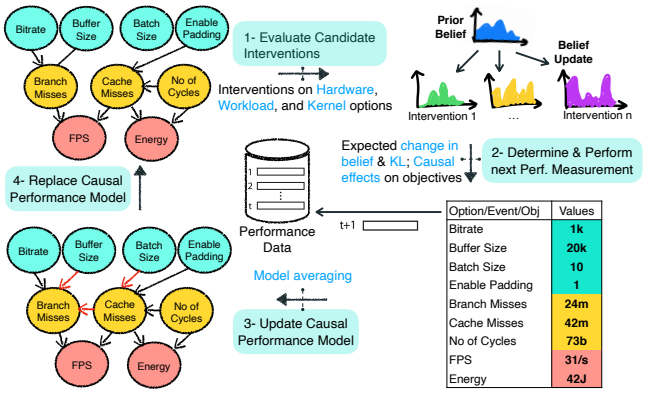


Figure 12. Causal model update.

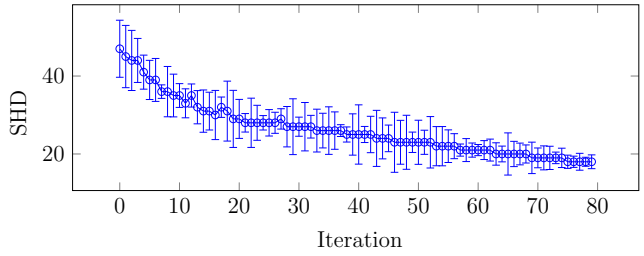


Figure 13. The structural distance (hamming distance) between the learned causal model and ground truth model iteratively decreases.

following a Bayesian approach where at iteration t the current causal model is considered as prior and given the new sample it gets updated.

Stage-V: Estimate Performance Queries At this stage, given the learned causal performance model, UNICORN's inference engine estimates the user-specified queries using the mathematics of causal reasoning–do-calculus. Specifically, the causal inference engine provides a quantitative estimate for the identifiable queries on the current causal model and may return some queries as unidentifiable. It also determines what assumptions or new measurements are required to answer the "unanswerable" questions, so, the user can decide to incorporate these new assumptions by defining more constraints or increase the sampling budgets.

Implementation. There are generic causal modeling and inference tools developed and maintained by industry (e.g., Microsoft's DoWhy [87], IBM's CausalLib [88], Uber's CausalML [97]) and academia (e.g., Ananke [12], Causal Fusion [11]); these tools only implement the core machinery in causal reasoning including different causal modeling, structure learning, inference, counterfactuals, and estimation tasks. We implemented UNICORN by integrating and building on top of (i) *Semopy* [86] for predictions with causal models, (ii) *Ananke* [12] for estimating the causal effects, (iii) *CausalML* [97] for counterfactual reasoning, *PyCausal* [81] for structure learning. In particular, we integrated the tools into a seamless pipeline to provide users with the following capabilities for performance modeling and analyses: (i) **Modeling**: specifying the causal variables and constraints needed for learning a correct causal performance model (see Section 2); (ii) **Analyses**: Iterative sampling, model learning and update, estimation of performance queries, and early stopping; and (iii) **User interface**: visualization, annotations, interactions with PCM models, and query estimations.

Problem [3]: For a real-time scene detection task, TX2 (faster platform) only processed 4 frames/sec whereas TX1 (slower platform) processed 17 frames/sec, i.e., the latency is 4× worse on TX2.
Observed Latency (frames/sec): 4 FPS
Expected Latency (frames/sec): 22-24 FPS *(30-40% better)*

Configuration Options	UNICORN	SMAC	BUGDOC	Forum	ACE†
CPU Cores	✓	✓	✓	✓	3%
CPU Frequency	✓	✓	✓	✓	6%
EMC Frequency	✓	✓	✓	✓	13%
GPU Frequency	✓	✓	✓	✓	22%
Scheduler Policy	·	✓	✓	·	.
Sched rt runtime	·	·	✓	·	.
Sched child runs	·	·	✓	·	.
Dirty bg. Ratio	·	·	·	·	.
Dirty Ratio	·	·	✓	·	.
Drop Caches	·	✓	✓	·	.
CUDA_STATIC	✓	✓	✓	✓	55%
Cache Pressure	·	·	·	·	.
Swappiness	·	✓	✓	·	1%
Latency (TX2 frames/sec)	**28**	24	20	23	
Latency Gain (over TX1)	**65%**	42%	21%	39%	
Latency Gain (over default)	**7×**	6×	5×	5.75×	
Resolution time	**22 mins**	4 hrs	3.5 hrs	2 days	

Figure 14. Using UNICORN on a real-world performance issue.

4 Case Study

Prior to a systematic evaluation in Section 5, here, we show how UNICORN can enable performance debugging in a real-world scenario discussed in [3], where a developer migrated a real-time scene detection system from NVIDIA TX1 to a more powerful hardware, TX2. However, the developer, surprisingly, experienced 4× worse latency in the new environment (from 17 frames/sec in TX1 to 3 frames/sec in TX2). After two days of discussions, the performance issue was diagnosed with a misconfiguration–an incorrect setting of a compiler option and two hardware options. Here, we assess whether and how UNICORN could facilitate the performance debugging by comparing with (i) the fix suggested by NVIDIA in the forum, and two academic performance debugging approaches– BUGDOC [65] and SMAC [47].
Findings. Fig. 14 illustrates our findings. We find that:

• UNICORN could diagnose the root cause of the misconfiguration and recommends a fix within 24 minutes. Using the recommended configuration fixes from UNICORN, we achieved a throughput of 28 frames/sec (65% higher than TX1 and 7× higher than the fault). This, surprisingly, exceeds the developers' initial expectation of 30 – 40% improvement.
• BUGDOC (a diagnosis approach) has the least improvement compared to other approaches (21% improvement over TX1), while taking 3.5 hours to suggest the fix. BUGDOC also changed several unrelated options (depicted by ✔) not endorsed by the domain experts.
• Using SMAC (an optimization approach), we aimed to find a configuration that achieves optimal throughput. However, after converging, SMAC recommended a configuration which achieved 24 frames/sec (42% better than TX1 and 6× better than the fault), however, could not outperform the configuration suggested by UNICORN and even took 4 hours (6× longer than UNICORN to converge). In addition, SMAC changed several unrelated options (✔ in Fig. 14).

Why UNICORN works better (and faster)? UNICORN discovers the misconfigurations by constructing a causal model (a simplified version of this is shown in Fig. 14). This causal model rules out irrelevant configuration options and focuses on the configurations that have the highest (direct or indirect) causal effect on latency, e.g., we found the root-cause `CUDA STATIC` in the causal graph which indirectly affects latency via context-switches (an intermediate system event); this is similar to other relevant configurations that indirectly affected latency (via energy consumption). Using counterfactual queries, UNICORN can reason about changes to configurations with the highest average causal effect (ACE) (last column in Fig. 14). The counterfactual reasoning occurs no additional measurements, significantly speeding up inference as shown in Fig. 14, UNICORN accurately finds all the configuration options recommended by the forum (depicted by ✔ in Fig. 14).

5 Evaluations

We examine the following by comparing UNICORN with state-of-the-art performance debugging and optimization approaches: (i) **Effectiveness** (Section 6) in terms of sample efficiency and performance gain. (ii) **Transferability** (Section 7) of learned models across environments, and (iii) **Scalability** (Section 8) to large-scale highly-configurable systems.

Systems. We selected six configurable systems including a video analytic pipeline, three deep learning-based systems (for image, speech, and NLP), a video encoder, and a database, see Table 1. We use heterogeneous deployment platforms, including NVIDIA TX1, TX2, and XAVIER, each having different resources (compute, memory) and microarchitectures.

Configuration. We choose a wide range of configuration options and system events (see Table 1), following NVIDIA's configuration guides/tutorials and other related work [36]. As opposed to prior work that only can support binary options due to scalability issues (e.g., [99, 100], we included options with binary, discrete, and continuous.

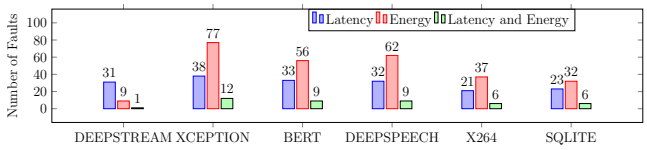


Figure 15. Distribution of 451 single-objective and 43 multi-objective non-functional faults across different software systems used in our study.

Table 1. Overview of the subject systems used in our study.

System	Workload	$\|C\|$	$\|O\|$	$\|S\|$	$\|\mathcal{H}\|$	$\|\mathcal{W}\|$	$\|\mathcal{P}\|$
DEEPSTREAM	Video analytics pipeline for detection and tracking from 8 camera streams	2461	53	288	2	1	2
XCEPTION [16]	Image recognition system to classify 5000/5000 test images from CIFAR10	6443	28	19	3	3	3
DEEPSPEECH [40]	Speech-to-text from 0.5/1932 hours of Common Voice Corpus5.1 (English) data	6112	28	19	3	1	3
BERT [22]	NLP system for sentiment analysis of 1000/25000 test reviews from IMDb	6188	28	19	3	1	3
x264	Encodes a 20 second 11.2 MB video of resolution 1920 x 1080 from UGC	17248	32	19	3	1	3
SQLITE	DBMS for sequential reads and writes, random reads, batch writes and deletions	15680	242	288	3	3	3

* C: Configurations, O: Options, S: System Events, $\mathcal{H}$: Hardware, $\mathcal{W}$: Workload, $\mathcal{P}$: Objectives

Ground truth. To ensure reliable and replicable results, following the common practice [20, 23, 53, 57], we measured 2000 samples for each 15 deployment settings (5 systems and 3 hardware) and repeated each measurement 5 times and recorded the median. We curated a ground truth of performance issues, called JETSON FAULTS, for each of the studied software and hardware systems using the measured ground truth data. By definition, non-functional faults are located in the tail of performance distributions [34, 59]. We, therefore, selected and labeled configurations that are worse than the 99th percentile as '*faulty*.' Fig. 15 shows the total 494 faults discovered across different software. Out of these 494 NF-FAULTS, 43 are faults with multiple types (both energy and latency). Of all the 451 single-objective and 43 multi-objective NF-FAULTS discovered in this study, only 2 faults had a single root cause, 411 faults had five or more root causes, and 81 remaining faults had two to four root causes.
Initial samples: 25–10% of the total sampling budget.

Baselines. We evaluate UNICORN for two performance tasks, including **Task 1. Performance Debugging and Repair** and **Task 2. Performance Optimization**. We compare UNICORN against state-of-the-art, including **CBI** [92], a feature

selection algorithm for fixing performance issues; **DD** [7], a delta debugging technique, that minimizes the difference between a pair of configurations; **EnCore** [106] learns to debug from correlational information about misconfigurations; **BugDoc** [65] infers the root causes and derive succinct explanations of failures using decision trees; **SMAC** [47], A sequential model-based auto-tuning approach; and **PESMO** [42], A multi-objective Bayesian optimization approach.

Evaluation Metrics. (i) **Recall**, the percentage of true root-causes that are correctly predicted, (ii) **Precision**, the percentage of true root-causes among the predicted ones, (iii) Accuracy, weighted Jaccard similarity between the predicted and true root-causes, where the weight vector was derived based on the causal effects of options to performance based on the ground-truth CPM. For example, if A is the recommended configuration by an approach and B is the configuration that fixes the performance issue in the ground truth data, we measure $accuracy = \frac{\sum_{ACE}(A \cap B)}{\sum_{ACE}(A \cup B)}$. (iv) **Gain**, percentage improvement of suggested fix over the observed fault– $\Delta_{gain} = \frac{NFP_{FAULT} - NFP_{NOFAULT}}{NFP_{FAULT}} \times 100$, where NFP_{FAULT} the observed faulty performance and $NFP_{NO\ FAULT}$ is the performance of suggested fix. (v) **Error**: Single objective: the absolute difference between the best suggested and the optimal in ground truth; Multi-objective: hypervolume error [109]. (vi) **Time**, overhead of an approach in terms of wallclock time in hours to converge and suggest a fix.

6 Effectiveness and Sample Efficiency

Setting: We only show the partial results, however, our results generalize to all evaluated settings. *Debugging*: latency faults in TX2 and energy faults in Xavier. *Optimization (single-objective)*: comparison with SMAC in TX2 for Xception for both latency and energy. *Optimization (multi-objective)*: comparison with PESMO in TX2. We repeat the entire debugging and optimization tasks 3 times.

Results (debugging). Tables 13a and 14 shows Unicorn *significantly outperforms correlation-based methods in all cases.* For example, in Deepstream on TX2, Unicorn achieves 6% more accuracy, 12% more precision, and 10% more recall compared to the next best method, BugDoc. We observed latency gains as high as 88% (9% more than BugDoc) on TX2 and energy gain of 86% (9% more than BugDoc) on Xavier for Xception. We observe similar trends in energy faults and multi-objective faults.The results confirm that Unicorn *can recommend repairs for faults that significantly improve latency and energy usage.* Applying the changes to the configurations recommended by Unicorn increases the performance drastically.

Figure 17a and Figure 17b demonstrate the sample efficiency results for different systems. We observe that for both latency and energy faults Unicorn achieved significantly higher gains with significantly less number of samples. For Xception, Unicorn required an 8x lower number of samples to obtain 32% higher gain than DD. The higher gain in Unicorn with a relatively lower number of samples in comparison to correlation-based methods indicates that Unicorn causal reasoning is more effective in guiding the search in the objective space. Unicorn does not waste budget with evaluating configurations with lower causal effects and finds a fix faster.

Unicorn *can resolve misconfiguration faults significantly faster than correlation-based approaches.* In Tables 13a and 14, the last two columns indicate the time taken (in hours) by each approach to diagnosing the root cause. For all methods, we set a maximum budget of 4 hours. We find that, while other approaches use the entire budget to diagnose and resolve the faults, Unicorn can do so significantly faster, e.g., Unicorn is 13× faster in diagnosing and resolving faults in energy usage for x264 deployed on Xavier and 10× faster for latency faults for Bert on TX2.

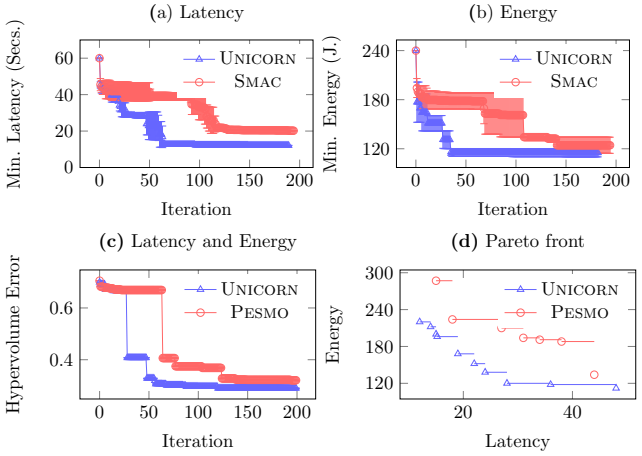


Figure 16. Unicorn vs. optimization with SMAC and PESMO.

Results (optimization). Fig. 16(a) and 16(b) demonstrate the single-objective optimization results—Unicorn finds configurations with optimal latency and energy for both cases. Fig. 16(a) illustrates that the optimal configuration discovered by Unicorn has 43% lower latency (12 seconds) than that of SMAC (21 seconds). Here, Unicorn reaches near-optimal configuration by only exhausting one-third of the entire budget. In Fig. 16(b), the optimal configuration discovered by Unicorn and SMAC had almost the same energy, but Unicorn reached this optimal configuration 4x faster than SMAC. In both single-objective optimizations, the iterative variation of Unicorn is less than SMAC–i.e., Unicorn finds more stable configurations. Figure 16(c) compares Unicorn with PESMO to optimize both latency and energy in TX2 (for image recognition). Here, Unicorn has 12% lower hypervolume error than PESMO and reaches the same level of hypervolume error of PESMO 4x times faster. Fig.16(d) illustrates the optimal Pareto front obtained by Unicorn

Table 2. Efficiency of UNICORN compared to other approaches. Cells highlighted in **blue** indicate improvement over faults.

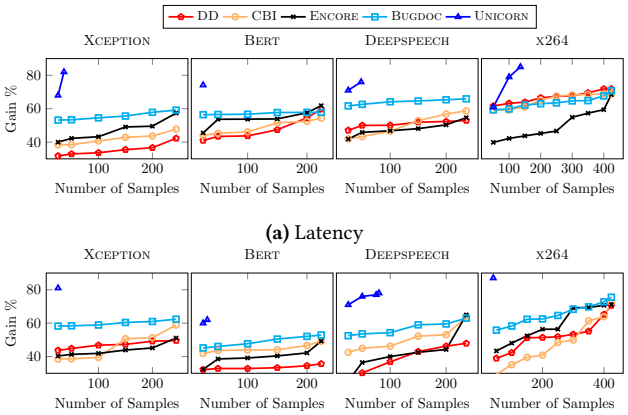
(a) Single objective performance fault in latency and energy consumption.

			Accuracy					Precision					Recall					Gain					Time†	
			UNICORN	CBI	DD	EnCore	BugDoc	UNICORN	CBI	DD	EnCore	BugDoc	UNICORN	CBI	DD	EnCore	BugDoc	UNICORN	CBI	DD	EnCore	BugDoc	UNICORN	Others
TX2	Latency	Deepstream	**87**	61	62	65	81	**83**	66	59	60	71	**80**	61	65	60	70	**88**	66	67	68	79	**0.8**	4
		Xception	**86**	53	42	62	65	**86**	67	61	63	67	**83**	64	68	69	62	**82**	48	42	57	59	**0.6**	4
		BERT	**81**	56	59	60	57	**76**	57	55	61	73	**71**	74	68	67	65	**74**	54	59	62	58	**0.4**	4
		Deepspeech	**81**	61	59	60	72	**76**	58	69	61	71	**81**	73	61	63	69	**76**	59	53	55	66	**0.7**	4
		x264	**83**	59	63	62	62	**82**	69	58	65	66	**78**	64	67	63	72	**85**	69	72	68	71	**1.4**	4
Xavier	Energy	Deepstream	**91**	81	79	77	87	**81**	61	62	64	73	**85**	63	61	62	75	**86**	68	62	61	78	**0.7**	4
		Xception	**84**	66	63	63	81	**78**	56	58	66	65	**80**	69	55	63	68	**83**	59	50	51	62	**0.4**	4
		BERT	66	59	53	63	**72**	**70**	62	64	64	65	**79**	61	54	63	66	**62**	49	36	49	53	**0.5**	4
		Deepspeech	**73**	68	63	72	71	**75**	55	59	54	68	**78**	53	52	59	71	**78**	64	48	65	63	**1.2**	4
		x264	**77**	71	70	74	74	**83**	63	53	61	66	**78**	67	53	54	72	**87**	73	71	76	76	**0.3**	4

(b) Multi-objective non-functional faults in *Energy, Latency.*

		Accuracy				Precision				Recall				Gain (Latency)				Gain (Energy)				Time†	
		UNICORN	CBI	EnCore	BugDoc	UNICORN	CBI	EnCore	BugDoc	UNICORN	CBI	EnCore	BugDoc	UNICORN	CBI	EnCore	BugDoc	UNICORN	CBI	EnCore	BugDoc	UNICORN	Others
Energy + Latency	Xception	**89**	76	81	79	**77**	53	54	62	**81**	59	59	62	**84**	53	61	65	**75**	38	46	44	**0.9**	4
	BERT	71	72	**73**	71	**77**	42	56	63	**79**	59	62	65	**84**	53	59	61	**67**	41	27	48	**0.5**	4
	Deepspeech	**86**	69	71	72	**80**	44	53	62	**81**	51	59	64	**88**	55	55	62	**77**	43	43	41	**1.1**	4
	x264	**85**	73	83	81	**83**	50	54	67	**80**	63	62	61	**75**	62	64	66	**76**	64	66	64	**1**	4

† Wallclock time in hours



(a) Latency

(b) Energy

Figure 17. UNICORN has significantly higher sampling efficiency than other baselines in debugging non-functional faults.

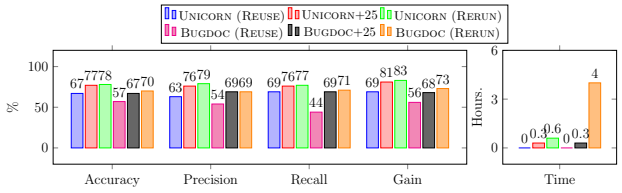


Figure 18. Accuracy, Precision, Recall, and Gain of debugging non-functional faults (XAVIER to TX2).

and PESMO. The Pareto front by UNICORN has higher coverage as it discovered a higher number of Pareto optimal configurations with lower energy and latency value than PESMO.

7 Transferability

Setting. We reuse the CPM constructed from a source environment, e.g., TX1, to resolve a non-functional fault in a target environment, e.g., XAVIER. We evaluated UNICORN for debugging energy faults for XCEPTION and used XAVIER as the source and TX2 as the target, since they have different microarchitectures, expecting to see large differences in their performance behaviors. We only compared with BUGDOC as it discovered fixes with higher energy gain in XAVIER than other correlation-based baseline methods (see Table 13a). We compared UNICORN and BUGDOC in the following scenarios: (I) REUSE: reusing the recommended configurations from Source to Target, (II) +25: reusing the performance models (i.e., causal model and decision tree) learned in Source and fined-tuned the models with 25 new samples in Target, and (III) RERUN: we rerun UNICORN and BUGDOC from scratch to resolve energy faults in Target. For optimization tasks, we use three larger additional XCEPTION workloads: 10000 (10k), 20000 (20k), and 50000 (50k) test images (previous experiments used 5000 (5k) test images). We evaluated three variants of SMAC and UNICORN: (1) SMAC and UNICORN (REUSE), where we *reuse* the near-optimum found with a 5K

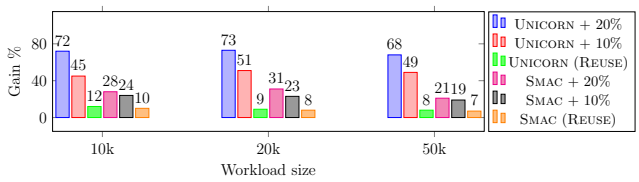


Figure 19. UNICORN finds configurations with higher gain when workloads are changed for performance optimization task.

tests image on the larger workloads; (2) SMAC +10% and UNICORN +10%, where we rerun with 10% budget in target and update the CPM with 10% additional budget; and (3) SMAC +20% and UNICORN +20%, where we rerun with 20% budget in target and update the model with 20% additional budget.

Results. Fig. 18 indicates the results in resolving energy faults in TX2. We observe that UNICORN +25 obtains 8% more accuracy, 7% more precision, 5% more recall and 8% more gain than BUGDOC (RERUN). Here, BUGDOC takes significantly longer than UNICORN, i.e., BUGDOC (RERUN) exceeds the 4-hour budget in whereas UNICORN takes at most 20 minutes to fix the energy faults. We have to rerun BugDoc every time the hardware changes, and this limits its practical usability. In contrast, UNICORN incrementally updates the internal causal model with new samples from the newer hardware to learn new relationships. Therefore, it is less sensitive and much faster. Since UNICORN uses causal models to infer candidate fixes using only the available observational data, it tends to be much faster than BUGDOC. We also observe that with little updates, UNICORN +25 (20 minutes) achieves a similar performance of UNICORN (RERUN) (36 minutes). Since the causal mechanisms are sparse, the CPM from XAVIER in UNICORN quickly reaches a fixed structure in TX2 using incremental learning by judiciously evaluating the most promising fixes until the fault is resolved.

Our experimental results demonstrate that UNICORN performs better than the two variants of three SMAC (c.f. Figure 19). SMAC (REUSE) performs the worst when the workload changes. With 10K images, reusing the near-optimal configuration from 5K images results in a latency gain of 10%, compared to 12% with UNICORN in comparison with the default configuration. We observe that UNICORN + 20% achieves 44%, 42%, and 47% higher gain than SMAC + 20% for workload sizes of 10k, 20k, and 50k images, respectively.

8 Scalability

Setting. We evaluated UNICORN for scalability with SQLITE (large configuration space) and DEEPSTREAM (large composed system). In SQLITE, we conducted the evaluation in three scenarios: (a) selecting the most relevant software/hardware options and events (34 configuration options and 19 system events), (b) selecting all modifiable software and hardware options and system events (242 configuration options and 19 events), and (c) selecting not only all modifiable software and hardware options and system events but also intermediate `tracepoint events` (242 configuration options and 288 events). In DEEPSTREAM, there are two scenarios: (a) 53 configuration options and 19 system events, and (b) 53 configuration options and 288 events when we select all modifiable software and hardware options, and system/tracepoint events.

Table 3. Scalability for SQLITE and DEEPSTREAM on XAVIER.

System	Configs	Events	Paths	Queries	Degree	Gain (%)	Time/Fault (in sec.) Discovery	Query Eval	**Total**
SQLITE	34	19	32	191	3.6	93	9	14	**291**
	242	19	111	2234	1.9	94	57	129	**1345**
	242	288	441	22372	1.6	92	111	854	**5312**
DEEPSTREAM	53	19	43	497	3.1	86	16	32	**1509**
	53	288	219	5008	2.3	85	97	168	**3113**

Results. In large systems, there are significantly more causal paths and therefore, causal learning and estimations of queries take more time. The results in Table 3 indicate that UNICORN can scale to a much larger configuration space without an exponential increase in runtime for any of the intermediate stages. This can be attributed to the sparsity of the causal graph (average degree of a node for SQLITE in Table 3 is at most 3.6, and it reduces to 1.6 when the number of configurations increase and reduces from 3.1 to 2.3 in DEEPSTREAM when systems events are increased).

9 Related Work

Performance Faults in Configurable Systems. Previous empirical studies have shown that a majority of performance issues are due to misconfigurations [38], with severe consequences in production environments [66, 95], and configuration options that cause such performance faults force the users to tune the systems themselves [108]. Previous works have used static and dynamic program analysis to identify the influence of configuration options on performance [64, 99, 100] and to detect and diagnose misconfigurations [8, 9, 105, 107]. Unlike UNICORN, none of the white-box analysis approaches target configuration space across system stack, where it limits their applicabilities in identifying the true causes of a performance fault.

Statistical and Model-based Debugging. Debugging approaches such as STATISTICAL DEBUGGING [92], HOLMES [15], XTREE [63], BUGDOC [65], ENCORE [65], REX [67], and PERFLEARNER [39] have been proposed to detect root causes of system faults. These methods make use of statistical diagnosis and pattern mining to rank the probable causes based on their likelihood of being the root causes of faults. However,

these approaches may produce correlated predicates that lead to incorrect explanations.

Causal Testing and Profiling. Causal inference has been used for fault localization [10, 28], resource allocations in cloud systems [30], and causal effect estimation for advertisement recommendation systems [13]. More recently, AID [27] detects root causes of a intermittent software failure using fault injection as interventions. Causal Testing and Holmes [56] modifies the system inputs to observe behavioral changes and utilizes counterfactual reasoning to find the root causes of bugs. Causal profiling approaches like CoZ [21] points developers where optimizations will improve performance and quantifies their potential impact. Causal inference methods like X-Ray [8] and ConfAid [9] had previously been applied to analyze program failures. All approaches above are either orthogonal or complimentary to Unicorn, mostly they focus on functional bugs (e.g., Causal Testing) or if they are performance related, they are not configuration-aware (e.g., CoZ).

10 Conclusion

Modern computer systems are highly-configurable with thousands of interacting configurations with a complex performance behavior. Misconfigurations in these systems can elicit complex interactions between software and hardware configuration options, resulting in non-functional faults. We propose Unicorn, a novel approach for diagnostics that learns and exploits the causal structure of configuration options, system events, and performance metrics. Our evaluation shows that Unicorn effectively and quickly diagnoses the root cause of non-functional faults and recommends high-quality repairs to mitigate these faults.

Acknowledgements

This work has been supported in part by NASA (RASPBERRY-SI 80NSSC20K1720) and NSF (Awards 2007202 and 2107463). We are grateful to all who provided feedback on this work, including Christian Kästner, Sven Apel, Yuriy Brun, Emery Berger, Tianyin Xu, Vivek Nair, Jianhai Su, Miguel Velez, Tobius Dürschmid, the anonymous reviewers, and the audiences of early presentations of this work.

References

[1] Xbox: Microsoft azure and xbox live servicese xperiencing outages. https://gadgets.ndtv.com/internet/news/microsoft-azure-and-xbox-live-services-experiencing-outages-622865, November 2014.

[2] Slow image classification with tensorflow on TX2. In NVIDIA developer forums: https://forums.developer.nvidia.com/t/54307, October 2017.

[3] Cuda performance issue on TX2. In NVIDIA developer forums: https://forums.developer.nvidia.com/t/50477, June 2020.

[4] General performance problems. In NVIDIA developer forums: https://forums.developer.nvidia.com/t/111704, February 2020.

[5] High CPU usage on jetson TX2 with GigE fully loaded. In NVIDIA developer forums: https://forums.developer.nvidia.com/t/124381, May 2020.

[6] Alcocer, J. P. S., Bergel, A., Ducasse, S., and Denker, M. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *Proc. of Working Conference on Software Visualization (VISSOFT)* (2013), IEEE, pp. 1–9.

[7] Artho, C. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer 13*, 3 (2011), 223–246.

[8] Attariyan, M., Chow, M., and Flinn, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)* (2012), pp. 307–320.

[9] Attariyan, M., and Flinn, J. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI* (2010), vol. 10, pp. 1–14.

[10] Baah, G. K., Podgurski, A., and Harrold, M. J. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis* (2010), pp. 73–84.

[11] Bareinboim, E. Causal fusion. https://www.causalfusion.net/, 2021.

[12] Bhattacharya, R., Nabi, R., and Shpitser, I. Semiparametric inference for causal effects in graphical models with hidden variables. *arXiv preprint arXiv:2003.12659* (2020).

[13] Bottou, L., Peters, J., Quiñonero-Candela, J., Charles, D. X., Chickering, D. M., Portugaly, E., Ray, D., Simard, P., and Snelson, E. Counterfactual reasoning and learning systems: The example of computational advertising. *The Journal of Machine Learning Research 14*, 1 (2013), 3207–3260.

[14] Bryant, R. E., David Richard, O., and David Richard, O. *Computer systems: a programmer's perspective*, vol. 2. 2003.

[15] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K. Holmes: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering* (2009), IEEE, pp. 34–44.

[16] Chollet, F. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 1251–1258.

[17] Colombo, D., and Maathuis, M. H. Order-independent constraint-based causal structure learning. *The Journal of Machine Learning Research 15*, 1 (2014), 3741–3782.

[18] Colombo, D., Maathuis, M. H., Kalisch, M., and Richardson, T. S. Learning high-dimensional directed acyclic graphs with latent and selection variables. *The Annals of Statistics* (2012), 294–321.

[19] Connelly, L. M. Fisher's exact test. *Medsurg Nursing 25*, 1 (2016).

[20] Curtsinger, C., and Berger, E. D. Stabilizer: Statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News 41*, 1 (2013), 219–228.

[21] Curtsinger, C., and Berger, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 184–197.

[22] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[23] Ding, Y., Pervaiz, A., Carbin, M., and Hoffmann, H. Generalizable and interpretable learning for configuration extrapolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), pp. 728–740.

[24] Elkhodary, A., Esfahani, N., and Malek, S. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)* (2010), ACM, pp. 7–16.

[25] Esfahani, N., Elkhodary, A., and Malek, S. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE Trans. Softw. Eng. (TSE) 39*, 11 (2013), 1467–1493.

[26] Farahmand, S., O'Connor, C., Macoska, J. A., and Zarringhalam, K. Causal inference engine: a platform for directional gene set enrichment analysis and inference of active transcriptional regulators. *Nucleic acids research 47*, 22 (2019), 11563–11573.

[27] Fariha, A., Nath, S., and Meliou, A. Causality-guided adaptive interventional debugging. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 431–446.

[28] Feyzi, F., and Parsa, S. Inference: effective fault localization based on information-theoretic analysis and statistical causal inference. *Frontiers of Computer Science 13*, 4 (2019), 735–759.

[29] Filieri, A., Hoffmann, H., and Maggio, M. Automated multi-objective control for self-adaptive software design. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)* (2015), ACM, pp. 13–24.

[30] Geiger, P., Carata, L., and Schölkopf, B. Causal models for debugging and control in cloud computing. *arXiv preprint arXiv 1603* (2016).

[31] Glymour, C., Zhang, K., and Spirtes, P. Review of causal discovery methods based on graphical models. *Frontiers in genetics 10* (2019), 524.

[32] Grebhahn, A., Siegmund, N., and Apel, S. Predicting performance of software configurations: There is no silver bullet. *arXiv preprint arXiv:1911.12643* (2019).

[33] Grebhahn, A., Siegmund, N., Köstler, H., and Apel, S. Performance prediction of multigrid-solver configurations. In *Software for Exascale Computing-SPPEXA 2013-2015*. Springer, 2016, pp. 69–88.

[34] Gunawi, H. S., et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS) 14*, 3 (2018), 1–26.

[35] Guo, J., Czarnecki, K., Apel, S., Siegmund, N., and Wasowski, A. Variability-aware performance prediction: A statistical learning approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (2013), IEEE.

[36] Halawa, H., Abdelhafez, H. A., Boktor, A., and Ripeanu, M. NVIDIA jetson platform characterization. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics) 10417 LNCS* (2017), 92–105.

[37] Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., and Baudry, B. Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. *Empirical Software Engineering 24*, 2 (2019), 674–717.

[38] Han, X., and Yu, T. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2016).

[39] Han, X., Yu, T., and Lo, D. Perflearner: learning from bug reports to understand and generate performance test frames. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2018).

[40] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

[41] Henard, C., Papadakis, M., Harman, M., and Le Traon, Y. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)* (2015), IEEE, pp. 517–528.

[42] Hernández-Lobato, D., Hernandez-Lobato, J., Shah, A., and Adams, R. Predictive entropy search for multi-objective bayesian optimization. In *International Conference on Machine Learning* (2016), pp. 1492–1501.

[43] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. In *In Proc. of Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[44] Hoos, H. H. Automated algorithm configuration and parameter tuning. In *Autonomous search*. Springer, 2011, pp. 37–71.

[45] Hoos, H. H. Programming by optimization. *Communications of the ACM 55*, 2 (2012), 70–80.

[46] Hu, Y., Huang, G., and Huang, P. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)* (2020), pp. 719–734.

[47] Hutter, F., Hoos, H. H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization* (2011), Springer, pp. 507–523.

[48] Iqbal, M. S., Kotthoff, L., and Jamshidi, P. Transfer Learning for Performance Modeling of Deep Neural Network Systems. In *USENIX Conference on Operational Machine Learning* (Santa Clara, CA, 2019), USENIX Association.

[49] Iqbal, M. S., Su, J., Kotthoff, L., and Jamshidi, P. Flexibo: Cost-aware multi-objective optimization of deep neural networks. *arXiv preprint arXiv:2001.06588* (2020).

[50] Jamshidi, P., and Casale, G. An uncertainty-aware approach to optimal configuration of stream processing systems. In *Proc. Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2016), IEEE.

[51] Jamshidi, P., and Casale, G. An uncertainty-aware approach to optimal configuration of stream processing systems. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2016), IEEE, pp. 39–48.

[52] Jamshidi, P., Ghafari, M., Ahmad, A., and Pahl, C. A framework for classifying and comparing architecture-centric software evolution research. In *Proc. of European Conference on Software Maintenance and Reengineering (CSMR)* (2013), IEEE, pp. 305–314.

[53] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., and Agarwal, Y. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (2017), ACM.

[54] Jamshidi, P., Velez, M., Kästner, C., and Siegmund, N. Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)* (2018), ACM.

[55] Jamshidi, P., Velez, M., Kästner, C., Siegmund, N., and Kawthekar, P. Transfer learning for improving model predictions in highly configurable software. In *Proc. Int'l Symp. Soft. Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2017), IEEE.

[56] Johnson, B., Brun, Y., and Meliou, A. Causal testing: Understanding defects' root causes. In *Proceedings of the 2020 International Conference on Software Engineering* (2020).

[57] Kaltenecker, C., Grebhahn, A., Siegmund, N., and Apel, S. The interplay of sampling and machine learning for software performance prediction. *IEEE Software* (2020).

[58] Kawthekar, P., and Kästner, C. Sensitivity analysis for building evolving and & adaptive robotic software. In *Proceedings of the IJCAI Workshop on Autonomous Mobile Service Robots (WSR)* (7 2016).

[59] Kleppmann, M. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017.

[60] Kocaoglu, M., Dimakis, A. G., Vishwanath, S., and Hassibi, B. Entropic causal inference. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (2017), p. 1156–1162.

[61] Kocaoglu, M., Shakkottai, S., Dimakis, A., Caramanis, C., and Vishwanath, S. Applications of Common Entropy for Causal Inference.

[62] Kolesnikov, S., Siegmund, N., Kästner, C., Grebhahn, A., and Apel, S. Tradeoffs in modeling performance of highly configurable

software systems. *Software & Systems Modeling 18*, 3 (2019), 2265–2283.

[63] Krishna, R., Menzies, T., and Layman, L. Less is more: Minimizing code reorganization using xtree. *Information and Software Technology 88* (2017), 53–66.

[64] Li, C., Wang, S., Hoffmann, H., and Lu, S. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16.

[65] Lourenço, R., Freire, J., and Shasha, D. Bugdoc: A system for debugging computational pipelines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 2733–2736.

[66] Maurer, B. Fail at scale: Reliability in the face of rapid change. *Queue 13*, 8 (2015), 30–46.

[67] Mehta, S., Bhagwan, R., Kumar, R., Bansal, C., Maddila, C., Ashok, B., Asthana, S., Bird, C., and Kumar, A. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th {USENIX} Symposium on Networked Systems Design and Implementation* (2020).

[68] Molyneaux, I. The art of application performance testing: Help for programmers and quality assurance.[sl]:" o'reilly media, 2009.

[69] Mühlbauer, S., Apel, S., and Siegmund, N. Accurate modeling of performance histories for evolving software systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), IEEE, pp. 640–652.

[70] Murashkin, A., Antkiewicz, M., Rayside, D., and Czarnecki, K. Visualization and exploration of optimal variants in product line engineering. In *Proc. Int'l Software Product Line Conference (SPLC)* (2013), ACM, pp. 111–115.

[71] Nair, V., Menzies, T., Siegmund, N., and Apel, S. Faster discovery of faster system configurations with spectral learning. *arXiv:1701.08106* (2017).

[72] Nistor, A., Chang, P.-C., Radoi, C., and Lu, S. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015).

[73] Nistor, A., Jiang, T., and Tan, L. Discovering, reporting, and fixing performance bugs. In *10th working conference on mining software repositories* (2013).

[74] NVIDIA. Nvidia deepstream sdk, 2021.

[75] Ogarrio, J. M., Spirtes, P., and Ramsey, J. A hybrid causal search algorithm for latent variable models. In *Conference on Probabilistic Graphical Models* (2016), pp. 368–379.

[76] Olaechea, R., Rayside, D., Guo, J., and Czarnecki, K. Comparison of exact and approximate multi-objective optimization for software product lines. In *Proc. Int'l Software Product Line Conference (SPLC)* (2014), ACM, pp. 92–101.

[77] Pearl, J. Graphical models for probabilistic and causal reasoning. *Quantified representation of uncertainty and imprecision* (1998), 367–389.

[78] Pearl, J. *Causality*. Cambridge university press, 2009.

[79] Pearl, J., and Mackenzie, D. *The book of why: the new science of cause and effect*. Basic Books, 2018.

[80] Pereira, J. A., Martin, H., Acher, M., Jézéquel, J.-M., Botterweck, G., and Ventresque, A. Learning software configuration spaces: A systematic literature review. *arXiv preprint arXiv:1906.03018* (2019).

[81] PyCausal. Pycausal. https://github.com/bd2kccd/py-causal, 2021.

[82] Reddy, C. M., and Nalini, N. Fault tolerant cloud software systems using software configurations. In *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)* (2016), IEEE, pp. 61–65.

[83] Sánchez, A. B., Delgado-Pérez, P., Medina-Bulo, I., and Segura, S. Tandem: A taxonomy and a dataset of real-world performance bugs. *IEEE Access 8* (2020), 107214–107228.

[84] Scherrer, N., Bilaniuk, O., Annadani, Y., Goyal, A., Schwab, P., Schölkopf, B., Mozer, M. C., Bengio, Y., Bauer, S., and Ke, N. R. Learning neural causal models with active interventions. *arXiv preprint arXiv:2109.02429* (2021).

[85] Schölkopf, B., Locatello, F., Bauer, S., Ke, N. R., Kalchbrenner, N., Goyal, A., and Bengio, Y. Toward causal representation learning. *Proceedings of the IEEE 109*, 5 (2021), 612–634.

[86] Semopy. Semopy. https://semopy.com/, 2021.

[87] Sharma, A., Kiciman, E., et al. DoWhy: A Python package for causal inference. https://github.com/microsoft/dowhy, 2021.

[88] Shimoni, Y., Karavani, E., Ravid, S., Bak, P., Ng, T. H., Alford, S. H., Meade, D., and Goldschmidt, Y. An evaluation toolkit to guide model selection and cohort definition in causal inference. *arXiv preprint arXiv:1906.00442* (2019).

[89] Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. Performance-influence models for highly configurable systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (August 2015), ACM, pp. 284–294.

[90] Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 284–294.

[91] Siegmund, N., Ruckel, N., and Siegmund, J. Dimensions of software configuration: on the configuration context in modern software development. In *Proceedings of the 28th ESEC/FSE* (2020), pp. 338–349.

[92] Song, L., and Lu, S. Statistical debugging for real-world performance problems. *ACM SIGPLAN Notices 49*, 10 (2014), 561–578.

[93] Spirtes, P., Glymour, C. N., Scheines, R., and Heckerman, D. *Causation, prediction, and search*. MIT press, 2000.

[94] Styles, J., Hoos, H. H., and Müller, M. Automatically configuring algorithms for scaling performance. In *Learning and Intelligent Optimization*. Springer, 2012, pp. 205–219.

[95] Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., and Karl, R. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 328–343.

[96] Tsakiltsidis, S., Miranskyy, A., and Mazzawi, E. On automatic detection of performance bugs. In *2016 IEEE international symposium on software reliability engineering workshops (ISSREW)* (2016), IEEE, pp. 132–139.

[97] Uber. Causal ml. https://github.com/uber/causalml, 2021.

[98] Valov, P., Petkovich, J.-C., Guo, J., Fischmeister, S., and Czarnecki, K. Transferring performance prediction models across different hardware platforms. In *Proc. Int'l Conf. on Performance Engineering (ICPE)* (2017), ACM, pp. 39–50.

[99] Velez, M., Jamshidi, P., Sattler, F., Siegmund, N., Apel, S., and Kastner, C. Configcrusher: White-box performance analysis for configurable systems. *arXiv preprint arXiv:1905.02066* (2019).

[100] Velez, M., Jamshidi, P., Siegmund, N., Apel, S., and Kästner, C. White-box analysis over machine learning: Modeling performance of configurable systems. *arXiv preprint arXiv:2101.05362* (2021).

[101] Wang, S., Li, C., Hoffmann, H., Lu, S., Sentosa, W., and Kistijantoro, A. I. Understanding and auto-adjusting performance-sensitive configurations. *ACM SIGPLAN Notices 53*, 2 (2018).

[102] Whitaker, A., Cox, R. S., Gribble, S. D., et al. Configuration debugging as search: Finding the needle in the haystack. In *OSDI* (2004), vol. 4, pp. 6–6.

[103] Wu, F., Weimer, W., Harman, M., Jia, Y., and Krinke, J. Deep parameter optimisation. In *Proc. of the Annual Conference on Genetic and Evolutionary Computation* (2015), ACM, pp. 1375–1382.

[104] Xia, K., Lee, K.-Z., Bengio, Y., and Bareinboim, E. The causal-neural connection: Expressiveness, learnability, and inference.

[105] Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L., and Pasupathy, S. Early detection of configuration errors to reduce failure damage. USENIX Association, pp. 619–634.

[106] Zhang, J., Renganarayana, L., Zhang, X., Ge, N., Bala, V., Xu, T., and Zhou, Y. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014), pp. 687–700.

[107] Zhang, S., and Ernst, M. D. Automated diagnosis of software configuration errors. In *2013 35th International Conference on Software Engineering* (2013).

[108] Zhang, Y., He, H., Legunsen, O., Li, S., Dong, W., and Xu, T. An evolutionary study of configuration design and implementation in cloud systems. In *Proceedings of International Conference on Software Engineering* (2021), ICSE'21.

[109] Zitzler, E., Brockhoff, D., and Thiele, L. The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration. In *International Conference on Evolutionary Multi-Criterion Optimization* (2007), Springer, pp. 862–876.

11 Appendix

11.1 Causal Performance Modeling and Analyses: Motivating Scenarios (Additional details)

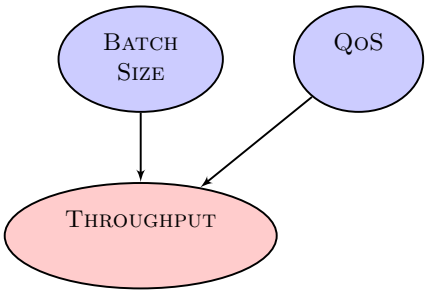


Figure 20. Regression model incorrectly identifies Batch Size and QoS are positively correlated with the term 0.08 Batch Size × QoS. Causal model correctly identifies the dependence relationship between Batch Size and QoS.

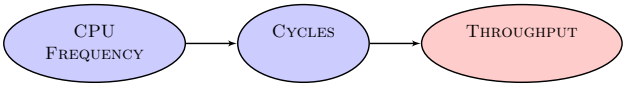


Figure 21. Causal model correctly identifies how CPU Frequency causally influences Throughput via Cycles whereas the regression 0.08 Batch Size × QoS cannot.

Fig. 20 and Fig. 21 present additional scenarios where performance influence models could produce incorrect explanations. The regression terms presented here incorrectly identify spurious correlations, whereas the causal model correctly identifies the cause-effect relationships.

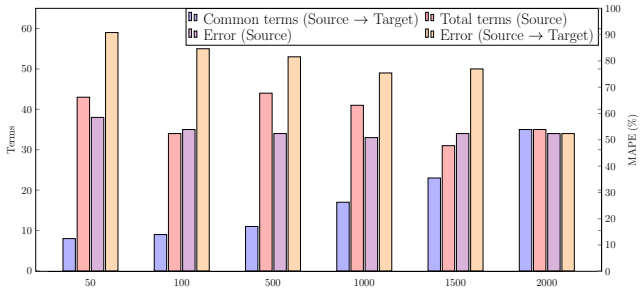


Figure 22. Performance influence models relying on correlational statistics are not stable as new samples are added and do no generalize well. Common terms refers to the individual predictors (i.e., options and interactions) in the performance models that are similar across envirnments.

Performance behavior of regression models for configurable systems varies when sample size varies. Fig. 22 shows

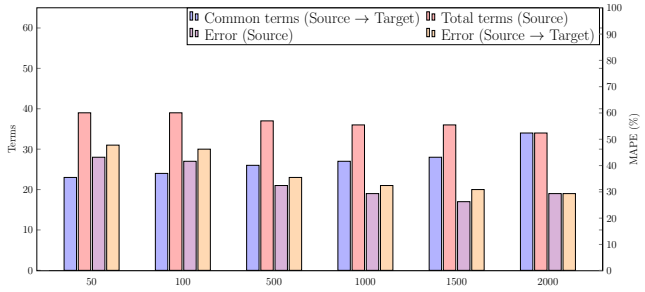


Figure 23. Causal performance models are relatively more stable as new samples are added and do generalize well.

the change of number of stable terms and error with different number of samples for building a performance influence models. Here, we vary the number of samples from 50 to 1500 to build a source regression model. We use sample size 2000 to build the target regression model. We observe that regression models cannot be reliably used in performance tasks, as they are sensitive to the number of training samples. The results indicate that this model classes as opposed to causal models cannot identify causal variables underlying system performance, so depending on the training sample, they try to find the best predictor to increase the prediction power with the i.i.d. assumption that does not hold in system performance. On the contrary, the number of stable predictor's variation is less in causal performance models and lead to better generalization as shown in Fig. 23. In addition to the number of stable predictors, the difference in error between source and target is negligible when compared to the performance regression models.

Extraction of predictor terms from the Causal Performance Model The constructed CPMs have performance objective nodes at the bottom (leaf nodes) and configuration options nodes at the top level. The intermediate levels are filled with the system events. To extract a causal term from the causal model, we backtrack starting from the performance objective until we reach a configuration option. If there are more than one path through a system event from performance objective to configuration options, we consider all possible interaction between those configuration options to calculate the number of causal terms.

11.2 Unicorn (Additional details)

Here, we explain some extra details in several stages in Unicorn to enable replicability of our approach.

Stage-II: Learn Causal Performance Model

In this section, we describe the edge orientation principles used in Unicorn.

Orienting undirected causal links. We orient undirected edges using prescribed edge orientation rules [17, 18, 31, 75, 93] to produce a *partial ancestral graph* (or PAG). A PAG contains the following types of (partially) directed edges:

- $X \longrightarrow Y$ indicating that vertex X causes Y.
- $X \longleftrightarrow Y$ which indicates that there are unmeasured confounders between vertices X and Y.

In addition, a PAG produces two types of partially directed edges:

- $X \circ\!\!\rightarrow Y$ indicating that either X causes Y, or that there are *unmeasured confounders* that cause both X and Y.
- $X \circ\!\!-\!\!\circ Y$ which indicates that either: (a) vertices X causes Y, or (b) vertex Y causes X, or (c) there are *unmeasured confounders* that cause both X and Y.

In the last two cases, the circle ($\circ$) indicates that there is an ambiguity in the edge type. In other words, given the current observational data, the circle can indicate an arrowhead ($\longrightarrow$) or no arrow head ($-$), i.e., for $X \circ\!\!-\!\!\circ Y$, all three of $X \longrightarrow Y$, $Y \longrightarrow X$, and $X \longleftrightarrow Y$ might be compatible with current data, i.e., the current data could be faithful to each of these statistically equivalent causal graphs inducing the same conditional independence relationships.

Resolving partially directed edges. For subsequent analyses over the causal graph, the PAG obtained must be fully resolved (directed with no $\circ$ ended edges) in order to generate an ADMG, i.e., we must fully orient partially directed edges by replacing the circles in $\circ\!\!\rightarrow$ and $\circ\!\!-\!\!\circ$ with the correct edge direction. We use the information-theoretic approach using entropy proposed in [60, 61] to discover the true causal direction between two variables. Entropic causal discovery is inspired by Occam's razor, and the key intuition is that, among the possible orientations induced by partially directed edges (i.e., $\circ\!\!\rightarrow$ and $\circ\!\!-\!\!\circ$), the most plausible orientation is that which has the lowest entropy.

Our work extends the theoretic underpinnings of entropic causal discovery to generate a fully directed causal graph by resolving the partially directed edges produced by FCI. For each partially directed edge, we follow two steps: (1) establish if we can generate a latent variable (with low entropy) to serve as a common cause between two vertices; (2) if such a latent variable does not exist, then pick the causal direction which has the lowest entropy.

For the first step, we assess if there could be an unmeasured confounder (say Z) that lies between two partially oriented nodes (say X and Y). For this, we use the *LatentSearch* algorithm proposed by Kocaoglu *et al.* [61]. *LatentSearch* outputs a joint distribution $q(X, Y, Z)$ of the variables X, Y, and Z which can be used to compute the entropy $H(Z)$ of the unmeasured confounder Z. Following the guidelines of Kocaoglu *et al.*, we set an entropy threshold $\theta_r = 0.8 \times min\{H(X), H(Y)\}$. If the entropy $H(Z)$ of the unmeasured confounder falls *below* this threshold, then we declare that there is a simple unmeasured confounder Z (with a low enough entropy) to serve as a common cause between X and Y and accordingly, we replace the partial edge with a bidirected (i.e., $\longleftrightarrow$) edge.

When there is no latent variable with a sufficiently low entropy, two possibilities exist: *(a)* variable X causes Y; then,

there is an arbitrary function $f(\cdot)$ such that $Y = f(X, E)$, where E is an exogenous variable (independent of X) that accounts for system noise; or *(b)* variable Y causes X; then, there is an arbitrary function $g(\cdot)$ such that $X = g(Y, \tilde{E})$, where $\tilde{E}$ is an exogenous variable (independent of Y) that accounts for noise in the system. The distribution of E and $\tilde{E}$ can be inferred from the data [60, see §3.1]. With these distributions, we measure the entropies $H(E)$ and $H(\tilde{E})$. If $H(E) < H(\tilde{E})$, then, it is simpler to explain the $X \longrightarrow Y$ (i.e., the entropy is lower when $Y = f(X, E)$) and we choose $X \longrightarrow Y$. Otherwise, we choose $Y \longrightarrow X$.

Example. Fig. 10 shows the steps involved in generating the final ADMG. First, we build a complete undirected graph by connecting all pairs of variables with an undirected egde,where only a small subset of connections are shown for readability). Next, we use Fisher's exact test [19] to evaluate the independence of all pairs of variables conditioned on all remaining variables. Pruning edges between the independent variables results in skeleton graph. Next, we orient undirected edges using edge orientation rules [18, 31, 75, 93] to produce a partial ancestral graph. In our example, we identify that there are two edges that are partially oriented: *(i)* Cache Misses $\circ\!\!-\!\!\circ$ No of Cycles; and *(ii)* Cache Misses $\circ\!\!\longrightarrow$ Energy. To resolve these two edges, we use the entropic orientation strategy to orient these edges to get the final ADMG.

11.3 Stage-III: Iterative Sampling.

We extract paths from the causal graph (referred to as *causal paths*) and rank them from highest to lowest based on their average causal effect on latency, and energy. Using path extraction and ranking, we reduce the complex causal graph into a few useful causal paths for further analyses. The configurations in this path are more likely to be associated with the root cause of the fault.

Extracting causal paths with backtracking. A causal path is a directed path originating from either the configuration options or the system event and terminating at a non-functional property (i.e., throughput and/or energy). To discover causal paths, we backtrack from the nodes corresponding to each non-functional property until we reach a node with no parents. If any intermediate node has more than one parent, then we create a path for each parent and continue backtracking on each parent.

Ranking causal paths. A complex causal graph can result in a large number of causal paths. It is not practical to reason over all possible paths as it may lead to a combinatorial explosion. Therefore, we rank the paths in descending of their causal effect on each non-functional property. For further analysis, we use paths with the highest causal effect. To rank the paths, we measure the causal effect of changing the value of one node (say Batch size or X) on its successor (say Cache Misses or Z) in the path (say Batch size $\longrightarrow$ Cache Misses $\longrightarrow$ FPS and Energy). We express this with the *do-calculus* [78] notation: $\mathbb{E}[Z \mid do(X = x)]$. This notation represents the expected value of Z (Cache Misses) if we set the value of the node X (Batch size) to x. To compute the *average causal effect* (ACE) of $X \rightarrow Z$ (i.e., Batch size $\longrightarrow$ Cache Misses), we find the average effect over all permissible values of X (Batch size), i.e.,

$$\text{ACE}(Z, X) = \frac{1}{N} \cdot \sum_{\forall a, b \in X} \mathbb{E}[Z \mid do(X = b)] - \mathbb{E}[Z \mid do(X = a)] \quad (1)$$

Here, N represents the total number of values X (Batch size) can take. If changes in Batch size result in a large change in Cache Misses, then ACE (Z, X) will be larger, indicating that Batch size on average has a large causal effect on Cache Misses. Note, if X is a continuous variable, we would replace the summation of Eq. (1) with an integral. For the entire path, we extend Eq. (1) as:

$$\text{Path}_{ACE} = \frac{1}{K} \cdot \sum \text{ACE}(Z, X) \qquad \forall X, Z \in path \quad (2)$$

Eq. (2) represents the average causal effect of the causal path. The configuration options that lie in paths with larger P_{ACE} tend to have a greater causal effect on the corresponding non-functional properties in those paths. We select the top K paths with the largest P_{ACE} values, for each non-functional property. In this paper, we use K=3, 5,7 and 9, however, this may be modified in our replication package.

Counterfactual queries can be different for different tasks. For debugging, we use the top K paths to (a) identify the root cause of non-functional faults; and (b) prescribe ways to fix the non-functional faults. Similarly, we use the top K paths to identify the options that can improve the non-functional property values near optimal. For both tasks, a developer may ask specific queries to Unicorn and expect an actionable response. (Say) For debugging, we use the example causal graph of where a developer observes low FPS and high energy, i.e., a multi-objective fault, and has the following questions:

"What are the root causes of my multi-objective (FPS and energy) fault?" To identify the root cause of a non-functional fault, we must identify which configuration options have the most causal effect on the performance objective. For this, we use the steps outlined in §3 to extract the paths from the causal graph and rank the paths based on their average causal effect (i.e., Path_{ACE} from Eq. (2)) on latency and energy. We return the configurations that lie on the top K paths. For example, in Fig. 7 we may return (say) the following paths:

- Batch size $\longrightarrow$ Cache Misses $\longrightarrow$ FPS and Energy
- Enable Padding $\longrightarrow$ Cache Misses $\longrightarrow$ FPS and Energy

and the configuration options Batch size, and Enable Padding being the probable root causes.

"How to improve my FPS and energy?" To answer this query, we first find the root causes as described above. Next, we discover what values each of the configuration options

must take in order that the new FPS and energy is better (high FPS and low energy) than the fault (low FPS and high energy). For example, we consider the causal path Batch size ⟶ Cache Misses ⟶ FPS and Energy, we identify the permitted values for the configuration options Batch size that can result in a high FPS and energy (Y^{LOW}) that is better than the fault (Y^{HIGH}). For this, we formulate the following counterfactual expression:

$$\Pr(Y^{\text{LOW}}_{repair} | \neg repair, Y^{\text{HIGH}}_{\neg repair}) \quad (3)$$

Eq. (3) measures the probability of "fixing" the latency fault with a "repair" (Y^{LOW}_{repair}) given that with no repair we observed the fault ($Y^{\text{HIGH}}_{\neg repair}$). In our example, the repairs would resemble Batch size=10. We generate a *repair set* ($\mathcal{R}_1$), where the configurations Batch size is set to all permissible values, i.e.,

$$\mathcal{R}_1 \equiv \bigcup \{\text{Batch size} = x, ...\} \forall x \in \text{Batch size} \quad (4)$$

Observe that, in the repair set ($\mathcal{R}_1$) a configuration option that is not on the path Batch size ⟶ Cache Misses ⟶ FPS and Energy is set to the same value of the fault. For example, Bit rate is set to 2 or Enable Padding is set to 1. This way we can reason about the effect of interactions between Batch size with other options, i.e., Bit rate, Buffer size. Say Buffer size or Enable padding were changed/recommended to set at any other value than the fault in some previous iteration i.e., 20 or 0, respectively. In that case, we set Buffer size and Enable padding=0. Similarly, we generate a repair set $\mathcal{R}_2$ by setting Enable Paddingto all permissible values.

$$\mathcal{R}_2 \equiv \bigcup \{\text{Enable padding} = x, ...\} \forall x \in \text{Enable padding} \quad (5)$$

Now, we combine the repair set for each path to construct a final repair set $\mathcal{R} = \mathcal{R}_1 \cup \ldots \cup \mathcal{R}_k$. Next, we compute the *individual treatment effect* (ITE) on the FPS and energy (Y) for each repair in the repair set $\mathcal{R}$. In our case, for each repair $r \in \mathcal{R}$, ITE is given by:

$$\text{ITE}(r) = \Pr(Y^{\text{LOW}}_{r} \mid \neg r,\ Y^{\text{HIGH}}_{\neg r}) - \Pr(Y^{\text{HIGH}}_{r} \mid \neg r,\ Y^{\text{HIGH}}_{\neg r}) \quad (6)$$

ITE measures the difference between the probability that the latency and energy *is low* after a repair r and the probability that the FPS and energy is *still high* after a repair r. If this difference is positive, then the repair has a higher chance of fixing the fault. In contrast, if the difference is negative, then that repair will likely worsen both FPS and energy. To find the most useful repair ($\mathcal{R}_{best}$), we find a repair with the largest (positive) ITE, i.e., $\mathcal{R}_{best} = \text{argmax}_{\forall r \in \mathcal{R}}[\text{ITE}(r)]$. This provides the developer with a possible repair for the configuration options that can fix the multi-objective latency and energy fault.

Remarks. The ITE computation of Eq. (6) occurs *only* on the observational data. Therefore we may generate any number of repairs and reason about them without having to deploy those interventions and measuring their performance in the real world. This offers significant runtime benefits.

Table 4. Deepstream software configuration options.

Configuration Options	Values/Range
CRF	13, 18, 24, 30
Bit Rate	1000, 2000, 2800, 5000
Buffer Size	6000, 8000, 20000
Presets	ultrafast, veryfast, faster medium, slower
Maxrate	600k, 1000k
Batch Size	0-30
Batched Push Timeout	0-20
Enable Padding	0,1
Buffer Pool Size	1-26
Sync Inputs	0,1
Nvbuf Memory Type	0,1,2,3
Net Scale Factor	0.01 -10
Batch Size	1-60
Interval	1-20
Offset	0,1
Process Mode	0,1
Use DLA Core	0,1
Enable DLA	0,1
Enable dbscan	0,1
Secondary Reinfer Interval	0-20
Maintain Aspect Ratio	0,1
IOU Threshold	0-60
Enable Batch Process	0,1
Enable Past Frame	0,1
Compute hw	0,1,2,3,4

11.4 Experimental setup (Additional details)

Table 4, Table 5, Table 6, and Table 7, show different software configuration options and their values for different systems considered in this paper. Table 8 shows the OS/kernel level configuration options and their values for different systems considered in this paper. Additionally, Table 10 shows the perf events considered in this paper. The hyperparameters considered for Xception, Bert, and Deepspeech are shown in Table 11.

We used the following four components for Deepstream implementation:

- **Decoder**: For the decoder we use x264. It uses the x264 and takes the encoded H.64, VP8, VP9 streams and produces a NV12 stream.
- **Stream Mux**: The streammux module takes the NV12 stream and outputs the NV12 batched buffer with information about input frames, including original timestamp and frame number.
- **Nvinfer**: For object detection and classification, we use the TrafficCamNet model that uses ResNet 18 architecture. This model is pre-trained in 4 classes on a dataset of 150k frames and has an accuracy of 83.5% for detecting and tracking cars from a traffic camera's

Table 5. Xception, Bert, and Deepspeech, software configuration options.

Configuration Options	Range
Memory Growth	0-1
Logical Devices	0,1

Table 6. x264 software configuration options.

Configuration Options	Values/Range
CRF	13, 18, 24, 30
Bit Rate	1000, 2000, 2800, 5000
Buffer Size	6000, 8000, 20000
Presets	ultrafast, veryfast, faster medium, slower
Maxrate	600k, 1000k

Table 7. sqlite software configuration options.

Configuration Options	Range
PRAGMA TEMP_STORE	DEFAULT, FILE, MEMORY
PRAGMA JOURNAL_MODE	DELETE, TRUNCATE,PERSIST,MEMORY, OFF
PRAGMA SYNCHRONOUS	FULL, NORMAL, OFF
PRAGMA LOCKING_MODE	NORMAL, EXCLUSIVE
PRAGMA CACHE_SIZE	0,1000,2000,4000,10000
PRAGMA PAGE_SIZE	2048,4096,8192
PRAGMA MAX_PAGE_COUNT	32,64

viewpoint. The 4 classes are Vehicle, BiCycle, Person, and Roadsign. We use the Keras (Tensorflow backend) pre-trained model from TensorRT.

- **Nvtracker**: The plugin accepts NV12- or RGBA-formated frame data from the upstream component and scales (converts) the input buffer to a buffer in the format required by the low-level library, with tracker width and height. NvDCF tracker uses a correlation filter-based online discriminative learning algorithm as a visual object tracker, while using a data association algorithm for multi-object tracking.

Table 5 shows different software configuration options and their values for each components considered in this paper.

Table 6 shows different software configuration options and their values for each component considered in this paper.

11.5 Case Study (Additional details)

Fig. 25 shows the causal graph to resolve the real world latency fault.

11.6 Effectiveness (Additional Results)

Table 13(a) shows the effectiveness of Unicorn in resolving single objective faults due to heat in NVIDIA TX1. Here, Unicorn outperforms correlation-based methods in all cases. For example, in Bert on TX1, Unicorn achieves 9% more accuracy, 11% more precision, and 10% more recall compared to the next best method, BugDoc. We observed heat gains as high as 7% (2% more than BugDoc) on x264. The results confirm that Unicorn *can recommend repairs for faults that significantly improve latency and energy usage.* Applying the changes to the configurations recommended by Unicorn increases the performance drastically.

Table 8. Linux OS/Kernel configuration options.

Configuration Options	Range
vm.vfs_cache_pressure	1,100,500
vm.swappiness	10,60,90
vm.dirty_bytes	30,60
vm.dirty_background_ratio	10,80
vm.dirty_background_bytes	30,60
vm.dirty_ratio	5,50
vm.nr_hugepages	0,1,2
vm.overcommit_ratio	50,80
vm.overcommit_memory	0,2
vm.overcommit_hugepages	0,1,2
kernel.cpu_time_max_percent	10-100
kernel.max_queues	32-512
kernel.max_pids	32768,65536
kernel.numa_balancing	0,1
kernel.sched_latency_ns	24000000,48000000
kernel.sched_nr_migrate	32,64,128
kernel.sched_rt_period_us	1000000,2000000
kernel.sched_rt_runtime_us	500000,950000
kernel.sched_time_avg_ms	1000,2000
kernel.sched_child_runs_first	0,1
Swap memory	1,2,3,4 (GB)
Scheduler Policy	CFP,NOOP
Drop caches	0,1,2,3

Table 9. Hardware configuration options.

Configuration Options	Range	Description
CPU Cores	1-4	
CPU Frequency	0.3 - 2.0 (GHz)	
GPU Frequency	0.1-1.3 (GHz)	
EMC Frequency	0.1-1.8 (Ghz)	

Unicorn *can resolve misconfiguration faults significantly faster than correlation-based approaches.* In Table 13, the last two columns indicate the time taken (in hours) by each approach to diagnosing the root cause. Unicorn can do resolve faults significantly faster, e.g., Unicorn is 13× faster in diagnosing and resolving latency and heat faults for Deepspeech.

11.7 Transferability (Additional Results)

Table. 15 indicates the results for different transfer scenarios: (I) We learn a causal model from TX1 and use them to resolve the latency faults in TX2, (I) We learn a causal model from

Table 10. Performance system events and tracepoints.

Events
Context switches
Major faults
Minor faults
Migrations
Scheduler wait time
Scheduler sleep time
Cycles
Instructions
Number of syscall enter
Number of syscall exit
L1 dcache load misses
L1 dcache loads
L1 dcache stores
Branch loads
Branch loads misses
Branch misses
Cache references
Cache misses
L1 dcache stores
Tracepoint subsystems
Block
Scheduler
IRQ
ext4

Table 11. Hyperparameters for DNNs used in Unicorn.

Hyperparameters	Range	Architecture
Number of filters entry flow	32	
Filter size entry flow	(3×3)	
Number of filters middle flow	64	
Filter size middle flow	(3×3)	Xception
Number of filters exit flow	728	
Filter size exit flow	(3×3)	
Batch Size	32	
Number of epochs	100	
Dropout	0.3	
Maximum batch size	16	
Maximum sequence length	13	Bert
Learning rate	$1e^{-4}$	
Weight decay	0.3	
Dropout	0.3	
Maximum batch size	16	DeepSpeech
Maximum sequence length	32	
Learning rate	$1e^{-4}$	
Number of epochs	10	

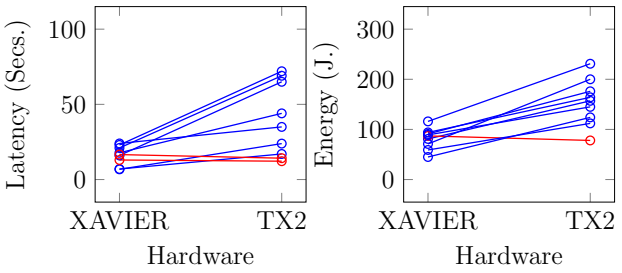


Figure 24. Ranking of configurations may change across environments, here between two hardware. The reason can be associated to differences in microarchitecture and different hardware resources. However, causal performance models capture the underlying causal mechanisms and therefore are able to capture the causal mechanisms and use them for performance related tasks in the new environments. On the other hand, performance influence models need to relearn the patterns from scratch, therefore, they demand for m ore sample in the new environments.

Table 12. Hyperparameters for FCI used in Unicorn.

Hyperparameters	Value
depth	-1
testId	fisher-z-test
maxPathLength	-1
completeRuleSetUsed	False

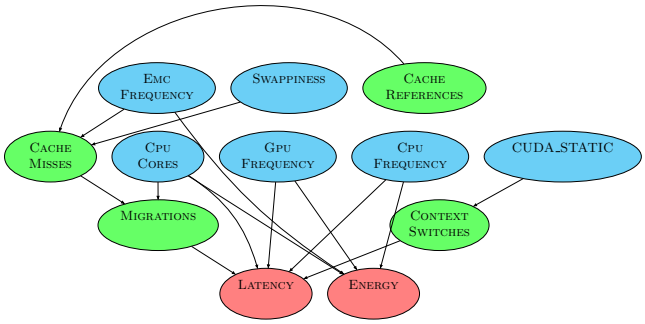


Figure 25. Causal graph used to resolve the latency fault in the real world case study.

TX2 and use them to resolve the energy faults in Xavier, and (III) We learn a causal model from Xavier and use them to resolve the heat faults in TX1. Here, we determine how transferable is Unicorn by comparing with Unicorn (Reuse), Unicorn +25, and Unicorn (Rerun). For all systems, we observe that performance of Unicorn (Reuse) is close to the performance of Unicorn (Rerun) which confirms the high transferability property of Unicorn. For example, in Xception and sqlite, Unicorn (Reuse) has the exact gain as

Table 13. Efficiency of UNICORN compared to other approaches. Cells highlighted in **blue** indicate improvement over faults.

(a) Single objective performance fault in heat.

			Accuracy					Precision					Recall					Gain					Time†	
			UNICORN	CBI	DD	ENCORE	BUGDOC	UNICORN	CBI	DD	ENCORE	BUGDOC	UNICORN	CBI	DD	ENCORE	BUGDOC	UNICORN	CBI	DD	ENCORE	BUGDOC	UNICORN	Others
TX1	Heat	XCEPTION	**69**	63	57	64	65	**75**	56	56	60	66	**68**	62	58	64	69	**4**	3	2	2	3	**0.6**	4
		BERT	**71**	62	61	61	62	**72**	56	59	56	61	**72**	65	62	67	62	**5**	3	2	2	3	**0.4**	4
		DEEPSPEECH	**71**	61	64	62	67	**71**	58	59	54	68	**69**	67	66	68	67	**3**	3	2	2	2	**0.7**	4
		x264	**74**	65	57	64	65	**74**	62	54	55	65	**74**	66	63	68	69	**7**	3	2	2	5	**1.4**	4

(b) Multi-objective non-functional faults in *Heat, Latency.*

		Accuracy				Precision				Recall				Gain (Latency)				Gain (Heat)				Time†	
		UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	Others
Latency + Heat	XCEPTION	**62**	52	55	57	**69**	57	50	61	**61**	48	51	60	**58**	42	47	51	**2**	1	1	1	**0.9**	4
	BERT	**64**	52	47	56	**62**	52	45	60	**68**		62	65	**65**	37	48	60	**4**	3	2	3	**0.4**	4
	DEEPSPEECH	**62**	52	43	55	**60**	48	48	55	**67**	58	41	59	**69**	37	45	65	**4**	1	1	4	**0.3**	4
	x264	**61**	53	53	60	**63**	50	54	61	**60**	53	55	55	**67**	54	54	65	**5**	3	3	4	**0.5**	4

(c) Multi-objective non-functional faults in *Energy, Heat.*

		Accuracy				Precision				Recall				Gain (Energy)				Gain (Heat)				Time†	
		UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	Others
Energy + Heat	XCEPTION	**65**	52	55	57	**64**	57	50	61	**67**	48	51	60	**58**	42	47	51	**2**	1	1	1	**0.9**	4
	BERT	**69**	52	47	56	**65**	52	45	60	**71**		62	65	**65**	37	48	60	**4**	3	2	3	**0.4**	4
	DEEPSPEECH	**72**	52	43	55	**73**	48	48	55	**69**	58	41	59	**69**	37	45	65	**4**	1	1	4	**0.3**	4
	x264	**71**	53	53	60	**73**	50	54	61	**65**	53	55	55	**67**	54	54	65	**5**	3	3	4	**0.5**	4

Table 14. Efficiency of UNICORN in detecting and repairing the root-cause of multiple non-functional faults: and *Energy, Latency, Heat.* Cells highlighted in green indicate improvement over faults and red indicate deterioration. UNICORN achieves better performance overall and is much faster. Note: the results are reported for NVIDIA Jetson TX2.

			Accuracy				Precision				Recall				Gain (Latency)				Gain (Energy)				Gain (Heat)				Time†	
			UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	CBI	ENCORE	BUGDOC	UNICORN	Others
All	Three	Image	**76**	57	48	66	**68**	61	57	61	**81**	53	46	70	**62**	33	30	42	**52**	23	18	24	**4**	1	0	0	**0.1**	4
		x264	**80**	59	47	54	**76**	61	56	63	**81**	56	46	51	**12**	2	1	2	**15**	4	2	4	**4**	1	0	1	**0.1**	4
		SQLite	**73**	56	51	53	**68**	59	56	60	**78**	54	45	51	**12**	1	1	4	**8**	4	2	5	**1**	1	-1	-1	**0.1**	4

† Wallclock time in hours

of UNICORN (Rerun) for heat faults. For latency and energy faults, the gain difference between UNICORN (Reuse) and UNICORN (Rerun) is less than 5% for all systems. We also observe that with little updates, UNICORN +25 (24 minutes) achieves a similar performance of UNICORN (RERUN) (40 minutes), on average. This confirms that as the causal mechanisms are sparse, the CPM from source in UNICORN quickly reaches a fixed structure in the target using incremental learning by judiciously evaluating the most promising fixes until the fault is resolved.

11.8 Scalability (Additional deatails)

Scalability of UNICORN depends on the scalability of each phase. Therefore, we design scenarios to test the scalability of each phase to determine the overall scalability. Since the initial number of samples and the underlying phases for each

Table 15. (RQ3) Transferring causal models across hardware platforms.

		Accuracy			Recall			Precision			Δ_{gain}		
	Software	Unicorn (Reuse)	Unicorn +25	Unicorn (Rerun)	Unicorn (Reuse)	Unicorn +25	Unicorn (Rerun)	Unicorn (Reuse)	Unicorn +25	Unicorn (Rerun)	Unicorn (Reuse)	Unicorn +25	Unicorn (Rerun)
TX1 (source) ⟶ TX2 (target)													
Latency	Xception	52	83	88	70	79	86	46	78	91	46	71	74
	DeepSpeech	55	72	76	57	79	75	45	57	81	33	75	79
	Bert	45	61	64	56	79	62	49	56	74	54	88	94
	x264	47	79	83	70	79	82	58	67	85	5	23	28
	SQLite	59	72	80	68	76	78	52	64	84	10	24	31
TX2 (source) ⟶ Xavier (target)													
Energy	Xception	53	70	84	48	71	86	51	59	78	30	83	88
	DeepSpeech	50	71	77	53	75	80	49	68	74	40	61	67
	Bert	57	70	73	45	74	70	43	53	77	42	71	76
	x264	54	72	79	46	79	77	42	65	83	6	21	24
	SQLite	45	75	82	48	79	70	47	63	81	7	16	23
Xavier (source) ⟶ TX1 (target)													
Heat	Xception	53	62	65	69	61	80	58	64	64	3	4	4
	DeepSpeech	55	65	69	69	59	76	52	63	69	3	3	4
	Bert	57	64	66	69	63	81	53	63	63	3	5	7
	x264	51	65	68	73	51	76	54	71	66	1	4	7
	SQLite	54	70	70	70	74	75	48	66	66	1	6	6

task is the same, it is sufficient to examine the scalability of Unicorn for the debugging non-functional fault task.

sqlite was chosen because it offers a large number of configurable options, much more than DNN-based applications, and video encoders. Further, each of these options can take on a large number of permitted values, making Deepstream a useful candidate to study the scalability of Unicorn. Deepstream was chosen as it has a higher number of components than others, and it is interesting to determine how Unicorn behaves when the number of options and events are increasing. As a result, SQLite exposes new system design opportunities to enable efficient inference and many complex interactions between software options.

In large systems, there are significantly more causal paths and therefore, causal learning and estimations of queries take more time. However, with as much as 242 configuration options and 19 events (Table 3, row 2), causal graph discovery takes roughly one minute, evaluating all 2234 queries takes roughly two minutes, and the total time to diagnose and fix a fault is roughly 22 minutes for sqlite. This trend is observed even with 242 configuration options, 288 events (Table 3, row 3), and finer granularity of configuration values—the time required to causal model recovery is a little over 1 minute and the total time to diagnose and fix a fault is less than 2 hours. Similarly, in Deepstream, with 53 configuration options and 288 events, causal model discovery is less than two minutes and the time needed to diagnose and fix a fault is less than an hour. The results in Table 3 indicate that Unicorn can scale to a much larger configuration space without an exponential increase in runtime for any of the intermediate stages. This can be attributed to the sparsity of the causal graph (average degree of a node for sqlite in Table 3 is at most 3.6, and it reduces to 1.6 when the number of configurations increase and reduces from 3.1 to 2.3 in Deepstream when systems events are increased). This makes sense because not all variables (i.e., configuration options and/or system events) affect non-functional properties and a high number of variables in the graph end up as isolated nodes. Therefore, the number of paths and consequently the evaluation time do not grow exponentially as the number of variables increase.

Finally, the latency gain associated with repairs from larger configuration space with configurations was similar to the original space of 34 and 53 configurations for sqlite and Deepstream, respectively. This indicates that: (a) imparting domain expertise to select most important configuration options can speed up the inference time of Unicorn, and (b) if the user chooses instead to use more configuration options (perhaps to avoid initial feature engineering), Unicorn can still diagnose and fix faults satisfactorily within a reasonable time.