CADET: A Systematic Method For Debugging Misconfigurations using Counterfactual Reasoning

Md 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 computing platforms are highly-configurable with hundreds of interacting configuration options. However, configuring these systems is challenging. Erroneous configurations can cause unexpected non-functional faults resulting in significant performance degradation in non-functional system properties like latency, energy consumption, heat dissipation, etc. This paper proposes CADET (short for Causal Debugging Toolkit)—a method that enables users to *identify, explain*, and *fix* the root cause of non-functional faults early and in a principled fashion. CADET builds a causal model by observing the performance of the system under different configurations. Then, it uses causal path extraction followed by counterfactual reasoning over the causal model to (a) identify the root causes of non-functional faults, (b) estimate the effects of various configuration options on the non-functional system properties, and (c) prescribe candidate repairs to the relevant configuration options to fix the non-functional faults. We evaluate CADET on 5 highly-configurable software systems deployed on 3 NVIDIA Jetson hardware platforms. We compare CADET with four state-of-the-art machine learning (ML)-based debugging approaches. The experimental results indicate that CADET can find repairs for faults with (on average) 8% better accuracy in multiple non-functional properties 7× faster than the next best performance debugging method.

1 Introduction

Modern computer systems are composed of *multiple components*, each of which are *highly-configurable*, and are increasingly being deployed on *heterogeneous hardware platforms* (e.g., System-on-a-Chip, System-on-Module, IoT devices, cloud platforms) with *different deployment configurations* (local, distributed, multi-cloud). For example, most modern ML systems, cyber-physical systems, self-driving cars, robotics, and big data systems have such characteristics. The configuration space in such systems is combinatorially large with thousands of software and hardware configuration options that interact non-trivially with one another [1, 2, 3]. Unfortunately, configuring these systems to achieve specific goals is challenging and error-prone. Incorrect configurations (*misconfigurations*) happen as a result of unexpected interactions between software and hardware configuration options across the system stack resulting in *non-functional faults*, i.e., faults in terms of *non-functional* system properties such as latency, energy consumption, and/or heat dissipation. These non-functional faults—unlike regular software bugs—do not cause the system to crash or

*Joint First author.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

exhibit an obvious misbehavior [4, 5, 6]. Instead, misconfigured systems remain operational while being compromised, resulting in severe performance degradation in latency, energy consumption, and/or heat dissipation [7, 8, 9, 10]. The sheer number of modalities of software deployment is so large that exhaustively testing every conceivable software and hardware configuration is impossible.

Consequently, identifying the root cause of non-functional faults is notoriously difficult [11] with as much as 99% of them going unnoticed or unreported for extended durations [12]. Non-functional faults have tremendous monetary repercussions costing companies worldwide an estimated $5 trillion in 2018 and 2019 [13]. They also dominate discussions on online forums where some developers are quite vocal in expressing their dissatisfaction [14, 15]. Therefore, we seek methods that can *identify, explain, and fix* the root cause of non-functional faults early and in a principled fashion.

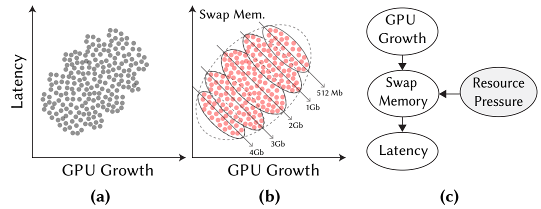


Figure 1: Observational data (in Fig. 1a) (incorrectly) shows that high `GPU growth` leads to high latency. The trend is reversed when the data is segregated by `swap memory` (Fig. 1b). Causal model constructed on the observational data indicates that `GPU growth` indirectly influences latency via `swap memory` (Fig. 1c).

Existing work. Much recent work has focused on *configuration optimization* aimed at finding a near-optimal configuration that optimizes a performance objective [16, 17, 18, 19]. Finding the optimum configuration using push-button optimization approaches are not applicable here because we tackle an essentially different problem—to find and repair the root causes of an *already observed* non-functional fault. The global optima do not give us any information about the underlying interactions between the faulty configuration options that caused the non-functional fault. This information is sought after by developers seeking to address these non-functional faults [4, 20].

Some previous works have used ML-based performance modeling in fixed [21, 22, 23, 24, 25, 26, 27] and variable environments [28, 29, 30, 31]. Several works attempted to debug systems using noisy logs [32], anomaly diagnosis [33, 34], sampling [2], data-driven approaches [35, 36, 37, 38], explanation tables [39], query-based diagnosis [40], statistical debugging and association rule mining based approaches [41, 42, 43, 44, 45, 46], and similarity analysis [47]. These approaches are adept at describing *if* certain configuration options influence performance, however, they lack the mathematical language to express *how* or *why* the configuration options affect performance. Without this knowledge, we risk drawing misleading conclusions. They also require significant time to gather the training samples, and this time grows exponentially with the number of configurations [48, 1]. Recent work has employed causal inference for detecting *functional* bugs (Holmes [49]) and intermittent failures of databases (AID [50]). These works are orthogonal to performance debugging of highly-configurable systems.

Limitations of existing work. In Fig. 1, we present an example to help illustrate the limitations of the current techniques. Here, the observational data gathered so far indicates that a configuration option `GPU growth` is positively correlated with increased latency (as in Fig. 1a). A black-box ML-model built on this data will, with high confidence, predict that larger `GPU growth` leads to larger latency. However, this is counter-intuitive because higher `GPU growth` should, in theory, reduce latency not increase it. When we segregate the same data on `swap memory` (as in Fig. 1b), we see that there is indeed a general downward trend for latency, i.e., within each group of `swap memory`, as `GPU growth` increases the latency decreases. We expect this because `GPU growth` controls how much memory the GPU can "borrow" from the `swap memory`. Depending on resource pressure imposed by other host processes, a resource manager may arbitrarily re-allocate some `swap memory`; this means the GPU borrows proportionately more/less swap memory thereby affecting the latency correspondingly. This is reflected by the data in Fig. 1b. If the ML-based model were to consult the available data (from Fig. 1a) unaware of such underlying causal structure, these models would be incorrect. With thousands of configurations, inferring such nuanced information from optimization or ML-based approaches would require a considerable amount of measurements and extensive domain expertise which can be impractical, if not impossible, to possess in practice.

Our approach. In this paper, we propose the use of causal models [51, 52] to express the complex interactions between the configuration options and the performance objectives with causal models—a

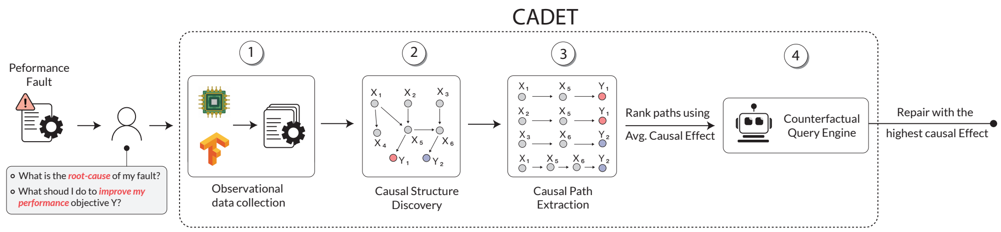


Figure 2: Overview of CADET.

rich graphical representation that can be learned from performance data and that can be reasoned upon using the mathematics of causal inference. To this end, we design, implement and evaluate CADET that uses causal structural discovery algorithms [53, 54] to construct a causal model using observational data. Then, it uses counterfactual reasoning [55] over the causal model to (a) identify the root causes of non-functional faults, (b) estimate the effects of various configurable parameters on the non-functional properties(s), and (c) prescribe candidate repairs to the relevant configuration options to fix the non-functional fault. In the example of Fig. 1, CADET constructs a *causal model* from observational data (as in Fig. 1c). This causal model indicates that `GPU growth` indirectly influences latency (via a `swap memory`) and that the configuration options may be affected by certain other factors, e.g., resource manager allocating resources for other processes running on the host. CADET uses counterfactual questions such as, "what is the effect of `GPU growth` on latency if the available `swap memory` was 2Gb?" to diagnose the faults and recommend changes to the configuration options to mitigate these faults. We evaluate CADET on 5 real-world highly configurable software systems including image recognition using Xception [56], natural language processing (NLP) using BERT [57], speech recognition using DeepSpeech [58], database management system using SQLite [59], and video encoder using x264 [60] deployed on 3 architecturally different NVIDIA Jetson devices such as TX1, TX2 and XAVIER [61, 62]. The key reason we evaluated these systems is that they expose numerous intrinsic non-linear interactions between options that are beyond the visibility of software developers who use them. We compare CADET with state-of-the-art ML-based performance debugging approaches including DELTADEBUGGING [63], CBI [41], BUGDOC [42], and ENCORE [43]. Overall, we find that CADET is (on average) $7\times$ faster in detecting the root cause of non-functional faults with 8% better accuracy and recommending fixes with 11% higher gain from the corresponding fault when compared to the next best ML-based approaches in some cases. We also present a real-world case study where we demonstrate that CADET finds repair with 14% higher gain than the experts' advice in 24 minutes. The dataset and the scripts to reproduce our findings are available in https://github.com/softsys4ai/CADET.

2 Motivation — A Real-World Example

We present an instance of a non-functional fault that was reported in the NVIDIA developer forum [2]. Here, a developer notices some strange behavior when trying to transplant their code for real-time computation of depth information from stereo-cameras for object detection from NVIDIA Jetson TX1 to TX2. Since TX2 has twice the compute power as TX1, the developer expects to have at least 30% – 40% lower latency in TX2. However, the developer observed that TX2 had $4\times$ the latency as TX1. To solve this problem, the developer solicits advice from the developer forums. After discussions spanning two days, the developer learns that she/he has made several misconfigurations:

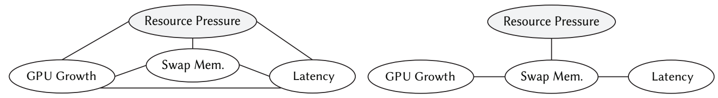
- *Wrong compilation flags*: The compilation does not take into account the microarchitectural differences between the two platforms that may be fixed by setting the `-gencode=arch` parameter and compiling the code dynamically by disabling the `CUDA_USE_STATIC` flag. TX1 is based on Maxwell microarchitecture, while TX2 is based on Pascal microarchitecture. These two microarchitectures have significant differences in terms of power usage and compute speed [64].

- *Wrong CPU/GPU clock frequency*: The hardware configuration is set incorrectly. These may be fixed by setting the configuration `-nvpmodel=MAX-N` which changes the CPU and GPU clock settings. The `Max-N` setting in TX2 provides almost twice the performance of TX1 [65] due to increased clock speeds and TX2's use of 128-bit memory bus width versus the 64-bit in TX1 [64].

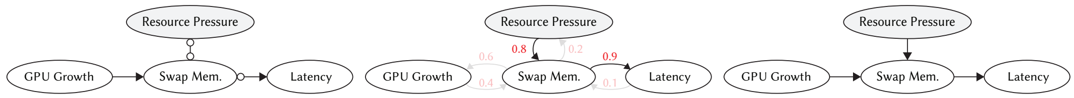
[2] https://forums.developer.nvidia.com/t/50477

	(Configurable Options)		(Sys. Events)	Latency
	GPU Growth	Swap Mem	Resource Use	*etc.*
c_1	0.25	2 *Gb*	10%	1 sec, …
c_2	0.5	1 *Gb*	20%	2 sec, …
⋮	⋮	⋮ ⋱	⋮	⋮
c_n	1.0	4 *Gb*	40%	0.1 sec, …

(a) Sample Observational Data



(b) Dense Graph (c) FCI Skeleton



(d) Partial Ancestral Graph (PAG) from FCI (e) DAG from NOTEARS (f) Final ADMG

Figure 3: From observational data to fully connected, skeleton graph, and partial ancestral graph (PAG).

- *Wrong fan modes*: The fan modes need to be configured correctly to account for higher CPU/GPU clock speeds. Otherwise, TX2 will thermal throttle the CPU and GPU to prevent overheating [66] and invariably increasing the latency [67].

This is only one of many examples in a single system, in which misconfigurations severely impact the non-functional properties of the system. Examples abound from many other systems and domains, including IoT (e.g. Amazon Alexa) [68, 69] and production-scale cloud-based systems [11].

3 CADET: Causal Debugging Toolkit

This section presents a brief description of CADET (outlined in Fig. 2). We gather a few dozen samples of observational data, by measuring the non-functional properties of the system (e.g., latency, etc) under different configuration settings (see ① in Fig. 2) to construct a graphical causal model using the observational data (see ② in Fig. 2). Then, we find paths that lead from configuration options to latency, energy consumption, and thermal output (see ③ in Fig. 2). Next, a query engine generates several counterfactual queries (what-if questions) about specific changes to each configuration option (see ④ in Fig. 2) and finds which of these queries has the highest causal effect on remedying the non-functional fault(s). Finally, we generate and evaluate the new configuration to assert if the newly generated configuration mitigates the non-functional fault(s). If not, we repeat the process by adding this to the current observational data.

Causal structure discovery. In this stage, we express the relationships between configuration options (e.g., CPU freq, *etc*.) and the non-functional properties (e.g., latency, *etc*) using a causal model. A causal model is a *acyclic directed mixed graph* (hereafter, ADMG) [70, 71]. The nodes of the ADMG have the configuration options and the non-functional properties (e.g., latency, *etc*). Additionally, we enrich the causal graph by including several nodes that represent the status of internal system events, e.g., resource pressure (as in Fig. 1). Unlike configuration options, these system events cannot be modified. However, they can be observed and measured to understand how the causal-effect of changing configurations propagates to latency, energy consumption, or heat dissipation, e.g., resource pressure in Fig. 1 determines how `GPU growth` affects latency. To build the causal model we gather two dozen samples of observational data (resembling Table 3a). To convert observational data into a causal graph, we use a prominent structure discovery algorithm called Fast Causal Inference (hereafter, FCI) [53]. We picked FCI because it accommodates for the existence of unobserved confounders [53, 72, 73]. This is important because we do not assume absolute knowledge about the configuration space, hence there could be certain configurations we could not modify or system events we have not observed. First, we build a dense graph by connecting all pairs of variables with an undirected edge (as seen in Fig. 3b). Next, we use Fisher's exact test [74] to evaluate the independence of all pairs of variables conditioned on all remaining variables. Pruning edges between the independent variables results in a skeleton graph as shown in Fig. 3c. Next, we orient undirected edges using edge orientation rules [53, 72, 73, 75] to produce a partial ancestral graph (as in Fig. 3d). We compare all the partially directed edges from the FCI's PAG (Fig. 3d) with their corresponding counterparts from NOTEARS' DAG (Fig. 3e). The final causal model would be an ADMG that resembles Fig. 3f.

Causal path extraction. In this stage, we extract paths from the causal graph (referred to as *causal paths*) and rank them based on their average causal effect on latency, energy consumption, and heat dissipation (our three non-functional properties). A causal path is a directed path originating from either the configuration options or the system events and terminating at a non-functional property (i.e., latency, energy consumption, or heat dissipation). To discover causal paths, we backtrack from the nodes corresponding to each non-functional property until we reach a node with no parents. For example, from Fig. 3f, we can extract two paths: (1) `GPU growth` $\longrightarrow$ `swap memory` $\longrightarrow$ Latency, and (2) Resource Pressure $\longrightarrow$ `swap memory` $\longrightarrow$ Latency.

A complex causal graph can result in a large number of causal paths. Therefore, we rank the paths in descending order from ones having the highest causal effect to ones having the lowest 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 `GPU growth` or X) on its successor in the path (say `swap memory` or Z). We express this with the *do-calculus* notation such as $\mathbb{E}[Z \mid \text{do}(X = x)]$ that represents the expected value of Z (`swap memory`) if we set the value of the node X (`GPU growth`) to x. To compute the *average causal effect* (ACE) of $X \rightarrow Z$ (i.e., `GPU growth` $\longrightarrow$ `swap memory`), we find the average over all permissible values the node X (`GPU growth`) can take, i.e., $\text{ACE}(Z, X) = \frac{1}{N} \cdot \sum_{\forall a, b \in X} \mathbb{E}[Z \mid \text{do}(X = b)] - \mathbb{E}[Z \mid \text{do}(X = a)]$. Here, N represents the total number of values X (`GPU growth`) can take. If changes in `GPU growth` result in a large change in `swap memory`, then the $\text{ACE}(Z, X)$ will be larger, indicating that `GPU growth` on average has a large causal effect on `swap memory`. The prior equation can be extended to the compute causal effect of a path P_{ACE}.

Repairing non-functional faults. In this stage, we use the top K paths with the largest P_{ACE} values to: (a) identify the root cause of non-functional faults; and (b) prescribe ways to fix the non-functional faults. When experiencing non-functional faults, a developer may ask specific queries to CADET and expect an actionable response. For this, we translate the developer's queries into formal probabilistic expressions that can be answered using the causal paths. We use counterfactual reasoning to generate these probabilistic expressions. To understand query translation, we use the example causal graph of Fig. 3f where a developer observes a latency fault and has the following questions:

❓ **"What is the root cause of my latency 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 above to extract the paths from the causal graph and rank the paths based on their average causal effect (i.e., P_{ACE} from) on latency. For example, in Fig. 3f we may return the path (say) `GPU growth` $\longrightarrow$ `swap memory` $\longrightarrow$ Latency and the configuration options `GPU growth` *and* `swap memory` both being probable root causes.

❓ **"How to improve my latency?"** To answer this query, we first find the root cause as described above. Next, we discover what values each of the configuration options must take so that the new latency is better (low latency) than the fault (high latency). For example, we consider the causal path `GPU growth` $\longrightarrow$ `swap memory` $\longrightarrow$ Latency, we identify the permitted values for the configuration options `GPU growth` and `swap memory` that can result in a low latency (Y^{LOW}) that is better than the fault (Y^{HIGH}). For this, we formulate a counterfactual expression of the form $\Pr(Y^{\text{LOW}}_{repair} | \neg repair, Y^{\text{HIGH}}_{\neg repair})$ that 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 `GPU growth` $=0.66$ or `GPU growth` $=0.66$, `swap memory` $=4Gb$, *etc*. We generate a *repair set* ($\mathcal{R}$) which contains the set of changes where the configurations `GPU growth` and `swap memory` are set to all permissible values, Next, we compute the *individual treatment effect* (ITE) on the latency (Y) for each repair in the repair set $\mathcal{R}$. In our case, for each repair $r \in \mathcal{R}$, ITE is $\text{ITE}(r) = \Pr(Y^{\text{LOW}}_{r} \mid \neg r,\ Y^{\text{HIGH}}_{\neg r}) - \Pr(Y^{\text{HIGH}}_{r} \mid \neg r,\ Y^{\text{LOW}}_{\neg r})$. ITE measures the difference between the probability that the latency *is low* after a repair r and the probability that the latency is *still high* after a repair r. To find the most useful repair ($\mathcal{R}_{best}$), we find the 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 latency fault.

Incremental learning. In this stage, we generate a new configuration using the recommended repairs from the $\mathcal{R}_{best}$ value. We reconfigured the system with the new configuration and we observe the system behavior. If the new configuration resolves the non-functional fault, we return the recommended repairs to the developer. Since the causal model uses limited observational data, there may be a discrepancy between the actual performance of the system after the repair and the value of the estimation from $\mathcal{R}_{best}$ derived from the current version of the causal graph. The more accurate the causal graph, the more accurate the recommended configuration will be [53, 72, 73, 75, 76]. 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 the causal effects will become more accurate. We terminate the incremental learning once we achieve the desired performance.

4 Case Study: Latency Fault in TX2

This section revisits the real-world latency fault previously discussed in §2. For this study, we reproduce the developers' setup to assess how effectively CADET can diagnose the root-cause of the misconfigurations and fix them. For comparison, we use BUGDOC (an ML-based diagnosis tool) and the recommendations by the domain experts on the forum. Fig. 4 illustrates our findings. We find that CADET could diagnose the root cause of the misconfiguration and recommends a fix within 24 minutes. Using the recommended configuration fixes from CADET, we achieved a frame rate of 26 FPS (53% better than TX1 and 6.5× better than the fault). This exceeds the developers' initial expectation of $30-40\%$ (or $22-24$ FPS). BUGDOC performed worse than CADET (21% improvement over TX1) while taking 3.5 hours (time mostly spent on collecting training samples to train internal decision tree) and changed several unrelated configurations (depicted by ✔) not endorsed by the domain experts.

Why CADET works better (and faster)?

CADET discovers the misconfigurations by constructing a causal model (a simplified version of this is shown in Fig. 4). 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, CADET can reason about changes to configurations with the highest average causal effect (last column in Fig. 4). The counterfactual reasoning occurs with no additional measurements, significantly speeding up inference.

Together, the causal model and the counterfactual reasoning enable CADET to pinpoint the configuration options that were misconfigured and recommend a fix for them promptly. As shown in Fig. 4, CADET accurately finds all the configuration options recommended by the forum (depicted by ✔ in Fig. 4). Further, CADET recommends fixes to these options that result in 14% better latency than the recommendation by domain experts in the forum. More importantly, CADET takes only 24 minutes (vs. 2 days of forum discussion) without modifying unrelated configurations.

Problem [77]: 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	CADET	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)	**26**	20	23	
Latency Gain (over TX1)	**53%**	21%	39%	
Latency Gain (over default)	**6.5×**	5×	5.75×	
Resolution time	**24 mins**	3.5 hrs	2 days	

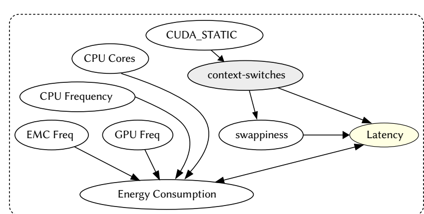


Figure 4: Using CADET on the real-world example from §2. CADET is better and faster than other methods.

5 Evaluation

Experimental Setup. This study uses three NVIDIA Jetson platforms: TX1, TX2, and XAVIER and five software systems on each platform: (1) An image recognition system with Xception to classify 5000 images from the CIFAR10 dataset [78]; (2) an NLP system with BERT to perform sentiment analysis on 10000 reviews from the IMDb dataset [79]; (3) An RNN based voice recognition system with DeepSpeech on 5 seconds long audio files; (4) SQLite, a database management system, to perform read, write, and insert operations; and (5) x264 video encoder to encode a video file of size 11MB with a resolution of 1920 x 1080. We use 28 configuration options that include 10 software options, 8 OS/Kernel options, and 10 hardware options. We curate a non-functional faults dataset, called the JETSON FAULTS dataset, and ground truth for each observed non-functional faults for each of the software and hardware system used in our study. We create a *ground-truth data* by measuring configurations for a fixed budget of 24 hours and identifying their root-causes manually for each fault by selecting the configuration with the highest performance gain from the fault. By definition, non-functional faults have latency, energy consumption, and heat dissipation that take tail values [11, 80], i.e., they are worse than the 99^{th} percentile. We filter our data set to find the configurations that result in tail values for latency, energy consumption, and/or heat dissipation and label these configurations as '*faulty*'. We evaluate the predicted root-causes in terms of accuracy (Jaccard similarity). To compute accuracy, we compare the set of configuration options identified by CADET to be the root cause with the true root-cause obtained from the ground truth data. To assess the quality of fixes, we measure the percentage improvement (gain %) after applying the recommended repairs using Δ_{gain}. We prefer higher accuracy and gain.

Results. We compare CADET with four state-of-the-art ML-based methods for fault diagnostics, namely: DELTADEBUGGING [63], CBI [41], BUGDOC [42], and ENCORE [43]. For all methods, we set a maximum budget of 4 hours. All methods require some initial observational data to operate. Within the budget, CADET samples 25 initial observational data to incrementally generate, evaluate, and update the causal model with candidate repairs. Other methods require a large and diverse pool of observational data for training. However, collecting observational data is expensive and time-consuming. Therefore, we use the entire budget of 4 hours to generate random configuration samples to train ML-based methods. We assess the effectiveness of diagnostics for "single-objective" non-functional faults, i.e., faults that occur only in one of latency, energy consumption, or heat dissipation. For brevity, we evaluate latency faults in TX2, energy consumption faults in XAVIER, and heat dissipation faults in TX1. Our findings generalize over other hardware. Table 1 summarizes the effectiveness of CADET over other ML-based fault diagnosis approaches. We observe the following:

Table 1: Efficiency of CADET compared to other approaches. Cells highlighted in **blue** indicate maximum improvement over faults. CADET achieves better performance overall and is much faster.

			Accuracy					Gain					Time†	
			CADET	CBI	δ-DEBUG	ENCORE	BUGDOC	**CADET**	CBI	δ-DEBUG	ENCORE	BUGDOC	**CADET**	Others
TX2	Latency	Image	**84**	66	65	68	71	**81**	48	42	57	59	**0.6**	4
		NLP	**76**	65	60	66	66	**74**	54	59	62	58	**0.2**	4
		Speech	**75**	64	63	63	72	**77**	59	53	55	66	**0.7**	4
		x264	**76**	67	60	61	70	**23**	9	12	8	11	**1.2**	4
		SQLite	**84**	65	68	65	70	**19**	13	11	12	8	**0.5**	4
XAVIER	Energy	Image	**74**	63	55	63	64	**83**	59	50	35	51	**0.2**	4
		NLP	**77**	60	63	66	64	**63**	49	36	49	53	**0.4**	4
		Speech	**73**	66	65	61	71	**82**	64	48	65	63	**1.1**	4
		x264	**74**	62	57	59	67	**26**	13	11	16	16	**0.1**	4
		SQLite	**80**	53	62	66	71	**21**	16	10	14	15	**0.5**	4
TX1	Thermal	Image	**69**	63	57	64	65	**3**	3	2	2	2	**0.7**	4
		NLP	**71**	62	61	61	62	**5**	4	1	2	4	**0.4**	4
		Speech	**71**	61	64	62	67	**3**	4	2	2	2	**1.1**	4
		x264	**74**	65	57	64	65	**7**	3	2	2	3	**0.2**	4
		SQLite	**66**	64	54	64	65	**6**	2	2	2	3	**0.9**	4

- ***Accuracy and gain.*** CADET *significantly outperforms ML-based methods in all cases*. For example, in SQLite database management system on TX2, CADET achieves 14% more accuracy compared to BUGDOC (best among the remaining ML-based approaches). We observe similar trends in energy faults, i.e., CADET outperforms other methods in all cases. CADET *can recommend repairs for faults that significantly improves latency and energy usage*. Applying the changes to the configurations recommended by CADET increases the performance drastically. We observed latency gains as high as 81% (22% more than BUGDOC) on TX2 and energy gain of 83% (32% more than BUGDOC) on XAVIER for image recognition.

- ***Wallclock time.*** CADET *can resolve misconfiguration faults significantly faster than ML-based approaches*. In Table 1, the last two columns indicate the time taken (in hours) by each approach to diagnosing the root cause. We find that while other approaches use the entire budget of 4 hours to diagnose and resolve the faults, CADET can do so significantly faster before the maximum budget is exhausted, e.g., CADET is $40\times$ faster in diagnosing and resolving faults in energy usage for x264 deployed on XAVIER and $20\times$ faster in diagnosing latency faults for NLP task on TX2. ML-based methods require a large number of initial observational data for training. They spend most of their allocated 4-hour budget on gathering these training samples. In contrast, CADET starts with only 25 samples and uses incremental learning to judiciously update the casual graph with new configurations until a repair has been found.

Discussion. Table 1 shows that image recognition, NLP and speech recognition deep neural network (DNN) systems had the most improvements with CADET compared to x264 and SQLite. Misconfigurations affecting the onboard GPU lead to severe degradation in latency and energy usage. Since DNN relies on GPU to optimize the operations, it must be configured appropriately to leverage the full hardware potential. Other applications were less sensitive to such misconfigurations. Further, all methods found it difficult to discover and resolve thermal faults. While CADET outperformed other methods, the overall accuracy, and *gain* were lower than those for latency and energy consumption faults. We believe there are two reasons for this: (1) The workloads exercised in this work did not significantly heat the system; and (2) the thermal measurements were taken in a controlled environment (indoor in a stable temperature), as a result, the variance temperature was relatively lower.

6 Conclusion

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

Acknowledgments. We like to acknowledge Christian Kästner, Sven Apel, Tianyin Xu, Vivek Nair, Jianhai Su, Miguel Velez, Mohsen Amini Salehi and Tobius Dürschmid for their valuable feedback and suggestions in improving the paper. This work was partially supported by NASA (RASPERRY-SI Grant Number 80NSSC20K1720) and NSF (SmartSight Award 2007202).

References

[1] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 307–319, 2015.

[2] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 643–654. IEEE, 2016.

[3] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. *Empirical Software Engineering*, 24(2):674–717, 2019.

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

[5] Sokratis Tsakiltsidis, Andriy Miranskyy, and Elie Mazzawi. On automatic detection of performance bugs. In *2016 IEEE international symposium on software reliability engineering workshops (ISSREW)*, pages 132–139. IEEE, 2016.

[6] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*, pages 237–246. IEEE, 2013.

[7] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

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

[9] Ana B Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. Tandem: A taxonomy and a dataset of real-world performance bugs. *IEEE Access*, 8:107214–107228, 2020.

[10] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 902–912. IEEE, 2015.

[11] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[12] 99% of misconfiguration incidents in the cloud go unnoticed. In Help Net Security: https://www.helpnetsecurity.com/2019/09/25/cloud-misconfiguration-incidents/, September 2019.

[13] Jonathan Greig. Cloud misconfigurations cost companies nearly $5 trillion. In TechRepublic: https://www.techrepublic.com/article/cloud-misconfigurations-cost-companies-nearly-5-trillion/, February 2020.

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

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

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

[17] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering*, 25(2):247–277, 2018.

[18] Pooyan Jamshidi and Giuliano Casale. 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)*. IEEE, 2016.

[19] Md Shahriar Iqbal, Jianhai Su, Lars Kotthoff, and Pooyan Jamshidi. Flexibo: Cost-aware multi-objective optimization of deep neural networks. *arXiv preprint arXiv:2001.06588*, 2020.

[20] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280. IEEE, 2012.

[21] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 301–311. IEEE, 2013.

[22] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 284–294, 2015.

[23] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 167–177. IEEE, 2012.

[24] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, pages 284–294. ACM, August 2015.

[25] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding product line configurations with high performance by random sampling. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*. ACM, 2017.

[26] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. Transferring performance prediction models across different hardware platforms. In *Proc. Int'l Conf. on Performance Engineering (ICPE)*, pages 39–50. ACM, 2017.

[27] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 301–311. IEEE, 2013.

[28] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 39–50, 2017.

[29] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *Proc. Int'l Symp. Soft. Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017.

[30] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, 2017.

[31] Md Shahriar Iqbal, Lars Kotthoff, and Pooyan Jamshidi. Transfer Learning for Performance Modeling of Deep Neural Network Systems. In *USENIX Conference on Operational Machine Learning*, Santa Clara, CA, 2019. USENIX Association.

[32] Fei Wu, Pranay Anchuri, and Zhenhui Li. Structural event detection from log messages. In *23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1175–1184, 2017.

[33] Tong Jia, Pengfei Chen, Lin Yang, Ying Li, Fanjing Meng, and Jingmin Xu. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *ICWS 2017*, pages 25–32.

[34] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. Data x-ray: A diagnostic tool for data errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1231–1245, 2015.

[35] Xiao Yu. Understanding and debugging complex software systems: A data-driven perspective. 2018.

[36] Charles M Curtsinger. Effective performance analysis and debugging. 2016.

[37] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. {REPT}: Reverse debugging of failures in deployed software. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 17–32, 2018.

[38] Mejbah Alam, Justin Gottschlich, Nesime Tatbul, Javier S Turek, Tim Mattson, and Abdullah Muzahid. A zero-positive learning approach for diagnosing software performance regressions. In *Advances in Neural Information Processing Systems*, pages 11627–11639, 2019.

[39] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. Interpretable and informative explanations of outcomes. *Proceedings of the VLDB Endowment*, 8(1):61–72, 2014.

[40] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. Qfix: Diagnosing errors through query histories. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1369–1384, 2017.

[41] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. *ACM SIGPLAN Notices*, 49(10):561–578, 2014.

[42] Raoni Lourenço, Juliana Freire, and Dennis Shasha. Bugdoc: A system for debugging computational pipelines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2733–2736, 2020.

[43] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 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*, pages 687–700, 2014.

[44] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 435–448, 2020.

[45] Xue Han, Tingting Yu, and David Lo. Perflearner: learning from bug reports to understand and generate performance test frames. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 17–28. IEEE, 2018.

[46] Rahul Krishna, Tim Menzies, and Lucas Layman. Less is more: Minimizing code reorganization using xtree. *Information and Software Technology*, 88:53–66, 2017.

[47] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571. IEEE, 2013.

[48] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[49] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. Causal testing: Understanding defects' root causes. In *Proceedings of the 2020 International Conference on Software Engineering*, 2020.

[50] Anna Fariha, Suman Nath, and Alexandra Meliou. Causality-guided adaptive interventional debugging. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 431–446, 2020.

[51] Judea Pearl et al. Models, reasoning and inference. *Cambridge, UK: CambridgeUniversityPress*, 2000.

[52] Judea Pearl. *Causality*. Cambridge university press, 2009.

[53] Peter Spirtes, Clark N Glymour, Richard Scheines, and David Heckerman. *Causation, prediction, and search*. MIT press, 2000.

[54] Xun Zheng, Bryon Aragam, Pradeep K Ravikumar, and Eric P Xing. Dags with no tears: Continuous optimization for structure learning. In *Advances in Neural Information Processing Systems*, pages 9472–9483, 2018.

[55] Judea Pearl. The algorithmization of counterfactuals. *Annals of Mathematics and Artificial Intelligence*, 61(1):29, 2011.

[56] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[57] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[58] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[59] Richard D Hipp. SQLite, https://www.sqlite.org/index.html, 2020.

[60] x264, http://www.videolan.org/developers/x264.html.

[61] Hassan Halawa, Hazem A. Abdelhafez, Andrew Boktor, and Matei Ripeanu. NVIDIA jetson platform characterization. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 10417 LNCS:92–105, 2017.

[62] Sparsh Mittal. A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *J. Syst. Archit.*, 97(January):428–442, 2019.

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

[64] Dustin Franklin. Nvidia jetson TX2 delivers twice the intelligence to the edge. In NVIDIA Developer Blog: https://developer.nvidia.com/blog/jetson-tx2-delivers-twice-intelligence-edge/, March 7, 2017.

[65] Nvidia jetson TX2 delivers twice the intelligence to the edge. In JetsonHacks Blog: https://www.jetsonhacks.com/2017/03/25/nvpmodel-nvidia-jetson-tx2-development-kit/, March 25, 2017.

[66] Question about thermal management. In NVIDIA developer forums: https://forums.developer.nvidia.com/t/59855, April 2018.

[67] Chia-Chang Chiu and Yi-Sheng Chueh. Method and computer system for thermal throttling protection, October 30 2012. US Patent 8,301,873.

[68] Dikla Barda, Roman Zaikin, and Yaara Shriki. Keeping the gate locked on your IoT devices: Vulnerabilities found on amazon's alexa. By Check Point Research: https://research.checkpoint.com/2020/amazons-alexa-hacked/, August 2020.

[69] Andrie Ene. Alexa hack jeopardized echo users network. http://bit.ly/AlexaHack2020, August 2020.

[70] Thomas Richardson, Peter Spirtes, et al. Ancestral graph markov models. *The Annals of Statistics*, 30(4):962–1030, 2002.

[71] Robin J Evans and Thomas S Richardson. Markovian acyclic directed mixed graphs for discrete data. *The Annals of Statistics*, pages 1452–1482, 2014.

[72] Juan Miguel Ogarrio, Peter Spirtes, and Joe Ramsey. A hybrid causal search algorithm for latent variable models. In *Conference on Probabilistic Graphical Models*, pages 368–379, 2016.

[73] Clark Glymour, Kun Zhang, and Peter Spirtes. Review of causal discovery methods based on graphical models. *Frontiers in genetics*, 10:524, 2019.

[74] Lynne M Connelly. Fisher's exact test. *Medsurg Nursing*, 25(1):58–60, 2016.

[75] Diego Colombo, Marloes H Maathuis, Markus Kalisch, and Thomas S Richardson. Learning high-dimensional directed acyclic graphs with latent and selection variables. *The Annals of Statistics*, pages 294–321, 2012.

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

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

[78] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 (Canadian Institute for Advanced Research), http://www.cs.toronto.edu/~kriz/cifar.html.

[79] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

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