Supplementary Materials for UNICORN

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

Rahul Krishna
IBM Research
rkrsn@ibm.com

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

A Appendix

A.1 Causal Performance Modeling and Analyses: Motivating Scenarios (Additional details)

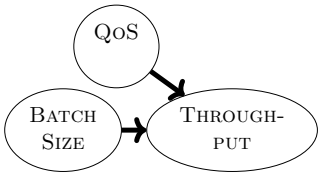

Figure 1. Performance influence model incorrectly identifies `Batch Size` and `QoS` are positively correlated with the term `0.08 Batch Size×QoS` whereas they are unconditionally independent. Causal model correctly identifies the dependence (no causal connection) relationship between `Batch Size` and `QoS` (no arrow between `Batch Size` and `QoS`).

Figure 2. Causal model correctly identifies how `CPU Frequency` causally influences `Throughput` via `Cycles` whereas the performance influence model `Throughput` $= 0.05 \times$ `CPU Frequency` $\times$ `Cycles` identified incorrect interactions.

Fig. 1 and Fig. 2 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.

The performance behavior of regression models for configurable systems varies when sample size varies. Fig. 3 shows

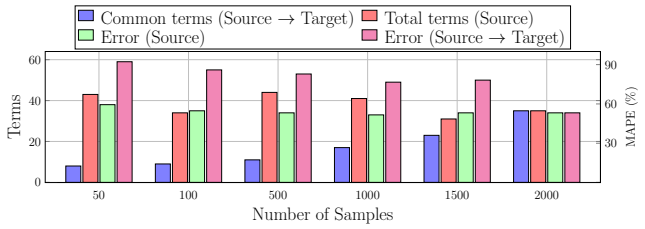

Figure 3. Performance influence models relying on correlational statistics are not stable as new samples are added and do not generalize well. Common terms refers to the individual predictors (i.e., options and interactions) in the performance models that are similar across envirnments.

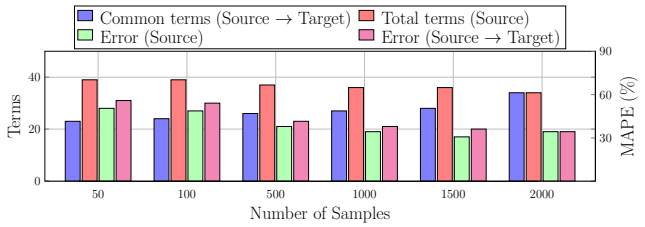

Figure 4. Causal performance models are relatively more stable as new samples are added and do generalize well.

the change of a number of stable terms and error with different numbers of samples used for building a performance influence model. Here, we vary the number of samples from 50 to 1500 to build a source regression model. We use a sample size of 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 these 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 leads to better generalization, as shown in Fig. 4. 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.

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519575

Table 1. Mapping between configuration options and options indexes. Only a subset of configuration options are shown here.

Option Index	Configuration Options	Option Index	Configuration Options
0	Swap Memory	14	kernel.numa_balancing
1	Scheduler Policy	15	kernel.sched_latency_ns
2	Drop Caches	16	kernel.sched_nr_migrate
3	Batch Size	17	kernel.sched_rt_period_us
4	Bitrate	18	kernel.sched_rt_runtime_us
5	Buffer Size	19	kernel.sched_time_avg_ms
6	CPU Freqeuncy	20	kernel.sched_child_runs_first
7	GPU Frequency	21	vm.vfs_cache_pressure
8	EMC Frequency	22	vm.swappiness
9	CPU Cores	23	Enable Padding
10	vm.overcommit_memory	24	vm.dirty_background_ratio
11	vm.overcommit_hugepages	25	vm.dirty_background_bytes
12	kernel.cpu_time_max_percent	26	vm.dirty_ratio
13	kernel.max_pids	27	vm.nr_hugepages

Table 2. Configuration options in Xception, Bert, and Deepspeech.

Configuration Options	Option Values/Range
Memory Growth	-1, 0.5, 0.9
Logical Devices	0, 1

Extraction of predictor terms from the causal performance model. The constructed causal performance models 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 is more than one path through a system event from performance objective to configuration options, we consider all possible interactions between those configuration options to calculate the number of causal terms.

A.2 Unicorn (Additional details)

Note, if X is a continuous variable, we would replace the summation of ACE with an integral. For the entire path, we extend it as:

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

Eq. (1) 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 to 25, 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

Table 3. x264 software configuration options.

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

Table 4. SQLite software configuration options.

Configuration Options	Option Values/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
PRAGMA MMAP_SIZE	30000000000, 60000000000,

Table 5. Linux OS/Kernel configuration options.

Configuration Options	Option Values/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_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 6. Hardware configuration options.

Configuration Options	Option Values/Range
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)

Table 7. Performance system events and tracepoints.

System 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
Emulation Faults
Tracepoint Subsystems
Block
Scheduler
IRQ
ext4

developer may ask specific queries to UNICORN and expect an actionable response. 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:

❷ "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 **??** to extract the paths from the causal graph and rank the paths based on their average causal effect (i.e., Path_{ACE} from Eq. (1)) on latency and energy. We return the configurations that lie on the top K paths. For example, in **??** we may return (say) the following paths:

- Batch Size ⟶ Cache Misses ⟶ FPS and Energy
- Enable Padding ⟶ Cache Misses ⟶ FPS and Energy

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

❷ "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:

$$\Pr(Y^{\text{LOW}}_{repair} | \neg repair, Y^{\text{HIGH}}_{\neg repair}) \tag{2}$$

Eq. (2) 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.,

$$\mathcal{R}_1 \equiv \bigcup \{\texttt{Batch Size} = x, ...\}\, \forall x \in \texttt{Batch Size} \tag{3}$$

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 BufferSize and Enable padding=0. Similarly, we generate a repair set $\mathcal{R}_2$ by setting Enable Paddingto all permissible values.

$$\mathcal{R}_2 \equiv \bigcup \{\texttt{Enable padding} = x, ...\}\, \forall x \in \texttt{Enable padding} \tag{4}$$

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 Causal Effect* (ICE) on the FPS and Energy (Y) for each repair in the repair set $\mathcal{R}$. In our case, for each repair $r \in \mathcal{R}$, ICE is given by:

$$\text{ICE}(r) = \Pr(Y^{\text{LOW}}_r \mid \neg r,\ Y^{\text{HIGH}}_{\neg r}) - \Pr(Y^{\text{HIGH}}_r \mid \neg r,\ Y^{\text{HIGH}}_{\neg r}) \tag{5}$$

ICE measures the difference between the probability that FPS 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) ICE, i.e., $\mathcal{R}_{best} = \text{argmax}_{\forall r \in \mathcal{R}}[\text{ICE}(r)]$. This provides the developer with a possible repair for the configuration options that can fix the multi-objective FPS and Energy fault.

Remarks. The ICE computation of Eq. (5) 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 measure their performance in the real world. This offers significant runtime benefits.

A.3 Evaluation (Additional details)

A.3.1 Experimental setup We used the following four components for Deepstream implementation:

Table 8. Deepstream software configuration options.

Component	Configuration Options	Option Values/Range
Decoder	CRF	13, 18, 24, 30
	Bitrate	1000, 2000, 2800, 5000
	Buffer Size	6000, 8000, 20000
	Presets	ultrafast, veryfast, faster medium, slower
	Maximum Rate	600k, 1000k
	Refresh	OFF, ON
Stream Mux	Batch Size	0 - 30
	Batched Push Timeout	0 - 20
	Num Surfaces per Frame	1, 2, 3, 4
	Enable Padding	0, 1
	Buffer Pool Size	1 - 26
	Sync Inputs	0, 1
	Nvbuf Memory Type	0, 1, 2, 3
Nvinfer	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
Nvtracker	IOU Threshold	0 - 60
	Enable Batch Process	0, 1
	Enable Past Frame	0, 1
	Compute HW	0, 1, 2, 3, 4

- **Decoder**: For the decoder, we use x264. It uses the x264 and takes the encoded H.64, VP8, VP9 streams, and produces an NV12 stream.
- **Stream Mux**: The streammux module takes the NV12 stream and outputs the NV12 batched buffer with information about input frames, including the 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 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.

Configuration options, events, and hyperparameters used for evaluation. Table 8, Table 2, Table 3, and Table 4, show different software configuration options and their values for different systems considered in this paper. Table 5 shows the OS/kernel level configuration options and their values for different systems considered in this paper. Additionally, Table 7 shows the performance events considered in this paper. The hyperparameters considered for Xception, Bert, and Deepspeech are shown in Table 9.

Table 9. Hyperparameters for DNNs used in Unicorn.

Architecture	Hyperparameters	Option Values
Xception	Number of Filters Entry flow	32
	Filter Size Entry Flow	(3×3)
	Number of Filters Middle Flow	64
	Filter Size Middle Flow	(3×3)
	Number of Filters Exit Flow	728
	Filter Size Exit Flow	(3×3)
	Batch Size	32
	Number of Epochs	100
Bert	Dropout	0.3
	Maximum Batch Size	16
	Maximum Sequence Length	13
	Learning Rate	$1e^{-4}$
	Weight Decay	0.3
Deepspeech	Dropout	0.3
	Maximum Batch Size	16
	Maximum Sequence Length	32
	Learning Rate	$1e^{-4}$
	Number of Epochs	10

Table 10. Hyperparameters for FCI used in Unicorn.

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

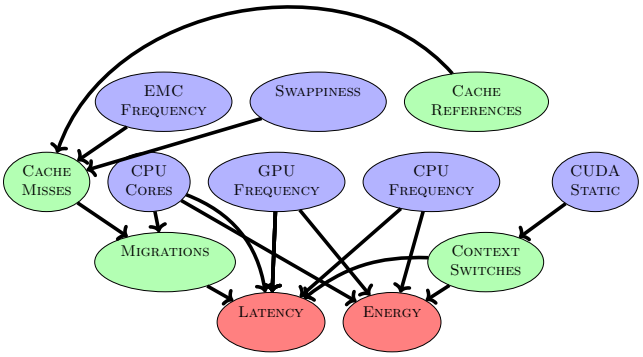

Figure 5. Causal graph used to resolve the latency fault in the real world case study in section ??

A.3.2 Case Study. Fig. 5 shows the causal graph to resolve the real-world latency fault.

Table 11. Efficiency of UNICORN compared to other approaches. Cells highlighted in **blue** indicate improvement over faults and red indicate deterioration. UNICORN achieves better performance overall and is much faster.

(a) Single objective performance fault for *heat* in TX1.

		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
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 for *Heat, Latency* in TX2.

		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**	54	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 for *Energy, Heat* in XAVIER.

		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**	55	57	63	**64**	55	51	62	**67**	47	53	60	**58**	44	51	54	**3**	1	1	1	**0.8**	4
	BERT	**69**	55	51	59	**65**	53	47	61	**71**	53	61	67	**65**	41	51	61	**4**	2	2	3	**0.4**	4
	DEEPSPEECH	**72**	55	49	61	**73**	51	51	61	**71**	57	53	64	**69**	47	51	64	**4**	1	1	3	**0.3**	4
	x264	**72**	59	57	66	**71**	51	55	62	**69**	61	59	59	**67**	51	51	61	**5**	2	3	4	**0.5**	4

(d) Multi-objective non-functional faults for *Energy, Heat, and Latency* in 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

A.3.3 Effectiveness. Table 11(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.

UNICORN *can resolve misconfiguration faults significantly faster than correlation-based approaches*. In Table 11, 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.

Table 12. Transferring causal models across hardware platforms. Cells highlighted in **blue** indicate the transferability potential of Unicorn when compared to Unicorn (Rerun).

		TX1 (source) ⟶ TX2 (target)											
		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)
Latency	Xception	52	83	86	70	79	86	46	78	83	46	71	82
	Bert	55	75	81	57	70	71	45	67	76	43	70	74
	Deepspeech	45	71	81	56	79	81	49	73	76	54	73	76
	x264	57	79	83	70	75	78	58	77	82	45	73	85
		TX2 (source) ⟶ Xavier (target)											
Energy	Xception	53	74	84	48	73	80	51	69	78	43	73	83
	Bert	50	61	66	53	71	79	49	66	70	40	55	62
	Deepspeech	57	70	73	45	74	78	43	69	75	49	71	78
	x264	54	72	77	46	72	78	42	75	83	46	79	87
		Xavier (source) ⟶ TX1 (target)											
Heat	Xception	63	64	69	61	67	68	58	74	75	3	4	4
	Bert	55	65	71	59	67	72	52	64	72	3	4	5
	Deepspeech	57	64	71	59	63	69	53	63	71	1	2	3
	x264	51	65	74	53	64	74	54	69	74	3	5	7

A.3.4 Transferability. Table 12 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 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 the 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 of Unicorn (Rerun) for heat faults. For latency and energy faults, the main 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 causal performance model 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.

A.3.5 Scalability. The 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 task is the same, it is sufficient to examine the scalability of Unicorn for the debugging non-functional fault task.

Table 13. Scalability for SQLite and DeepStream on Xavier.

							Time/Fault (in sec.)		
System	Configs	Events	Paths	Queries	Degree	Gain (%)	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**

SQLite was chosen because it offers a large number of configurable options, much more than neural 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 many as 242 configuration options and 19 events (Table 13, 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 13, 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 13 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 13 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 increases.

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.