Unicorn: Reasoning about Configurable System Performance through the Lens of Causality

Reproduktion und Evaluation

Alexander Vödisch
av21vupu@studserv.uni-leipzig.de

Abstract

In dieser Ausarbeitung werden die Ergebnisse von [ref] repliziert und evaluiert.

1 Einführung

Moderne Computersysteme sind oft aus einer Vielzahl von konfigurierbaren Komponenten aufgebaut, deren Zusammensetzung kritisch für die Systemperformance ist.[1] Da die optimale Konfiguration dieser Komponenten hardwareabhängig sein kann und insbesondere in Kombination zu Pipelines der Konfigurationsraum exponentiell wachsen kann, sind spezielle Algorithmen notwendig, um Nutzer solcher Systeme bei der Definition möglichst optimaler Konfigurationen zu unterstützen.

Das Ziel von UNICORN ist es, für konfigurierbare Systeme die optimalen Konfigurationen bezüglich eines vom Nutzer definierten Merkmals (z. B. Leistung, Energieverbrauch, Inferenzzeit) zu finden und im Zuge dessen einen kausalen Graphen zu generieren, der als Modell auch auf bisher unbekannte Harware übertragen werden kann.

Im Gegensatz zu Modellen, die beispielsweise auf Regression basieren und lediglich den Einfluss individueller Konfigurationsoptionen messen, bietet UNICORN den Vorteil, dass nicht die Korrelation von Konfigurationsoptionen zur Vorhersage von guten Konfigurationen, sondern die kausalen Zusammenhänge der einzelnen Optionen für die Entwicklung des Modells genutzt werden. Dadurch kann das Modell auch in unbekannten Environments genutzt werden, wo korrelationsbasierte Performance-Influence-Modelle oft nur unzuverlässige Vorhersagen treffen.

2 Funktionsweise

UNICORN ermöglicht sowohl das Debuggen als auch auch das Optimieren von Systemkonfigurationen. Hierzu definieren die Autoren ein kausales Modell das auf probabilistischen graphischen Modellen basiert. Diese kausalen Graphen bestehen aus

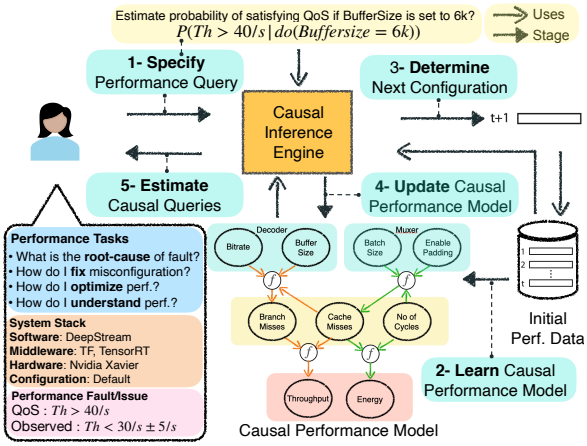

Abbildung 1: Arbeitsweise von Unicorn.

- Performance-Variablen als Knoten für mögliche Konfigurationsparameter (z. B. `Birate`, `Buffer Size` oder `Batch Size`),
- funktionalen Knoten, die funktionale Abhängigkeiten zwischen den Performance-Variablen modellieren,
- kausalen Verbindungen zwischen den Performance-Variablen und den funktionalen Knoten und
- Nebenbedingungen, um notwendige Einschränkungen bei der Modellbildung zu definieren (z. B. dürfen `Cache Misses` nur positive ganzzahlige Werte annehmen).

UNICORN arbeitet in 5 Schritten:

1. Spezifikation der Optimierungsanforderung als Klartext durch den Nutzer
2. Lernen des kausalen Modells mittels vordefinierter Anzahl an Musterkonfigurationen: Hierfür wird der *Fast Causal Inference-*

[1] Unter (System-)Performance werden alle Leistungsmerkmale eines Systems wie beispielsweise Durchsatz und Energieverbrauch zusammengefasst.

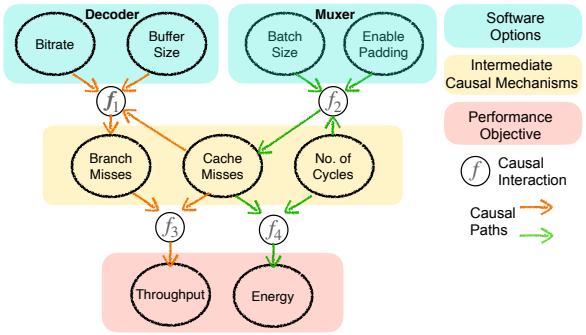

Abbildung 2: Grundlegende Struktur des kausalen Graphen am Beispiel von Deepsteam, einer hochkonfigurierbaren Video-Analyse-Pipeline. Die Performance-Variablen sindd über funktionale Knoten miteinander Verbunden. Die Ausrichtung der Pfeile gibt an, zwischen welchen Performance-Variablen ein kausaler Zusammenhang besteht.

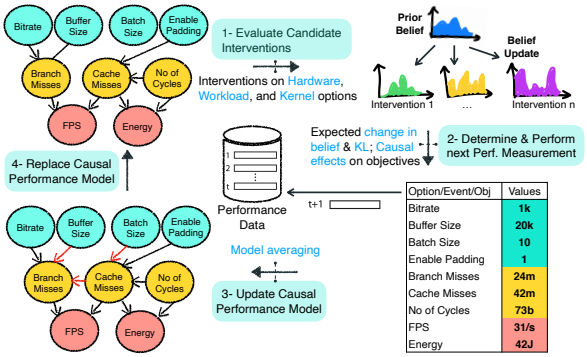

Abbildung 3: Update des kausalen Modells. Zunächst wird werden die Einflüsse der vorigen Konfigurationsänderungen abgeleitet. Anschließend wird die Konfiguration entsprechend den erwarteten Auswirkungen auf die vom Nutzer gestellten Performance-Objectives angeglichen. Daraufhin wird der kausale Graph angepasst und das Modell aktualisiert.

Algorithmus genutzt, um Kanten von dem zunächst vollständigem probabilistischen Graphen entsprechend der Nebenbedingungen zu entfernen und ungerichtete Kanten entsprechend der kausalen Einflüsse der Performance-Variablen auszurichten.

3. Bestimmung der Folgekonfiguration und Messen der Systemperformance: Für die Folgekonfiguration werden diejenigen Performance-Variablen angepasst, die kausal am stärksten Zusammenhängen, um den Lernprozess zu beschleunigen (*Active Learning*).
4. Inkrementelles Update des kausalen Modells
5. Wiederholung der Schritte 3 und 4 bis ein vordefiniertes Limit (z. B. Laufzeit) überschritten wurde und anschließend Ausgabe

3 Fallbeispiele

In dieser Ausarbeitung werden die Ergebnisse der Evaluation von UNICORN anhand der Fallbeispiele aus dem Originalpaper repliziert.

UNICORN kann im Online- oder Offline-Modus betrieben werden. Im Online-Modus werden die Performance-Metriken direkt auf der zugrundeliegenden Hardware ausgeführt. Der Offline-Modus erlaubt die Reproduktion auf beliebiger Hardware. Da uns die im Paper genutzte Hardware nicht zur Verfügung steht, werden lediglich die Ergebnisse der Offline-Evaluation repliziert.

Die Autoren stellen den Quellcode als Python-Bibliothek auf GitHub zur Verfügung.[2] Eine Dokumentation sowie eine ausführliche Anweisungen zur Replikation der Ergebnisse sind im Repository angehängt. Entsprechend den Anweisungen wurde Docker in Version `20.10.16` auf den Testsystemen installiert.

Bis auf ein fehlendes Paket, das zur Ausführung auf unseren Systemen notwendig war, konnten die Tests ohne Probleme ausgeführt werden. UNICORN erzeugt nach dem Testläufen mit `matplotlib` Graphen. In manchen Fällen konnten die Graphen nicht gespeichert werden, was jedoch nicht auf die eigentlichen Tests und deren Ergebnisse zurückzuführen ist.

Die Replikation findet auf zwei Testsystemen statt:[3]

- `Windows 10 21H2` mit `Intel Core i7-8700K CPU @ 12x3.50 GHz` und `32GB DDR4 RAM @ 3280Mhz`
- `MacOS Monterey 12.4` mit `Apple M1 CPU @ 4x3.20 GHz` und `16 GB LPDDR-DDR4X RAM @ 4266 Mhz`

Den Docker-Containern werden je 4 CPU-Kerne und 8 GB Arbeitsspeicher zur Verfügung gestellt.

Das System der Autoren im Offline-Mode verwendet als Prozessor `Intel Core i7-8700 CPU @ 12x3.20 GHz` und besitzt `31.2 GB` an Arbeitsspeicher, die zugrundeliegende Hardware ist zu unserem ersten System sehr ähnlich. Als Betriebssystem verwenden die Autoren jedoch `Ubuntu 18.04`.

[2] https://github.com/softsys4ai/unicorn

[3] Die Testsysteme werden im Folgenden kurz als `Win` und `Mac` bezeichnet.

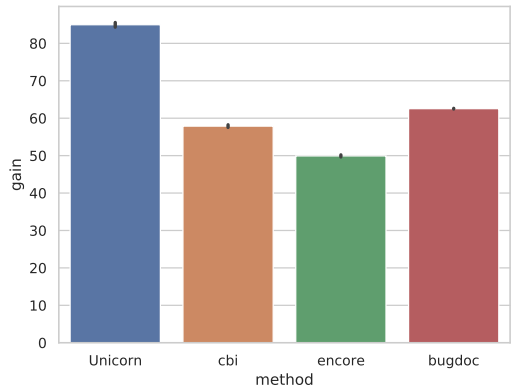

Abbildung 4: Verbesserung des Energieverbrauchs gegenüber Standardkonfiguration bei Einsatz von UNICORN, CBI, ENCORE und BUGDOC. Die Ergebnisse sind für `Win` und `Mac` identisch.

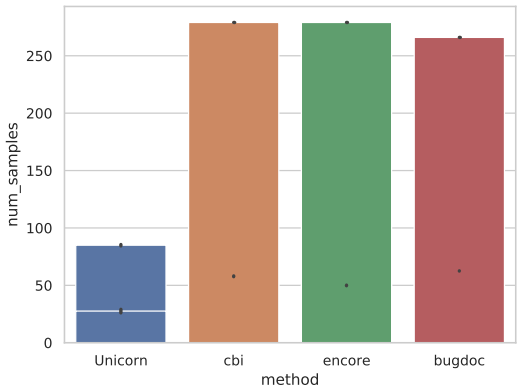

Abbildung 5: Notwendige Anzahl an Samples

Wir werden im Folgenden die Resultate der drei Kernbehauptungen des Originalpapers reproduzieren:

- UNICORN kann für das Aufsprüren der Ursachen von nicht-funktionalen Fehlern (Latenz, Energieverbrauch) genutzt werden.
- UNICORN kann als Werzeug bei der Durchführung von Performanceoptimierungen unterstützen.
- UNICORN ist effizient, auch wenn sich das Einsatzenvironment ändert.

4 Replikation der Ergebnisse

Für die Reproduktion der Ergebnisse werden die Anweisungen der Dokumentation in `artifact/REPRODUCE.md` genutzt. Leider konnten bei einigen der Experimente die von UNICORN ausgegeben Graphen nicht gespeichert werden, so dass der zeitliche Verlauf beispielsweise bei der Optimierung nicht ersichtlich ist.

4.1 Erkennung der Ursachen für hohen Energieverbrauch

In diesem Experiment soll die Ursache für den hohen Energieverbrauch ermittelt und eine Lösung hierfür gefunden werden.

Wie in Abbildung **??** zu sehen ist, entspricht der *Gain*, d. h. der Zuwachs an Performance und somit die Verringerung des Energieverbrauchs, bei unseren Experimenten den Ergebnissen der Autoren. Der in Abbildung **??** gezeigte kausale Graph gibt Aufschluss darüber, welche Konfigurationsparameter Einfluss auf den Energieverbrauch haben. Die genauen Werte sind im von UNICORN ausgegeben Log zu sehen:

Wie zu erwarten schlägt UNICORN Änderungen bei den Frequenzen (`core_freq`, `gpu_freq` und `emc_freq`) vor. Die Ergebnisse der Autoren konnten in diesem Fall sehr gut repliziert werden.

4.2 Optimierung der Inferenzzeit

In diesem Experiment wird die minimale Latenz, d. h. die Inferenzzeit, optimiert.

Das lokale Optimum von 12 Sekunden Inferenzzeit erreicht UNICORN bereits nach 69 Iterationen. Auf `Win` benötigt es dafür ca. 280 Minuten, auf `Mac` benötigt es 70 Minuten, die nötige Laufzeit von Unicorn auf `Win` war also sehr viel länger war als auf `Mac` und damit deutlich länger als in der Dokumentation angegeben (ca. 90 Minuten). Grundsätzlich ist hierbei nicht davon auszugehen, dass die zugrundeliegende Hardware dafür verantwortlich ist, da die Autoren bei ihren Experimenten die gleiche CPU verwendet haben. Möglicherweise ist eine Systemkonfiguration außerhalb des Containers, in dem die Berechnungen stattfanden, dafür verantwortlich (Docker von `Win` basiert auf WSL2 was beim Testsystem der Autoren, das auf Ubuntu basiert, nicht der Fall ist).

Auch hier stimmen die Resultate mit den Ergebnissen im Paper überein. UNICORN erreicht das lokale Optimum bereits nach 69 Iterationen und damit schneller als SMAC. Andererseits ist das von UNICORN gefundene Optimum um 8 Sekunden besser als das von SMAC.

Beim Vergleich der Laufzeit von SMAC und UNICORN konnten wir sehr große Unterschiede feststellen. Obwohl die Laufzeit auf `Mac` wie im Paper beschrieben bis zu 90 Minuten dauern kann

	BUG	FIX
memory_growth	0.5	5.0000e-01
logical_devices	1.0	1.0000e+00
core_freq	1651200.0	1.5744e+06
gpu_freq	1651200.0	1.6512e+06
emc_freq	800000000.0	2.1330e+09
num_cores	2.0	2.0000e+00
scheduler.policy	1.0	1.0000e+00
vm.swappiness	100.0	1.0000e+02
vm.vfs_cache_pressure	500.0	5.0000e+02
vm.dirty_background_ratio	80.0	8.0000e+01
vm.drop_caches	3.0	3.0000e+00
vm.nr_hugepages	1.0	1.0000e+00
vm.overcommit_ratio	50.0	5.0000e+01
vm.overcommit_memory	1.0	1.0000e+00
vm.overcommit_hugepages	2.0	2.0000e+00
kernel.sched_child_runs_first	0.0	0.0000e+00
kernel.sched_rt_runtime_us	500000.0	5.0000e+05
vm.dirty_bytes	30.0	3.0000e+01
vm.dirty_background_bytes	60.0	6.0000e+01
vm.dirty_ratio	5.0	5.0000e+00
swap_memory	1.0	1.0000e+00
kernel.max_pids	32768.0	6.5536e+04
kernel.sched_latency_ns	24000000.0	2.4000e+07
kernel.sched_nr_migrate	256.0	2.5600e+02
kernel.cpu_time_max_percent	50.0	5.0000e+01
kernel.sched_time_avg_ms	1000.0	1.0000e+03

Abbildung 6: Ausgabe einer Iteration von UNICORN bei der Optimierung der Inferenzzeit. In der Mittleren Spalte befinden sich die Werte der fehlerhaften Konfiguration. Rechts befindet sich die von UNICORN vorgeschlagene Konfiguration.

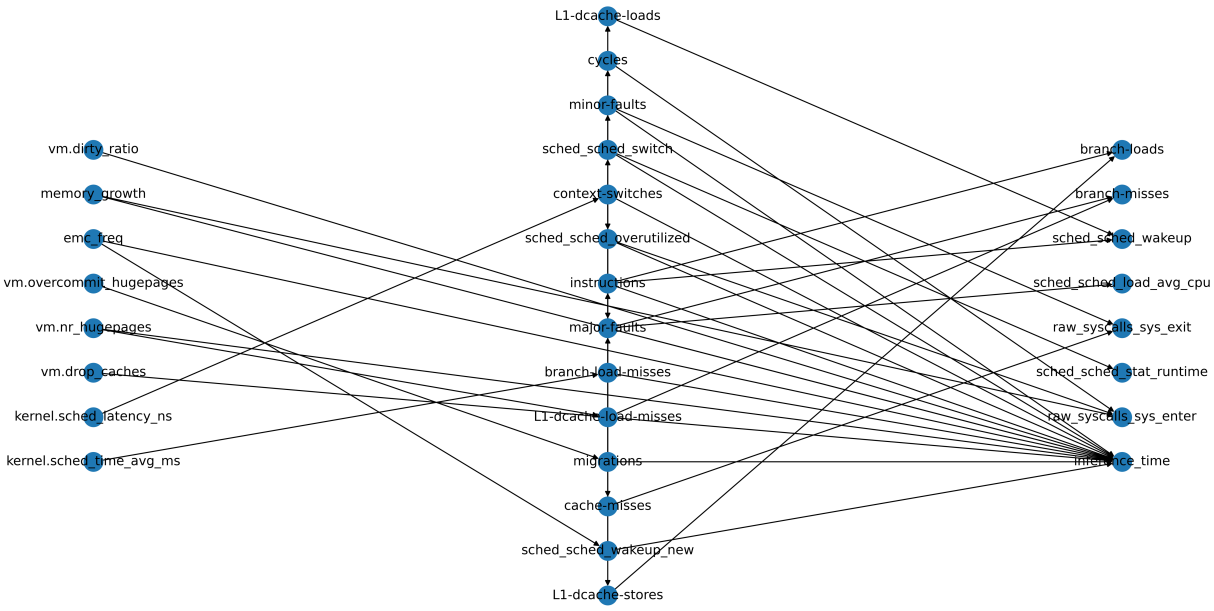

Abbildung 7: Von UNICORN gelernter kausaler Graph des Experiments aus Abschnitt **??**.

(in unserem Fall waren es ca. 70 Minuten), war SMAC mit einer Laufzeit von nur 50 Sekunden auf `Win` und 23 Sekunden auf `Mac` um ein Vielfaches schneller. Für die Optimierung der Inferenzzeit sollte also die lange Laufzeit von UNICORN berücksichtigt werden. Wird die Optimierung häufig ausgeführt ist SMAC mit seiner Geringeren Laufzeit vorzuziehen.

4.3 Änderung der Hardware

In diesem Experiment wird die Übertragbarkeit des von UNICORN gelernten Modells untersucht.

Für den Vergleich der Performance auf unterschiedlicher Hardware werden von den Autoren drei Fälle untersucht: Im ersten Fall (*Reuse*) wird das Modell mit der initialen Hardwarekonfiguration trainiert und anschließend der Gain auf der neuen Hardware ermittelt, ohne dabei das Modell neu zu trainieren oder neue Samples zur verfügung zu stellen. Im zweiten Fall (*Reuse+25*) wird das Modell auf die neue Hardware übertragen und mit 25 neuen Samples an die neue Hardwarekonfiguration angepasst. Wie weiter oben erwähnt benötigt UNICORN lediglich etwa 25 Samples um gute Ergebnisse zu erziehlen. Durch diese geringe Anzahl an Samples sollten somit die Performancezuwächse gegenüber dem *Reuse*-Fall verbessert werden, ohne dabei die gesamte Laufzeit für ein vollständiges Retraining zu benötigen. Im dritten Fall wird das Modell nicht übertragen und somit komplett neu trainiert.

Es ist zu erwarten, dass zwar der Gain von Fall 1 bis 3 zunimmt, gleichzeitig aber auch die Laufzeit.

Wir konnten die Ergebnisse der Autoren genau reproduzieren und konnten keine signifikanten Abweichungen bei den benötigen Laufzeiten feststellen. Wie vermutet ist der Gain bei *Reuse* des Modells am geringsten mit ca. 68 %, benötigt dabei allerdings kein Retraining und somit existiert keine Trainingszeit. Im Vergleich zum *Reuse+25*, bei dem das Modell ein Gain von etwa 80 % erzielt hat, benötigt UNICORN etwa 10 Minuten. Der Performancezuwachs zwischen *Reuse* und *Reuse+25* liegt somit bei ca. 12 %. Bei vollständigem Neutrainieren des Modells benötigt Unicorn mit 20 Minten am längsten, erzielt aber einen Gain von 82 %, also etwa 2 % mehr als im zweiten Fall mit nur 25 Samples. Der Performancezuwachs ist also zwischen diesen Fällen gering, was zeigt, dass UNICORN einerseits bereits mit keinen oder wenigen Samples gute Performaancezunahmen auf neuer Hardware erreicht, andererseits für noch bessere Ergebnisse nur wenige Samples benötigt.

Fazit

UNICORN ermöglicht das Optimieren von hochkonfigurierbaren System hinsichtlich Energieverbrauch undd Inferenzzeit. Dabei sind die von UNICORN gefundenen Konfigurationen, wie von den Autoren empirisch gezeigt und von uns bestätigt, fast immer besser als die von gängigen Modellen wie Bugdoc oder SMAC, benötigt jedoch wesentlich länger um diese Ergebnisse zu erzielen. Als Vorteil von UNICORN ist die geringe Anzahl an benötigten Samples und die guten Resultate bei der Übertragung des Modells auf neue Hardware zu nennen.

Bei der Reproduktion der Resultate des Originalpapers ist die gute Dokumentation hervorzuheben, wodurch die Replikation der Ergebnisse – bis auf wenige unkritische Probleme bei der Ausführung von Windows als Betriebssystem – sehr einfach war. Unsere Ergebnisse bei der Optimierung entsprechen denen im Originalpaper. Lediglich die Laufzeit auf Windows war in wenigen Ausnahmen wesentlich länger als von den Autoren angegeben.

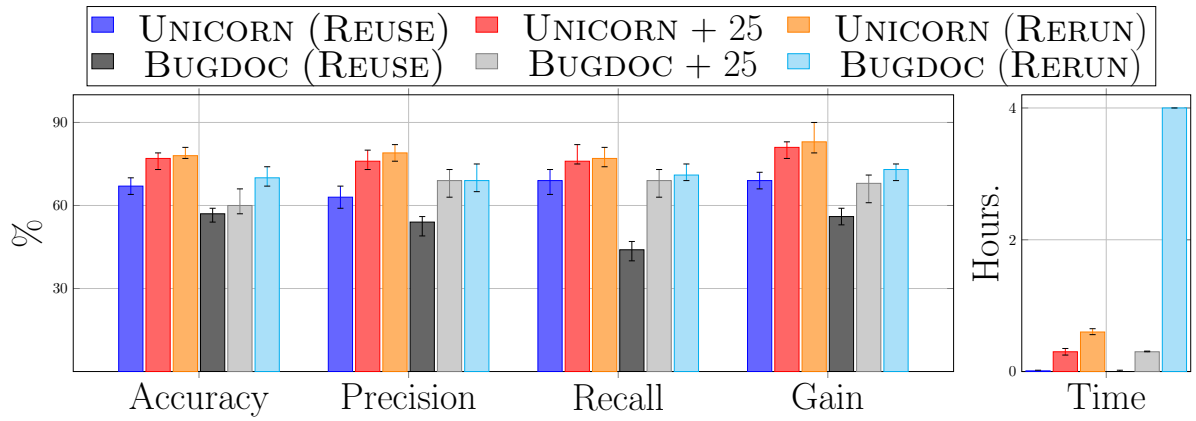

Abbildung 8: Vergleich der Performancemetriken von UNICORN mit Bugdoc. In allen Fällen hat UNIFORM bessere Werte als Bugdoc erziehlt. Für das Retraining der Modelle mit 25 Samples benötigen beide Algorithmen etwa gleich lang.