Automated Performance Analysis for Python Programs

Static Slicing and Path Reconstruction

Joshua Dawes[1,2,3] Giles Reger[1] Giovanni Franzoni[2]
Andreas Pfeiffer[2]

University of Manchester, Manchester, UK[1]

CERN, Geneva, Switzerland[2]

joshua.dawes@cern.ch[3]

Contents of this talk

Purpose of this talk

In this talk, I'll introduce some final parts of the theory for explanation of performance drops.

This moves us towards "coming full circle" - describing the intended behaviour of a system in Control-Flow Temporal Logic, monitoring it at runtime for agreement, and forming explanations of any performance patterns we see.

Control-Flow Temporal Logic is a low-level logic; it expresses constraints over state and time in control flow. Hence, explanations are in terms of the two things we can record at runtime: state and control flow.

Control-Flow Temporal Logic (CFTL)

A language in which to describe the state and time behaviour of a program, based on states and transitions.

State - instantaneous checkpoint, mapping variables to their values at that point in time.

Transition - computation required to move between states.

States can be denoted by q, and we use $q(x)$ to denote the value given to x in the state q.

Transitions can be denoted by $\langle q_i, q_j \rangle$, and we usually measure their duration $\text{duration}(\langle q_i, q_j \rangle)$.

$\text{time}(q)$ gives the time at which q was attained and $\text{time}(\langle q, q' \rangle) = \text{time}(q)$.

Semantics

A Symbolic Control-Flow Graph is a directed graph, with cycles allowed, whose vertices are symbolic states (these become states with concrete values at runtime) and whose edges are instructions moving between symbolic states (these become transitions at runtime).

A program's execution is represented by a Dynamic Run $\mathcal{D}$. Dynamic runs are a sequence of concrete states and transitions.

States in a dynamic run are generated by symbolic states in an SCFG.

Transitions in a dynamic run are generated by edges in an SCFG.

This correspondence is called symbolic support and is denoted by $s(\text{state/transition}) = \text{vertex/edge}$.

Static Slicing for CFTL

Consider a transition $t = \langle q, q' \rangle$ that represents a function f being called. Formally, we write $\langle q, q' \rangle \vdash \text{calls}(f)$.

Static slicing is used in the context of CFTL to determine the initial variables in control flow that contribute to the computation of the arguments of f, $\text{params}(f)$.

We denote the set of vertices of an SCFG which represent assignment to initial variables by $\bigcup_{\Pi} \text{initial}(\pi, e)$ for an edge e and set Π of possible paths through control flow.

Hence, initial variables depend on observations whose symbolic supports are the edge e.

Path Ambiguity

The main focus of this talk.

Entire program runs can be modelled by most-general dynamic runs $\mathcal{D}$.

Instrumenting a program wrt φ filters the entire dynamic run to give $\mathcal{D}'$, containing enough information to check φ.

What if φ requires observations between which multiple paths can be taken or up to which multiple paths can be taken?

Omitting the formal reasoning, this is the general idea behind path ambiguity.

How do we determine the path?

Extension of Theory

At the moment, we have a dynamic run $\mathcal{D}$. Here, there is no information about paths or branching conditions, hence the ambiguity.

We introduce Dynamic Runs with Conditions (DRCs); $\langle \mathcal{D}, \Phi \rangle$.

To define Φ, we need the notion of a branching point in the SCFG on which $\mathcal{D}$ is based.

Branching points of an SCFG are vertices whose out-degree is greater than 1.

So we define Φ to be a map
$\Phi : \{o \in \mathcal{D} : s(o) \text{ is a branching point}\} \to \text{Conditions}$.

Intuitively...

Dynamic Runs with Conditions are Dynamic Runs paired with maps that tell us at which vertices a decision was made on the path taken.

Notice that branching functions map from sets of observations. This is to take into account loops.

A DRC $\langle \mathcal{D}, \Phi \rangle$ does not tell us how to build Φ; it assumes it exists.

We build it via instrumentation.

Building a Branching Function Φ

We place instruments immediately after branching points in the SCFG.

This way, whenever a branch is chosen at runtime, we add to Φ.

This means that the domain of Φ, $\mathrm{dom}(\Phi)$, admits a natural total order wrt time, since it is populated at runtime. That is, for all $o, o' \in \mathrm{dom}(\Phi)$,

$$o \prec o' \iff \mathrm{time}(o) < \mathrm{time}(o')$$

This property of branching functions is vital to path reconstruction.

Path Reconstruction

Given a DRC $\langle \mathcal{D}, \Phi \rangle$ and some observation $o \in \mathcal{D}$ that is either a state or transition, we concern ourselves with reconstructing $\pi(o)$, the path from the start of the SCFG to the symbolic support of o, but including loop multiplicity.

Intuitively, loops in SCFGs are cycles, and paths reconstructed based on DRCs are allowed to go around cycles multiple times.

The construction of Φ by instruments at runtime mean Φ necessarily encodes loop iterations, which are not encoded in the static SCFG program representation.

We can almost perform path reconstruction, but we need to do a bit of work on our current notion of symbolic control-flow graphs first.

A Coordinate System for SCFGs

To perform path reconstruction, we need to be able to identify vertices in a symbolic control-flow graph uniquely.

Why? We need a way to refer to the target of the path reconstruction (normally $s(o)$ for some observation o).

The approach taken here is to build a coordinate system over symbolic control-flow graphs.

Let $\text{SCFG}(P) = \langle V, E, v_s \rangle$ be a symbolic control-flow graph.

Let Cond be the set of all branching conditions possible in a Python program.

A Coordinate System - continued

Then let $I_v : \text{Cond} \times \mathbb{N} \to V$ and let $I_e : \text{Cond} \times \mathbb{N} \to E$.

For any vertex v, v can be uniquely identified by:

1. Setting π_v to be the shortest path from v_s to v and taking the conjunction $\bigwedge \psi$ of all conditions of branching points on π_v.
2. Defining $C(v)$ to be $|\pi|$ of the shortest path π to v from the closest backwards reachable branching point.

Hence, the coordinate $\langle \bigwedge \psi, C(v) \rangle$ uniquely identifies v in SCFG(P).

Note: we omit conjuncts from $\bigwedge \psi$ if multiple branches from that branching point converge to the same path before v is encountered. We take a similar policy with computing $C(v)$.

Now for an Algorithm

To recap:

1. We can construct a branching function Φ at runtime, and
2. We can uniquely identify vertices to which a reconstructed path should go.

Note: the target vertex may appear multiple times in the path due to loop multiplicity. This is why we consider observations, not vertices.

A Path Reconstruction Algorithm

Data: SCFG$(P) = \langle V, E, v_s, C \rangle$, $\langle \mathcal{D}, \Phi \rangle$, $o \in \mathcal{D}$
Result: The path $\pi(o)$ from v_s to o based on $\mathcal{D}$
$\mathrm{dom}(\Phi) \leftarrow \langle \mathrm{dom}(\Phi), \prec_{\mathrm{OBS}} \rangle$;
$\mathrm{index}_\Phi \leftarrow 0$; $\mathrm{curr} \leftarrow v_s$; $\pi \leftarrow \langle\rangle$;
while $\mathrm{index}_\Phi < |\mathrm{dom}(\Phi)|$ **do**
  if $\mathrm{curr} \in \{s(q) : q \in \mathrm{dom}(\Phi)\}$ **then**
    $\mathrm{curr} \leftarrow$ target of edge leaving curr with condition $\Phi(\mathrm{dom}(\Phi)_{\mathrm{index}_\Phi})$;
    $\mathrm{index}_\Phi \leftarrow \mathrm{index}_\Phi + 1$;
  else
    $\mathrm{curr} \leftarrow$ next vertex;
  end
  $\pi \leftarrow \pi + \langle \mathrm{curr} \rangle$;
end
for $0 \leq i \leq C(s(o))$ **do**
  $\mathrm{curr} \leftarrow$ next vertex; $\pi \leftarrow \pi + \langle \mathrm{curr} \rangle$;
end

Where does this fit in?

In its abstract setting, the algorithm requires as input a symbolic control-flow graph $\mathrm{SCFG}(P) = \langle V, E, v_s, C \rangle$ (we can construct this easily from source code), a dynamic run with conditions $\langle \mathcal{D}, \Phi \rangle$ (information obtained from instrumentation) and an observation $o \in \mathcal{D}$ (a vertex coordinate combined with a timestamp).

All of these things are available offline, so path reconstruction is straightforward once we collect enough information to form a branching function Φ.

A sample path

A sample reconstructed path from monitoring a slightly altered function in the PMP service is given here.

I inserted spaces between lines to show loop multiplicity.

```
[<Vertex changing names [] 139772466283872>,
<Vertex changing names [] 139772466283872>,
<Vertex changing names ['i', 'sanitize'] 139772466344304>,
<Vertex changing names ['call', 'models.APICall'] 139772466344232>,
<Vertex changing names ['res', 'make_response'] 139772466344520>,

<Vertex changing names ['loop'] 139772467466464>,
<Vertex changing names ['print'] 139772466283368>,

<Vertex changing names ['loop'] 139772467466464>,
<Vertex changing names ['print'] 139772466283368>,

<Vertex changing names ['loop'] 139772467466464>,
<Vertex changing names ['print'] 139772466283368>,

<Vertex changing names ['loop'] 139772467466464>,
<Vertex changing names ['print'] 139772466283368>,

<Vertex changing names ['loop'] 139772467466464>,
<Vertex changing names ['print'] 139772466283368>,

<Vertex changing names ['post-loop'] 139772466283152>,

<Vertex changing names ['conditional'] 139772466753120>,
<Vertex changing names ['print'] 139772467467904>]
```

What's next

From here, we begin to develop theory that will allow us to collect and analyse statistics on the paths followed at runtime.

But I give opportunity for a pause here; if the audience prefers, we can pick this up in a second session.

From Paths to Initial Values

Consider initial(π, e), the set of initial variables on a path π based on an edge e. From this:

Initial variables depend on observations (via symbolic support) by $s(o) = e$ for some observation o.

Initial variables also depend on paths.

Path Statistics - a Key Problem

Suppose we want to look at the initial variables across multiple runs of the same program, on the same path.

So we start with a family of dynamic runs with conditions, $\langle \mathcal{D}_1, \Phi_1 \rangle, \langle \mathcal{D}_2, \Phi_2 \rangle, \ldots, \langle \mathcal{D}_n, \Phi_n \rangle$.

Problem: How do we know that they all took the same path through the SCFG?

We could just check for equality of the branching functions...

Roughly $f = g \iff \mathrm{dom}(f) = \mathrm{dom}(g) \wedge \forall x \in \mathrm{dom}(f), f(x) = g(x)$

But this won't work...

Path Statistics - a Key Problem continued

The domains $\mathrm{dom}(\Phi_1), \mathrm{dom}(\Phi_2), \ldots, \mathrm{dom}(\Phi_n)$ are likely disjoint, since:

- For a dynamic run with conditions $\langle \mathcal{D}, \Phi \rangle$, $\mathrm{dom}(\Phi)$ contains concrete states.
- Concrete states are generated at runtime, so contain timing information!
- Hence, for two concrete states $q_1 \in \mathrm{dom}(\Phi_1)$ and $q_2 \in \mathrm{dom}(\Phi_2)$ with $s(q_1) = s(q_2)$, probably $\mathrm{time}(q_1) \neq \mathrm{time}(q_2)$ and so $q_1 \in \mathrm{dom}(\Phi_1)$ but $q_1 \notin \mathrm{dom}(\Phi_2)$.

We need a more sophisticated notion of equality to develop path statistics. But first, there are a couple of things we need.

Labelling dom(Φ)

Domains of branching functions have a total order with respect to time.

This means we can label each element with a natural number, so that increasing time means increasing corresponding natural numbers.

We denote by $l_i : \mathbb{N} \to \text{dom}(\Phi_i)$ a labelling for Φ_i.

This holds the vital property that, for two states q, q', $l_i(n) = q$ and $l_i(n+1) = q' \implies$ there is no q'' with $\text{time}(q) < \text{time}(q'') < \text{time}(q')$.

Isomorphism of Branching Functions

Let $\langle \mathcal{D}_1, \Phi_1 \rangle$ and $\langle \mathcal{D}_2, \Phi_2 \rangle$ be dynamic runs with conditions.

We say the branching functions are isomorphic and write $\Phi_1 \cong \Phi_2$ if and only if

$$\overbrace{|\mathrm{dom}(\Phi_1)| = |\mathrm{dom}(\Phi_2)|}^{\text{there are the same number of branching points}}$$

$$\wedge$$

$$\underbrace{\forall\ 0 \leq i < |\mathrm{dom}(\Phi_1)| : \Phi_1(l_1(i)) = \Phi_2(l_2(i))}_{\text{no chance for divergence of paths}}$$

Path Equivalence

Path Equivalence is a relation between dynamic runs with conditions. We'll denote this by $\cong_\pi$, and write $\langle \mathcal{D}_1, \Phi_1 \rangle \cong_\pi \langle \mathcal{D}_2, \Phi_2 \rangle$ to denote path equivalent dynamic runs with conditions.

We write $\langle \mathcal{D}_1, \Phi_1 \rangle \cong_\pi \langle \mathcal{D}_2, \Phi_2 \rangle \iff \Phi_1 \cong \Phi_2$.

If dynamic runs with conditions are path equivalent, they follow the same path (including loop multiplicity).

Path Statistics

For a family of dynamic runs with conditions, $\langle \mathcal{D}_1, \Phi_1 \rangle, \langle \mathcal{D}_2, \Phi_2 \rangle, \ldots, \langle \mathcal{D}_n, \Phi_n \rangle$, if we care about the initial variables recorded for a fixed path π, then we first find the family of dynamic runs with conditions which are pairwise path equivalent.

$$\{\langle \mathcal{D}_i, \Phi_i \rangle \mid \forall \langle \mathcal{D}_j, \Phi_j \rangle \neq \langle \mathcal{D}_i, \Phi_i \rangle : \langle \mathcal{D}_i, \Phi_i \rangle \cong_\pi \langle \mathcal{D}_j, \Phi_j \rangle\}$$

We then take the initial variable values from those.

Path Statistics - continued

Given a program P, let Π be the finite set of paths through its SCFG.

For a dynamic run $\mathcal{D}$, let $\text{value}_{\text{init}}(x, \mathcal{D})$ denote the initial value of x in $\mathcal{D}$.

Then we can build a map, for a fixed SCFG(P), observation o with symbolic support $s(o) = e$ and family $\langle \mathcal{D}_1, \Phi_1 \rangle, \langle \mathcal{D}_2, \Phi_2 \rangle, \ldots, \langle \mathcal{D}_n, \Phi_n \rangle$ of DRCs:

$$\begin{aligned}&\text{valuations}(\mathcal{D}, o) = \\ &\quad \{[x_1 \mapsto \text{value}_{\text{init}}(x_1, \mathcal{D}), \ldots, x_n \mapsto \text{value}_{\text{init}}(x_n, \mathcal{D})] : \\ &\qquad x_i \in \text{initial}(\pi(o), s(o))\}\end{aligned}$$

Intuitive Meaning

We can fix a path through a program P, along with a point in code $e = s(o)$ (which generates an observation o) and, for any program run, we can see the initial variable values associated with that path.

This is automatic - engineers need only write a specification φ and path reconstruction, and subsequent statistic gathering, can be done fully automatically.

valuations$(\mathcal{D}, o)$ for a fixed pair $(\mathcal{D}, o)$ can further be used to determine the classes of the initial variable assignments that result in certain verdicts.

Do certain paths always violate our specification? Or certain initial variable assignments on a given path?

Thank you - any questions?