A Journey Towards Efficient Profiling

Alpine Verification Meeting 2023

Jiří Pavela

E-mail: ipavela@fit.vutbr.cz
Github: `https://github.com/JiriPavela/`
Perun Github: `https://github.com/Perfexionists/perun/`
Paper Demo VM: `10.5281/zenodo.6783242`

Brno University of Technology, Faculty of Information Technology

Supported by:

Red Hat, Inc.
Czech Science Foundation Project 20-07487S
Czech Science Foundation Project 23-06506S
JCMM PhD Talent Scholarship Programme

Motivation:
Why Care About Performance?

Performance Bugs Are Everywhere

- Software performance bugs are an omnipresent problem[1]:

[1] Source: https://accidentallyquadratic.tumblr.com/

Performance Bugs Are Everywhere

- Software performance bugs are an omnipresent problem[1]:
  - **Cluster computing engine** may *freeze* after an update!

- An internal check for uniqueness
  $\rightarrow$ *hanging effectively forever* for large job batch.

[1] Source: `https://accidentallyquadratic.tumblr.com/`

Performance Bugs Are Everywhere

- Software performance bugs are an omnipresent problem[1]:
  - **Cluster computing engine** may *freeze* after an update!
  - **Cloud services** may *crash*!

Apache Spark

- An internal check for uniqueness
  → *hanging effectively forever* for large job batch.

stackoverflow

- A regular expression for stripping whitespaces
  → *34 minutes long outage*.

[1]Source: `https://accidentallyquadratic.tumblr.com/`

Performance Bugs Are Everywhere

- Software performance bugs are an omnipresent problem[1]:
  - **Cluster computing engine** may *freeze* after an update!
  - **Cloud services** may *crash*!
  - **Parsers** may experience significant *slowdown*!

- An internal check for uniqueness
  → *hanging effectively forever* for large job batch.
- A regular expression for stripping whitespaces
  → *34 minutes long outage*.
- One of *Chrome*'s parsers
  → *noticeable slowdown* for long lines.

[1] Source: `https://accidentallyquadratic.tumblr.com/`

Performance Bugs Are Everywhere (yes, even in your code)

- $\mathcal{O}(n^2)$ space implementation of C# constant folding
  $\rightarrow$ *compiler runs out of memory*.
- $\mathcal{O}(n^2)$ pattern matching algorithm in Elasticsearch
  $\rightarrow$ up to $\frac{1}{2}$ *CPU-time spent in* `Regex.simpleMatch`.
- Array used for tags lookup in Vim
  $\rightarrow$ $\mathcal{O}(n^2)$ complexity in the number of matches.
- Godoc source code parsing
  $\rightarrow$ $\mathcal{O}(n^2)$ loop for Go structs definitions.

Performance Bugs Are Everywhere (yes, even in your code)

- $\mathcal{O}(n^2)$ space implementation of C# constant folding
  → *compiler runs out of memory*.
- $\mathcal{O}(n^2)$ pattern matching algorithm in Elasticsearch
  → up to $\frac{1}{2}$ *CPU-time spent in* `Regex.simpleMatch`.
- Array used for tags lookup in Vim
  → $\mathcal{O}(n^2)$ complexity in the number of matches.
- Godoc source code parsing
  → $\mathcal{O}(n^2)$ loop for Go structs definitions.

- Such bugs usually manifest only under certain conditions.
  - Highly granular analysis may detect them sooner!

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

+ Possible formal guarantees.
+ *Soundness*/*Completeness*.
+ Often the only possibility for safety-critical software.

– High skill/tool barrier.
– Scaling issues.
– The analysis *may fail* for complex cases.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

+ Easier tools adoption.
+ Usually scales well.

– Few formal guarantees.
– High-level coarse analysis.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

\+ Well-established approach and easy adoption.

\+ Good *CI/CD* support.

\+ Scales reasonably well.

– No formal guarantees.

– Typically coarse analysis.

– Garbage in, garbage out.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

+ Well-established approach and easy adoption.
+ Good *CI/CD* support.
+ Scales reasonably well.

– No formal guarantees.
– Typically coarse analysis.
– Garbage in, garbage out.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

\+ Well-established approach.

\+ Good data granularity.

\+ Suitable for finding root causes of performance issues.

– No formal guarantees.

– Insufficient *CI/CD* support.

– Potentially significant overhead.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

\+ Well-established approach.

\+ Good data granularity.

\+ Suitable for finding root causes of performance issues.

– No formal guarantees.

– Insufficient *CI/CD* support.

– Potentially significant overhead.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

\+ Well-established approach.

\+ Good data granularity.

\+ Suitable for finding root causes of performance issues.

– No formal guarantees.

– Insufficient *CI/CD* support.

– Potentially significant overhead.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

+ Well-established approach.
+ Good data granularity.
+ Suitable for finding root causes of performance issues.

– No formal guarantees. ☹
– Insufficient *CI/CD* support.
– Potentially significant overhead.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

+ Well-established approach.
+ Good data granularity.
+ Suitable for finding root causes of performance issues.

– No formal guarantees. ☹
– Insufficient *CI/CD* support.
– Potentially significant overhead.

How Do We Find The Bugs?

- **Numerous static and dynamic analysis approaches:**
  - Worst-case resource bounds analysis.
  - Anti-patterns detection and log analysis.
  - Performance testing and benchmarking.
  - Profiling (event-based tracing).

+ Well-established approach.
+ Good data granularity.
+ Suitable for finding root causes of performance issues.

– No formal guarantees. ☹
– Insufficient *CI/CD* support. 💡
– Potentially significant overhead. 💡

The Roots of Profiling Inefficiency

Too Much Overhead

- **High granularity** of performance data.
  - $\Rightarrow$ Easier identification of performance bugs root causes.
  - $\Rightarrow$ Significant time (and possibly memory) overhead.

Too Much Overhead

- **High granularity** of performance data.
  - $\Rightarrow$ Easier identification of performance bugs root causes.
  - $\Rightarrow$ Significant time (and possibly memory) overhead.

Too Much Overhead

- **High granularity** of performance data.
  - $\Rightarrow$ Easier identification of performance bugs root causes.
  - $\Rightarrow$ Significant time (and possibly memory) overhead. ☹

Too Much Overhead

- **High granularity** of performance data.
  - $\Rightarrow$ Easier identification of performance bugs root causes.
  - $\Rightarrow$ Significant time (and possibly memory) overhead. ☹

- Can we somehow reduce the overhead?

Too Much Overhead

- **High granularity** of performance data.
  - $\Rightarrow$ Easier identification of performance bugs root causes.
  - $\Rightarrow$ Significant time (and possibly memory) overhead. ☹

- Can we somehow reduce the overhead?
  1. Reduce the granularity.

Too Much Overhead

- **High granularity** of performance data.
  - $\Rightarrow$ Easier identification of performance bugs root causes.
  - $\Rightarrow$ Significant time (and possibly memory) overhead. ☹

- Can we somehow reduce the overhead?
  1. Reduce the granularity.
     - We risk reaching the benchmarking territory *precision-wise*.

Too Much Overhead

- **High granularity** of performance data.
  - $\Rightarrow$ Easier identification of performance bugs root causes.
  - $\Rightarrow$ Significant time (and possibly memory) overhead. ☹

- Can we somehow reduce the overhead?
  1. ~~Reduce the granularity.~~
     - We risk reaching the benchmarking territory *precision-wise*.
  2. Optimize the instrumentation.

Too Much Overhead

- **High granularity** of performance data.
  ⇒ Easier identification of performance bugs root causes.
  ⇒ Significant time (and possibly memory) overhead. ☹

- Can we somehow reduce the overhead?
  1. ~~Reduce the granularity.~~
     - We risk reaching the benchmarking territory *precision-wise*.
  2. Optimize the instrumentation.
     - Typically tailored only for a specific language or environment.

Too Much Overhead

- **High granularity** of performance data.
  - ⇒ Easier identification of performance bugs root causes.
  - ⇒ Significant time (and possibly memory) overhead. ☹
- Can we somehow reduce the overhead?
  1. ~~Reduce the granularity.~~
     - We risk reaching the benchmarking territory *precision-wise*.
  2. ~~Optimize the instrumentation.~~
     - Typically tailored only for a specific language or environment.
  3. General profiling optimization techniques.[2]

[2] *Disclaimer: publication of the following concepts and ideas is a work in progress.*

Too Much Overhead

- **High granularity** of performance data.
  - ⇒ Easier identification of performance bugs root causes.
  - ⇒ Significant time (and possibly memory) overhead. ☹

- Can we somehow reduce the overhead?
  1. ~~Reduce the granularity.~~
     - We risk reaching the benchmarking territory *precision-wise*.
  2. ~~Optimize the instrumentation.~~
     - Typically tailored only for a specific language or environment.
  3. General profiling optimization techniques.[2]
     - ⇒ The idea: Limit high granularity to **where it actually matters**.

[2] *Disclaimer: publication of the following concepts and ideas is a work in progress.*

Recency

- **Recency is important:** it pays off to discover bugs quickly.
  - Recently introduced bugs, as opposed to dormant bugs[3],
    - take on average less time to fix;
    - can be fixed by less experienced developers;
    - the fix is generally smaller.

[3] T.-H. Chen et al.: *An empirical study of dormant bugs*

Recency

- **Recency is important:** it pays off to discover bugs quickly.
    - Recently introduced bugs, as opposed to dormant bugs[3],
        - take on average **less time to fix** (5 vs. 8 days);
        - can be fixed by less experienced developers;
        - the fix is generally smaller.

[3] T.-H. Chen et al.: *An empirical study of dormant bugs*

Recency

- **Recency is important:** it pays off to discover bugs quickly.
  - Recently introduced bugs, as opposed to dormant bugs[3],
    - take on average **less time to fix** (5 vs. 8 days);
    - can be fixed by **less experienced developers**;
    - the fix is generally smaller.

[3] T.-H. Chen et al.: *An empirical study of dormant bugs*

Recency

- **Recency is important:** it pays off to discover bugs quickly.
  - Recently introduced bugs, as opposed to dormant bugs[3],
    - take on average **less time to fix** (5 vs. 8 days);
    - can be fixed by **less experienced developers**;
    - the fix is generally **smaller** (10 vs. 19 LoC).

[3] T.-H. Chen et al.: *An empirical study of dormant bugs*

Recency

- **Recency is important:** it pays off to discover bugs quickly.
  - Recently introduced bugs, as opposed to dormant bugs[3],
    - take on average **less time to fix** (5 vs. 8 days);
    - can be fixed by **less experienced developers**;
    - the fix is generally **smaller** (10 vs. 19 LoC).
- Hence, new bugs should be discovered **as soon as possible**.
  - Already commonly utilized for testing the project's functionality.

[3] T.-H. Chen et al.: *An empirical study of dormant bugs*

Recency

- **Recency is important:** it pays off to discover bugs quickly.
  - Recently introduced bugs, as opposed to dormant bugs[3],
    - take on average **less time to fix** (5 vs. 8 days);
    - can be fixed by **less experienced developers**;
    - the fix is generally **smaller** (10 vs. 19 LoC).
- Hence, new bugs should be discovered **as soon as possible**.
  - Already commonly utilized for testing the project's functionality.

[3] T.-H. Chen et al.: *An empirical study of dormant bugs*

Recency in Profiling

- Profiling is usually only done **late** in the project development.

Recency in Profiling

- Profiling is usually only done **late** in the project development.
  ⇒ The idea: *Automated profiling throughout the development process.*

Recency in Profiling

- Profiling is usually only done **late** in the project development.
  - $\Rightarrow$ The idea: *Automated profiling throughout the development process.*
- Profiling tools generally **ignore project and profiling history**.
- Yet, past profiles coupled with version history are valuable.

Recency in Profiling

- Profiling is usually only done **late** in the project development.
  ⇒ The idea: *Automated profiling throughout the development process.*
- Profiling tools generally **ignore project and profiling history**.
- Yet, past profiles coupled with version history are valuable.
  ⇒ The idea: *Reuse profiling data when possible.*

Meet Perun:
Performance Version System

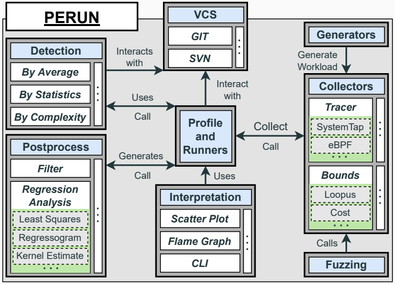
Perun: Beyond Mere Profiling

Perun[4] = Complex Solution for Performance Analysis and Testing

[4] T. Fiedor, J. Pavela, A. Rogalewicz and T. Vojnar: *Perun: Performance Version System*, in Proc. of *ICSME*'22

Perun[4] = Complex Solution for Performance Analysis and Testing =

= **Collects** performance data



[4] T. Fiedor, J. Pavela, A. Rogalewicz and T. Vojnar: *Perun: Performance Version System*, in Proc. of *ICSME*'22

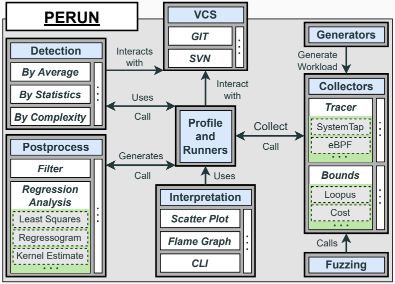
Perun: Beyond Mere Profiling

Perun[4] $=$ Complex Solution for Performance Analysis and Testing $=$

$=$ **Collects** performance data

$+$ **Creates** performance models

- Constant c, linear $an + b$, ...



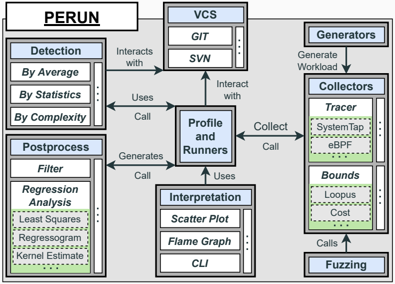
[4] T. Fiedor, J. Pavela, A. Rogalewicz and T. Vojnar: *Perun: Performance Version System*, in Proc. of *ICSME*'22

Perun[4] = Complex Solution for Performance Analysis and Testing =

= **Collects** performance data

+ **Creates** performance models
  - Constant c, linear $an + b$, ...

+ **Integrates** VCS
  - Access to project history.



[4] T. Fiedor, J. Pavela, A. Rogalewicz and T. Vojnar: *Perun: Performance Version System*, in Proc. of *ICSME*'22

Perun: Beyond Mere Profiling

Perun[4] = Complex Solution for Performance Analysis and Testing =

= **Collects** performance data

\+ **Creates** performance models

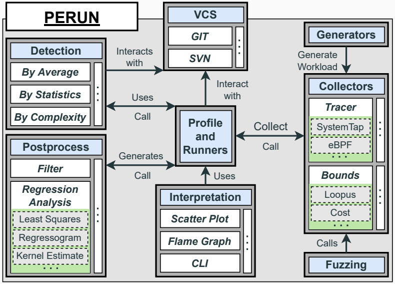
- Constant c, linear $an + b$, ...

\+ **Integrates** VCS

- Access to project history.

\+ **Detects** performance changes

- Degradations, optimizations.



[4] T. Fiedor, J. Pavela, A. Rogalewicz and T. Vojnar: *Perun: Performance Version System*, in Proc. of *ICSME*'22

Perun[4] = Complex Solution for Performance Analysis and Testing =

= **Collects** performance data

\+ **Creates** performance models

- Constant c, linear $an + b$, ...

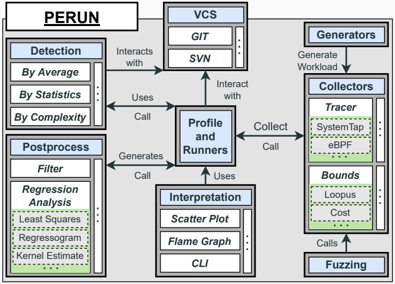
\+ **Integrates** VCS

- Access to project history.

\+ **Detects** performance changes

- Degradations, optimizations.

\+ **Visualizes** performance



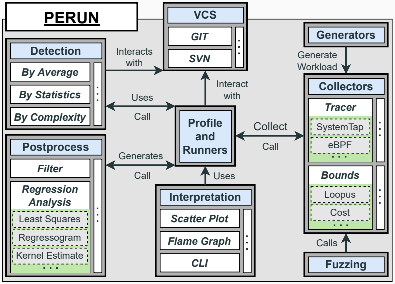
[4] T. Fiedor, J. Pavela, A. Rogalewicz and T. Vojnar: *Perun: Performance Version System*, in Proc. of *ICSME*'22

Perun: Beyond Mere Profiling

Perun[4] = Complex Solution for Performance Analysis and Testing =

= **Collects** performance data*

+ **Creates** performance models
    - Constant c, linear $an + b$, ...
+ **Integrates** VCS
    - Access to project history.
+ **Detects** performance changes
    - Degradations, optimizations.
+ **Visualizes** performance*

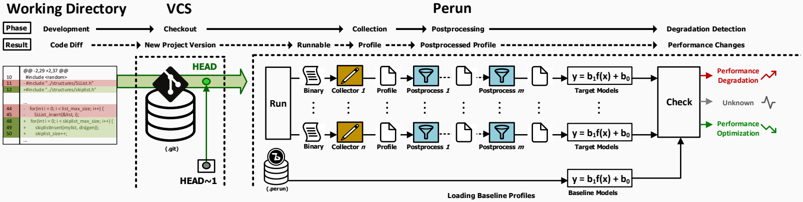


* Often the only steps done by traditional profilers.

[4] T. Fiedor, J. Pavela, A. Rogalewicz and T. Vojnar: *Perun: Performance Version System*, in Proc. of *ICSME*'22

Perun Workflow: Overview

- **Four** major steps: Repository → Profiles → Models → Detection



Perun Workflow: Repository

Working Directory

Phase **Development**

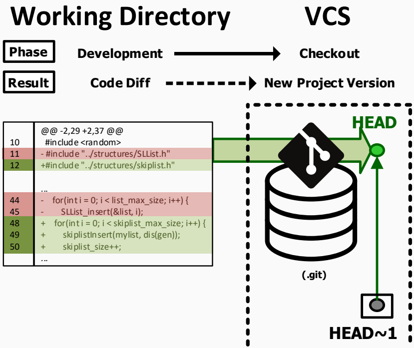
Result **Code Diff**

```
@@ -2,29 +2,37 @@
 #include <random>
- #include "../structures/SLList.h"
+#include "../structures/skiplist.h"

...
-    for(int i = 0; i < list_max_size; i++) {
-        SLList_insert(&list, i);
+    for(int i = 0; i < skiplist_max_size; i++) {
+        skiplistInsert(mylist, dis(gen));
+        skiplist_size++;
...
```

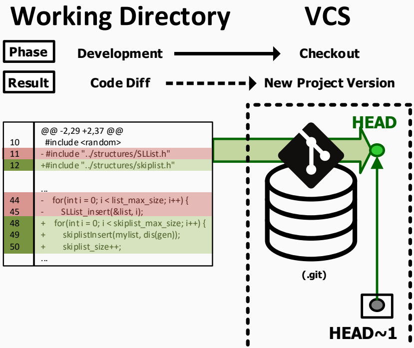
1. We create the project's **working directory**.

Perun Workflow: Repository



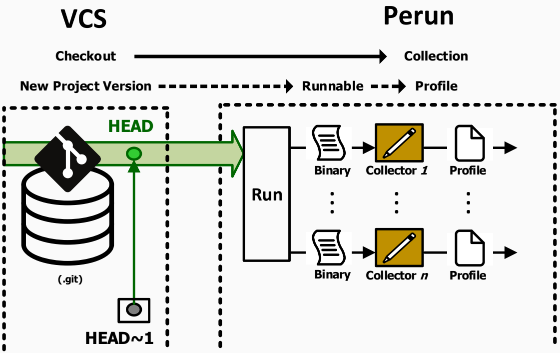
1. We create the project's **working directory**.
2. We initialize a **VCS** (e.g., Git) for project versioning.

Perun Workflow: Repository



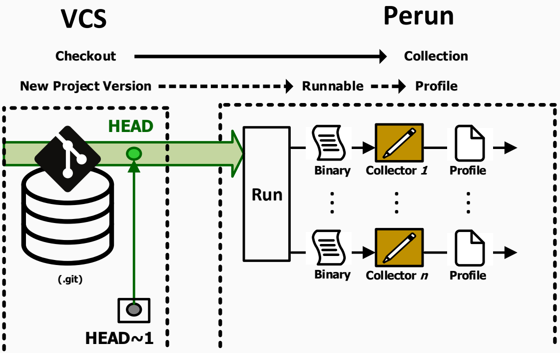
1. We create the project's **working directory**.
2. We initialize a **VCS** (e.g., Git) for project versioning.
3. We initialize **Perun** in the repository alongside the VCS.

Perun Workflow: Profiles



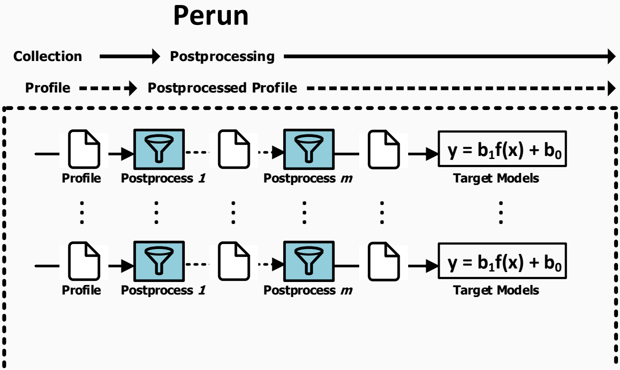
4. We measure project's performance and obtain **profiles**.
    - Profiles are stored within Perun and linked to the corresponding VCS version (e.g., commit).

Perun Workflow: Profiles



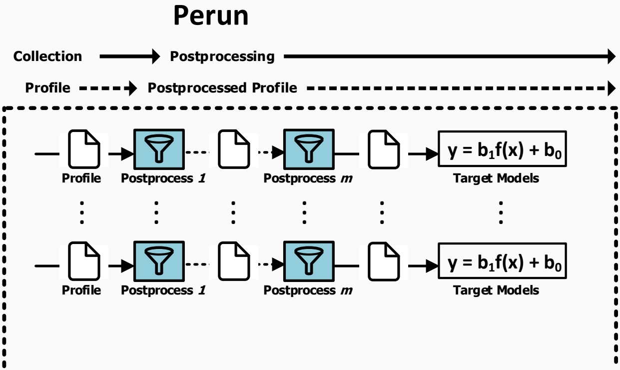
4. We measure project's performance and obtain **profiles**.
   - Profiles are stored within Perun and linked to the corresponding VCS version (e.g., commit).

Perun Workflow: Models



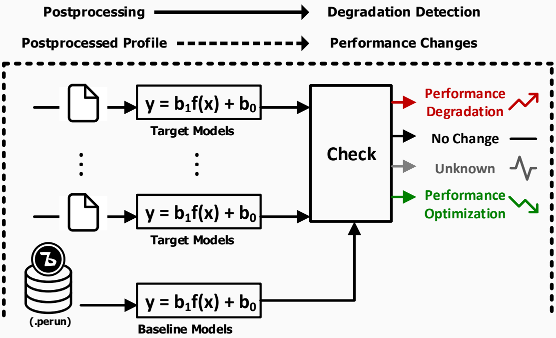
5. We create **performance models** from profiles using postprocessors.
    - Models are stored within Perun alongside the profiles.

Perun Workflow: Models



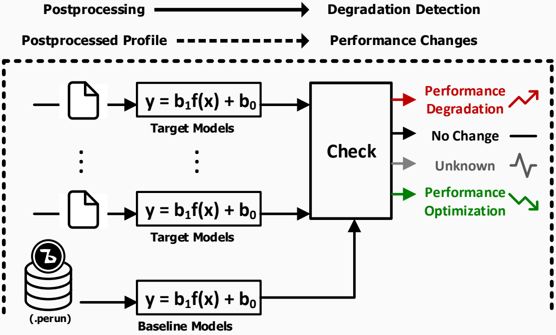
5. We create **performance models** from profiles using postprocessors.
   - Models are stored within Perun alongside the profiles.

Perun Workflow: Detection



6. We detect **performance changes** using models or directly profiles.
   - Target refers to the current version.
   - Baseline refers to the previous version used for comparison.

Perun Workflow: Detection



6. We detect **performance changes** using models or directly profiles.
   - Target refers to the current version.
   - Baseline refers to the previous version used for comparison.

Perun Demonstration: Finding Performance Changes

CPython: Issue #92356

- **CPython**: Reference C implementation of a Python interpreter.
- **Issue #92356**[5]: A performance regression in `ctypes` module.
  - $\approx$ 8% higher function call overhead (py3.11.0a7 vs. py3.10.4).
  - Replicated using the `pyperformance ctypes` benchmark.

[5] Reported by user `mdboom`: `https://github.com/python/cpython/issues/92356`

CPython: Issue #92356

- **CPython**: Reference C implementation of a Python interpreter.
- **Issue #92356**[5]: A performance regression in `ctypes` module.
  - $\approx$ 8% higher function call overhead (py3.11.0a7 vs. py3.10.4).
  - Replicated using the `pyperformance ctypes` benchmark.
  - **Fixed soon after the report.**

[5] Reported by user `mdboom`: `https://github.com/python/cpython/issues/92356`

CPython: Issue #92356

- **CPython**: Reference C implementation of a Python interpreter.
- **Issue #92356**[5]: A performance regression in `ctypes` module.
  - $\approx 8\%$ higher function call overhead (py3.11.0a7 vs. py3.10.4).
  - Replicated using the `pyperformance ctypes` benchmark.
  - **Fixed soon after the report.**
- Discovering such issues and finding their root cause is the hard part.

[5] Reported by user `mdboom`: `https://github.com/python/cpython/issues/92356`

CPython: Issue #92356

- **CPython**: Reference C implementation of a Python interpreter.
- **Issue #92356**[5]: A performance regression in `ctypes` module.
  - $\approx 8\%$ higher function call overhead (py3.11.0a7 vs. py3.10.4).
  - Replicated using the `pyperformance ctypes` benchmark.
  - **Fixed soon after the report.**
- Discovering such issues and finding their root cause is the hard part.
  - Can Perun help us here?

[5] Reported by user `mdboom`: `https://github.com/python/cpython/issues/92356`

CPython: Setup and Baseline

Using **Perun**, we could handle the **issue #92356** as follows:

CPython: Setup and Baseline

Using **Perun**, we could handle the **issue #92356** as follows:

1. We **initialize** a CPython repository with Perun.

Perun commands

```
perun init
```

CPython: Setup and Baseline

Using **Perun**, we could handle the **issue #92356** as follows:

1. We **initialize** a CPython repository with Perun.
2. We **store** a profile for CPython **v3.10.4** ctypes benchmark in Perun.

Perun commands

```
perun init
```

CPython: Setup and Baseline

Using **Perun**, we could handle the **issue #92356** as follows:

1. We **initialize** a CPython repository with Perun.
2. We **store** a profile for CPython **v3.10.4** ctypes benchmark in Perun.
    - We denote this profile as baseline.
    - Perun handles the *profile-commit* link internally.

Perun commands

```
perun init
```

CPython: Detecting Changes

3. CPython v3.11.0a7 **rolls out**.

CPython: Detecting Changes

3. CPython v3.11.0a7 **rolls out**.
4. We **profile** the ctypes benchmark for CPython **v3.11.0a7**.

Perun commands

```
perun collect -c <py3.11.0a7> -a <benchmark> trace -b <files>
```

CPython: Detecting Changes

3. CPython v3.11.0a7 **rolls out**.
4. We **profile** the ctypes benchmark for CPython **v3.11.0a7**.
   - We denote the resulting profile as target.

Perun commands

```
perun collect -c <py3.11.0a7> -a <benchmark> trace -b <files>
perun add <target>
```

CPython: Detecting Changes

3. CPython v3.11.0a7 **rolls out**.
4. We **profile** the ctypes benchmark for CPython **v3.11.0a7**.
    - We denote the resulting profile as target.
5. We **compare** the baseline and target profiles.

Perun commands

```
perun collect -c <py3.11.0a7> -a <benchmark> trace -b <files>
perun add <target>
perun check -f profiles <baseline> <target>
```

CPython: Detecting Changes

3. CPython v3.11.0a7 **rolls out**.
4. We **profile** the ctypes benchmark for CPython **v3.11.0a7**.
   - We denote the resulting profile as target.
5. We **compare** the baseline and target profiles.
   - Perun supports multiple comparison algorithms.
   - For this particular issue, we used *Exclusive-Time Outliers*.

Perun commands

```
perun collect -c <py3.11.0a7> -a <benchmark> trace -b <files>
perun add <target>
perun check -f profiles <baseline> <target>
```

CPython: Regression Detected

Location	**Result**	**T∆ [ms]**	**T∆ [%]**
_ctypes_init_fielddesc	NotInBaseline	77.95	5.23
_ctypes_get_fielddesc	SevereDegradation	52.9	3.55
_ctypes_callproc	Degradation	2.84	0.19
	...		
_ctypes.cpython-311	TotalDegradation	136.92	**9.19**

⋆ T∆: exclusive-time delta of *target* − *baseline*.

CPython: Regression Detected

Location	Result	TΔ [ms]	TΔ [%]
_ctypes_init_fielddesc	NotInBaseline	77.95	**5.23**
_ctypes_get_fielddesc	SevereDegradation	52.9	**3.55**
_ctypes_callproc	Degradation	2.84	0.19
	...		
_ctypes.cpython-311	TotalDegradation	136.92	9.19

⋆ TΔ: exclusive-time delta of *target* − *baseline*.

CPython: Regression Detected

Location	Result	TΔ [ms]	TΔ [%]
_ctypes_init_fielddesc	NotInBaseline	77.95	**5.23**
_ctypes_get_fielddesc	SevereDegradation	52.9	**3.55**
_ctypes_callproc	Degradation	2.84	0.19
	...		
_ctypes.cpython-311	TotalDegradation	136.92	9.19

* TΔ: exclusive-time delta of *target* − *baseline*.

- **Root cause** of the issue: repeated calls of an init function.

Function `_ctypes_get_fielddesc`

```
if (!initialized) {
    _ctypes_init_fielddesc();
}
```

CPython: Hotfix

6. We **create** a new hotfix branch and **fix** the issue.

Fixing `_ctypes_get_fielddesc`

```
  if (!initialized) {
+     initialized = 1;
      _ctypes_init_fielddesc();
  }
```

CPython: Hotfix

6. We **create** a new hotfix branch and **fix** the issue.

Fixing `_ctypes_get_fielddesc`

```
  if (!initialized) {
+      initialized = 1;
       _ctypes_init_fielddesc();
  }
```

1. We **Profile** the CPython hotfixed version.

Perun commands

```
perun collect -c <py3.11.0a7-fix> -a <benchmark> trace <...>
```

CPython: Hotfix

6. We **create** a new hotfix branch and **fix** the issue.

Fixing `_ctypes_get_fielddesc`

```
  if (!initialized) {
+     initialized = 1;
      _ctypes_init_fielddesc();
  }
```

1. We **Profile** the CPython hotfixed version.
    - We denote the resulting profile as hotfix.

Perun commands

```
perun collect -c <py3.11.0a7-fix> -a <benchmark> trace <...>
perun add <hotfix>
```

CPython: Hotfix

6. We **create** a new hotfix branch and **fix** the issue.

Fixing _ctypes_get_fielddesc

```
 if (!initialized) {
+    initialized = 1;
     _ctypes_init_fielddesc();
 }
```

1. We **Profile** the CPython hotfixed version.
   - We denote the resulting profile as hotfix.
2. We **compare** the baseline and hotfix profiles.

Perun commands

```
perun collect -c <py3.11.0a7-fix> -a <benchmark> trace <...>
perun add <hotfix>
perun check -f profiles <baseline> <hotfix>
```

CPython: Issue Fixed!

Location	Result	Δ [ms]	Δ [%]	Δ_{old} [ms]	Δ_{old} [%]
	...				
_ctypes_get_fielddesc	MaybeDegradation	0.89	0.06	52.9	3.55
_ctypes_init_fielddesc	NotInBaseline	0.02	0.00	77.95	5.23
_ctypes.cpython-311	TotalDegradation	23.45	1.70	136.92	9.19

⋆ Δ: exclusive-time delta of *hotfix*−*baseline*.

⋆ Δ_{old}: exclusive-time delta of *target*−*baseline*.

CPython: Issue Fixed!

Location	Result	Δ [ms]	Δ [%]	Δ_{old} [ms]	Δ_{old} [%]
		...			
_ctypes_get_fielddesc	MaybeDegradation	0.89	**0.06**	52.9	3.55
_ctypes_init_fielddesc	NotInBaseline	0.02	0.00	77.95	5.23
_ctypes.cpython-311	TotalDegradation	23.45	1.70	136.92	9.19

⋆ Δ: exclusive-time delta of *hotfix*−*baseline*.

⋆ Δ_{old}: exclusive-time delta of *target*−*baseline*.

- The **_ctypes_get_fielddesc** Δ has improved significantly.

CPython: Issue Fixed!

Location	Result	Δ [ms]	Δ [%]	Δ_{old} [ms]	Δ_{old} [%]
		...			
_ctypes_get_fielddesc	MaybeDegradation	0.89	**0.06**	52.9	3.55
_ctypes_init_fielddesc	NotInBaseline	0.02	**0.00**	77.95	5.23
_ctypes.cpython-311	TotalDegradation	23.45	1.70	136.92	9.19

* Δ: exclusive-time delta of *hotfix*−*baseline*.

* Δ_{old}: exclusive-time delta of *target*−*baseline*.

- The **_ctypes_get_fielddesc** Δ has improved significantly.
- The **_ctypes_init_fielddesc** Δ is now negligible.

CPython: Issue Fixed!

Location	Result	Δ [ms]	Δ [%]	Δ_{old} [ms]	Δ_{old} [%]
		...			
_ctypes_get_fielddesc	MaybeDegradation	0.89	**0.06**	52.9	3.55
_ctypes_init_fielddesc	NotInBaseline	0.02	**0.00**	77.95	5.23
_ctypes.cpython-311	TotalDegradation	23.45	1.70	136.92	9.19

* Δ: exclusive-time delta of *hotfix*−*baseline*.

* Δ_{old}: exclusive-time delta of *target*−*baseline*.

- The **_ctypes_get_fielddesc** Δ has improved significantly.
- The **_ctypes_init_fielddesc** Δ is now negligible.

$\Rightarrow$ Perun leverages **VCS** and **Recency** to successfully discover and help locate performance issues in new project versions as soon as possible.

Efficient Profiling Techniques

Core Concepts of Efficient Profiling

Observation 1

A **subset** of profiled functions is responsible for sizable portion of the overhead while producing **uninteresting performance models**, e.g.:

- Hundreds of millions of times called $\mathcal{O}(1)$ functions.
- Functions with constant-like runtime behaviors.

Core Concepts of Efficient Profiling

Observation 1

A **subset** of profiled functions is responsible for sizable portion of the overhead while producing **uninteresting performance models**, e.g.:

- Hundreds of millions of times called $\mathcal{O}(1)$ functions.
- Functions with constant-like runtime behaviors.

$\Rightarrow$ Such functions usually need not be profiled but their (possibly indirect) callers should.

Core Concepts of Efficient Profiling

Observation 1

A **subset** of profiled functions is responsible for sizable portion of the overhead while producing **uninteresting performance models**, e.g.:

- Hundreds of millions of times called $\mathcal{O}(1)$ functions.
- Functions with constant-like runtime behaviors.

$\Rightarrow$ Such functions usually need not be profiled but their (possibly indirect) callers should.

Observation 2

Even **performance-wise** significant functions that are to be profiled may generate substantial amount of records and thus cause **significant** time and space **overhead**.

Core Concepts of Efficient Profiling

Observation 1

A **subset** of profiled functions is responsible for sizable portion of the overhead while producing **uninteresting performance models**, e.g.:

- Hundreds of millions of times called $\mathcal{O}(1)$ functions.
- Functions with constant-like runtime behaviors.

$\Rightarrow$ Such functions usually need not be profiled but their (possibly indirect) callers should.

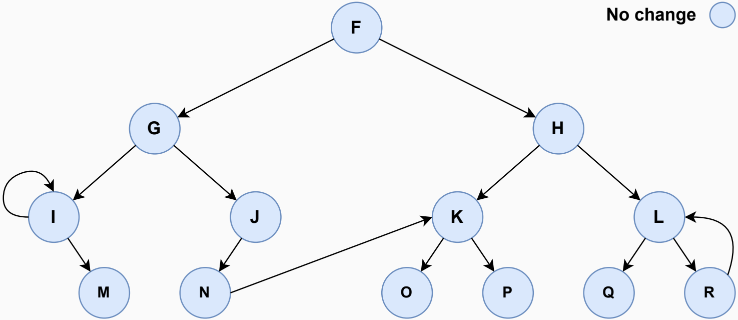
Observation 2

Even **performance-wise** significant functions that are to be profiled may generate substantial amount of records and thus cause **significant** time and space **overhead**.

$\Rightarrow$ Only a subset of the total data may be captured at the cost of profiling precision.

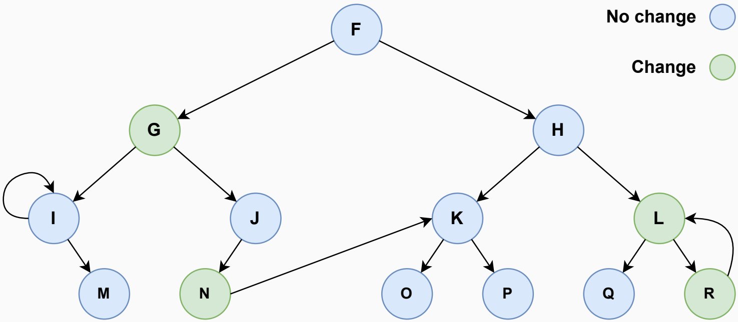
Recency: Diff Tracing

- We identify functions that have *changed* since the last profiling.



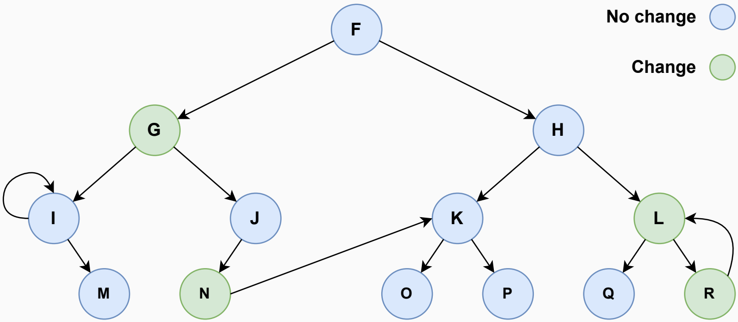
Recency: Diff Tracing

- We identify functions that have *changed* since the last profiling.
  - $\Rightarrow$ Such functions must be profiled.



Recency: Diff Tracing

- We identify functions that have *changed* since the last profiling.
  - ⇒ Such functions must be profiled.
- **Challenge:** how to define and detect a *changed function* with respect to performance metrics?

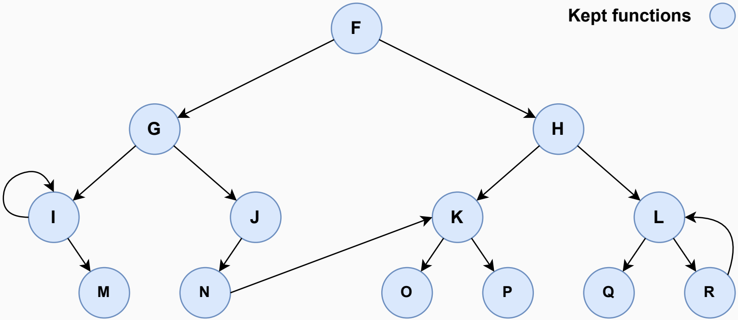


Code Structure: Call Graph Projection

Call Graph Observation

The *number of calls* of a function from a given call site *often grows* with the length that the call stack has upon reaching the call site.

- Backed by experiments on CCSDS and CPython projects (87–96 %).

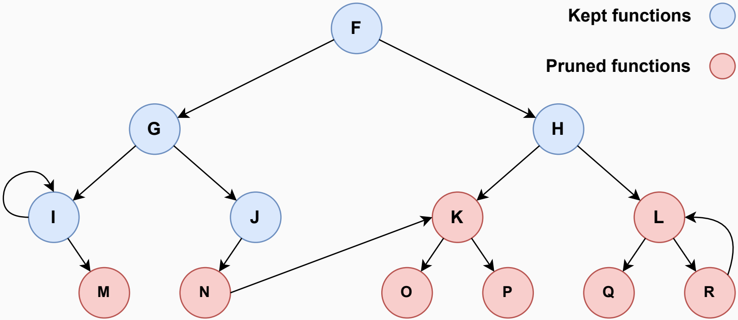


Code Structure: Call Graph Projection

Call Graph Observation

The *number of calls* of a function from a given call site *often grows* with the length that the call stack has upon reaching the call site.

- Backed by experiments on CCSDS and CPython projects (87–96 %).
  - We do not profile functions below certain Call Graph depth.

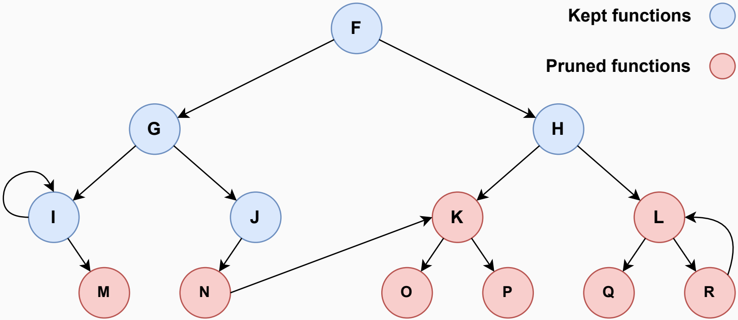


Code Structure: Call Graph Projection

Call Graph Observation

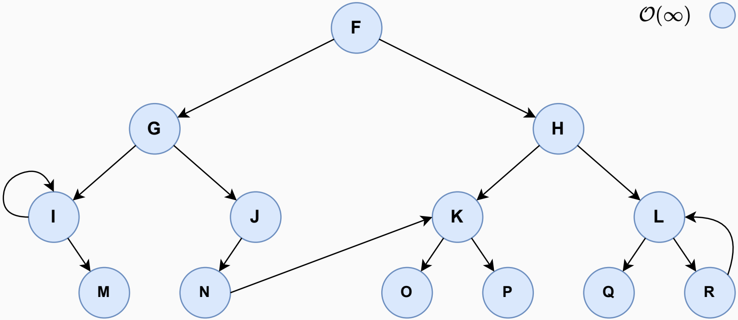
The *number of calls* of a function from a given call site *often grows* with the length that the call stack has upon reaching the call site.

- Backed by experiments on CCSDS and CPython projects (87–96 %).
  - We do not profile functions below certain Call Graph depth.
- **Challenge:** how to define the *Call Graph Depth* with respect to the *Call Graph Observation*?



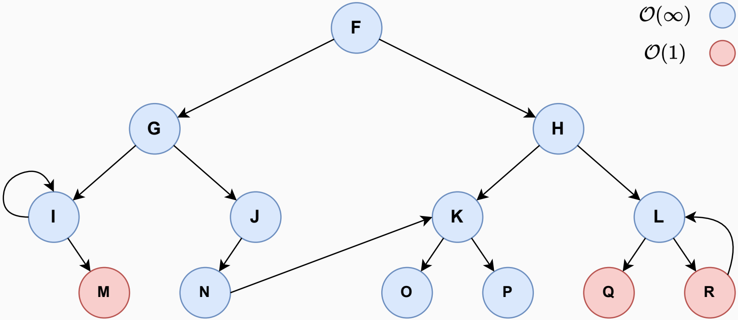
Expected Performance: Performance Baseline

- *Static:*
- *Dynamic:*



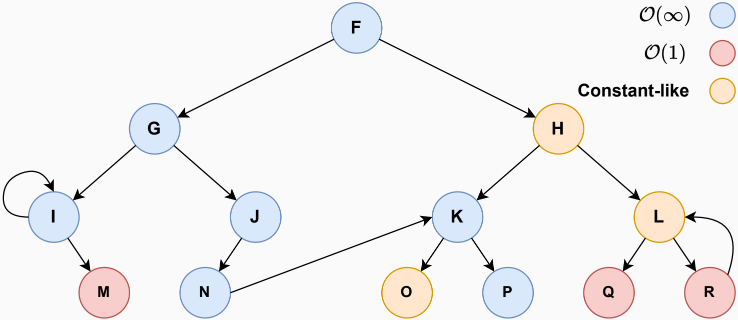
Expected Performance: Performance Baseline

- *Static:* We do not profile functions below certain complexity.
- *Dynamic:*



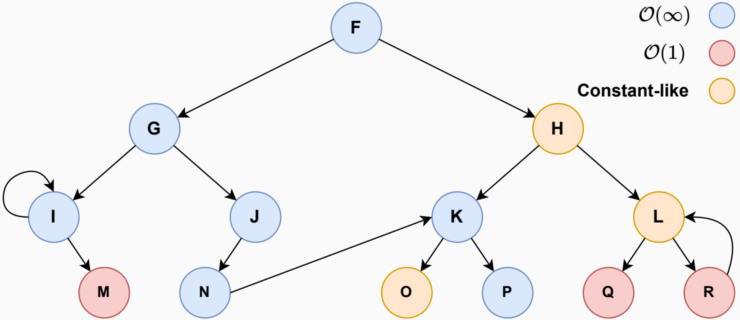
Expected Performance: Performance Baseline

- *Static:* We do not profile functions below certain complexity.
- *Dynamic:* We do not profile functions with constant-like behavior.



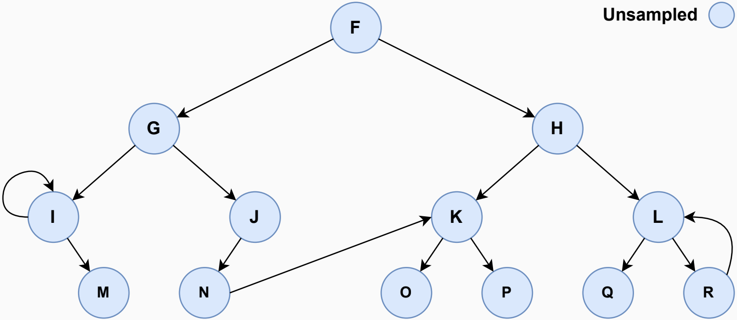
Expected Performance: Performance Baseline

- *Static:* We do not profile functions below certain complexity.
- *Dynamic:* We do not profile functions with constant-like behavior.
- **Challenge:** How to compute *function complexity* and identify *constant-like behavior*?



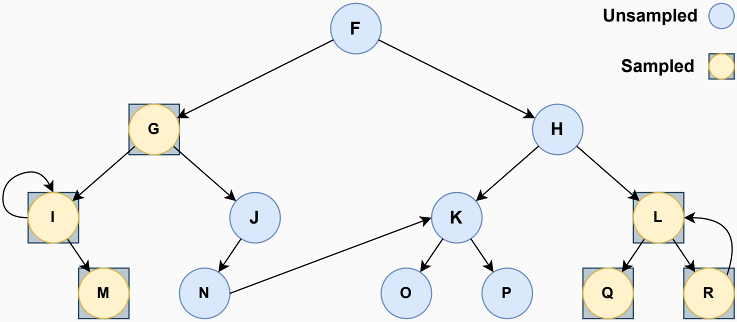
Profiling Process Refining: Sampling Control

- Only a subset of performance data are necessary for sufficiently precise models.



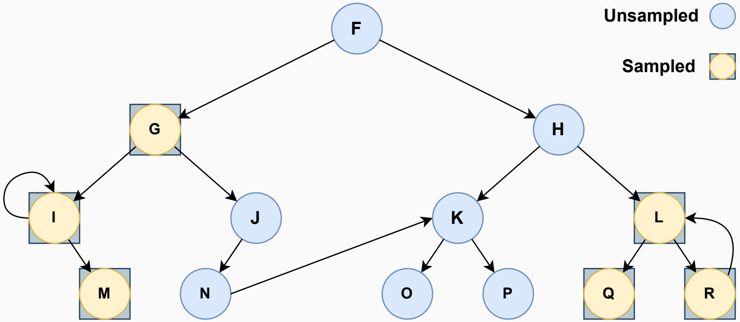
Profiling Process Refining: Sampling Control

- Only a subset of performance data are necessary for sufficiently precise models.
  - ⇒ We record only every Nth function call.



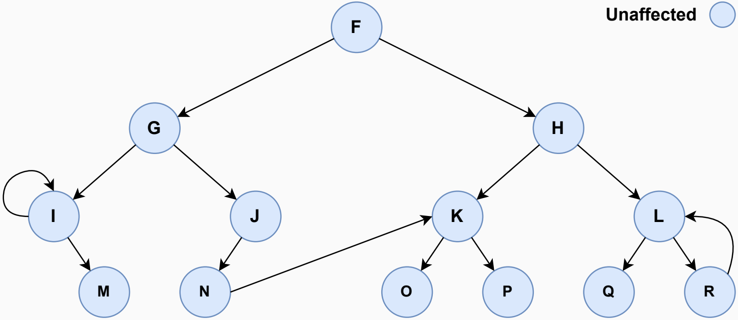
Profiling Process Refining: Sampling Control

- Only a subset of performance data are necessary for sufficiently precise models.
  $\Rightarrow$ We record only every Nth function call.
- **Challenge:** how to identify suitable *functions to sample* and estimate the N?



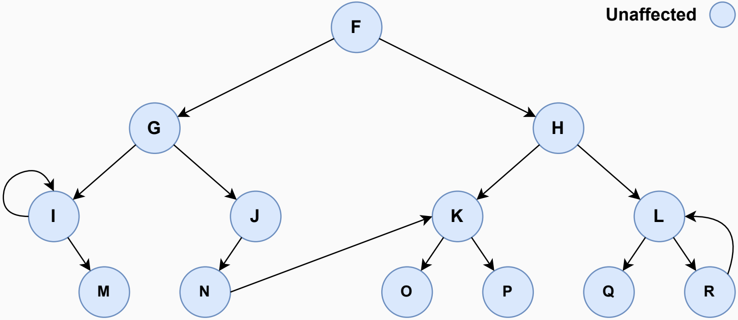
Optimization Pipelines

- Key idea: **combine** the aforementioned approaches.



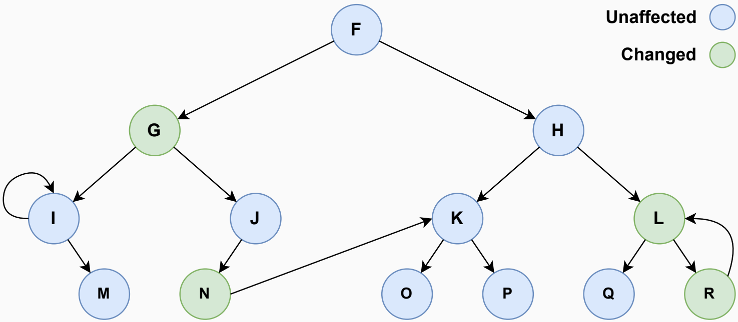
Optimization Pipelines

- Key idea: **combine** the aforementioned approaches.
  - Each technique may be enabled/disabled.
  - Each technique has its own set of parameters guiding the *optimization strength*.



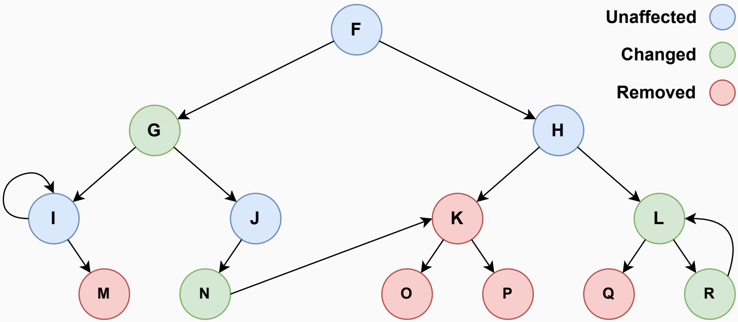
Optimization Pipelines

- Key idea: **combine** the aforementioned approaches.
  - Each technique may be enabled/disabled.
  - Each technique has its own set of parameters guiding the *optimization strength*.



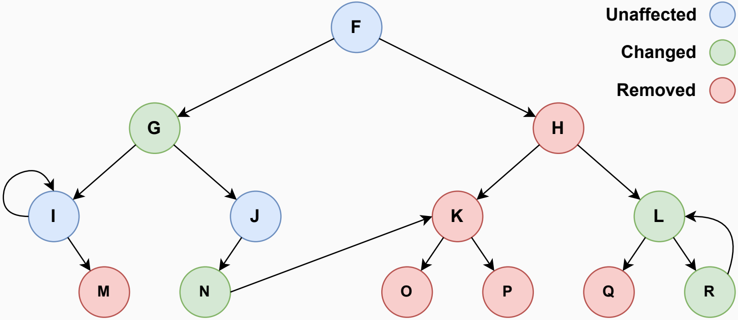
Optimization Pipelines

- Key idea: **combine** the aforementioned approaches.
  - Each technique may be enabled/disabled.
  - Each technique has its own set of parameters guiding the *optimization strength*.



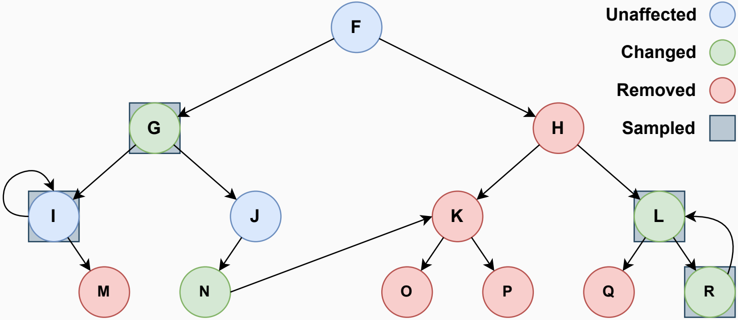
Optimization Pipelines

- Key idea: **combine** the aforementioned approaches.
  - Each technique may be enabled/disabled.
  - Each technique has its own set of parameters guiding the *optimization strength*.



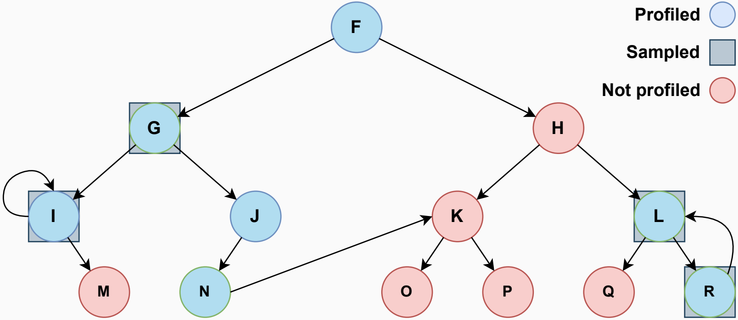
Optimization Pipelines

- Key idea: **combine** the aforementioned approaches.
  - Each technique may be enabled/disabled.
  - Each technique has its own set of parameters guiding the *optimization strength*.



Optimization Pipelines

- Key idea: **combine** the aforementioned approaches.
  - Each technique may be enabled/disabled.
  - Each technique has its own set of parameters guiding the *optimization strength*.



Preliminary Experimental Evaluation

- So far, we have conducted two case studies (more in progress):
  1. (RQ1) *How significant is the impact of the individual optimizations on the profiling process?*
  2. (RQ2) *How significant is the impact of the optimisation pipelines for different degrees of strength on the profiling process?*

Preliminary Experimental Evaluation

- So far, we have conducted two case studies (more in progress):
  1. (RQ1) *How significant is the impact of the individual optimizations on the profiling process?*
  2. (RQ2) *How significant is the impact of the optimisation pipelines for different degrees of strength on the profiling process?*
  - **Optimization strength:** 10 %, 25 %, 50 %, 75 %, 90 %.

Preliminary Experimental Evaluation

- So far, we have conducted two case studies (more in progress):
  1. (RQ1) *How significant is the impact of the individual optimizations on the profiling process?*
  2. (RQ2) *How significant is the impact of the optimisation pipelines for different degrees of strength on the profiling process?*
  - **Optimization strength:** 10 %, 25 %, 50 %, 75 %, 90 %.

	open122	**CPython**	**Gedit**	**emacs**	**vim**
LoC	10,000	500,000	35,000	400,000	480,000
$\lvert\mathcal{F}_P\rvert$	83	1,883/1,227	569	1,826	4,382
branch	2984883	3.8/2.7	3-36	2.71	8.2
$\texttt{runs}_{opt}$	5/10	0/1	2/3	2/3	2/3
$\texttt{runs}_{\pi}$	5/10	0/1	5/5	5/5	5/5

Results: CPython

Python3

Pipeline	Total [s]	Data [MiB]	$\|\mathcal{F}_P\|$	$Cov_{\mathcal{H}}$ [%]	$\delta_{\mathcal{H}}$	$U_{\mathcal{F}_P}$ [%]	$O_{\mathcal{F}_P}$ [%]
no-opt	39,304.19	164,022.73	53,360.00				
π_{10}	14,514.99	56,907.35	39,981.00	40.00	1.00	2.02	3.38
π_{25}	3,847.49	11,482.11	28,741.00	20.00	1.00	2.56	5.59
π_{50}	3,425.64	9,339.42	26,826.00	20.00	1.00	2.79	6.34
π_{75}	2,647.76	5,602.77	24,419.00	20.00	1.00	3.55	6.43
π_{90}	1,683.27	1,572.93	5,471.00	20.00	1.00	0.00	3.85
no-prof	577.58						

Python2

Pipeline	Total [s]	Data [MiB]	$\|\mathcal{F}_P\|$	$Cov_{\mathcal{H}}$ [%]	$\delta_{\mathcal{H}}$	$U_{\mathcal{F}_P}$ [%]	$O_{\mathcal{F}_P}$ [%]
no-opt	43,568.59	171,842.30	34,356.00				
π_{10}	20,704.52	73,768.39	25,432.00	50.00	1.00	2.42	3.57
π_{25}	11,421.38	37,963.51	18,833.00	40.00	1.00	3.41	4.63
π_{50}	8,674.45	26,793.04	15,300.00	40.00	1.00	2.96	4.77
π_{75}	5,270.95	12,825.88	6,969.00	30.00	1.00	2.74	0.00
π_{90}	3,345.23	4,800.91	3,180.00	20.00	1.00	0.00	0.00
no-prof	527.19						

Conclusion

Conclusion

- **Perun** = Complex Performance Analysis and Testing Solution.

Conclusion

- **Perun** = Complex Performance Analysis and Testing Solution.
  - **Integrates** VCS.
  - **Collects** performance data.
  - **Derives** performance models.
  - **Detects** performance changes.
  - **Visualises** performance.

Conclusion

- **Perun** = Complex Performance Analysis and Testing Solution.
  - **Integrates** VCS.
  - **Collects** performance data.
  - **Derives** performance models.
  - **Detects** performance changes.
  - **Visualises** performance.

  ⇒ Not just mere profiling!

Conclusion

- **Perun** = Complex Performance Analysis and Testing Solution.
  - **Integrates** VCS.
  - **Collects** performance data.
  - **Derives** performance models.
  - **Detects** performance changes.
  - **Visualises** performance.

  $\Rightarrow$ Not just mere profiling!
- We believe **profiling efficiency can be significantly improved**.
  - Reuse of historic profiling data when possible.
  - General profiling optimizations and their combinations.

Conclusion

- **Perun** = Complex Performance Analysis and Testing Solution.
  - **Integrates** VCS.
  - **Collects** performance data.
  - **Derives** performance models.
  - **Detects** performance changes.
  - **Visualises** performance.

  $\Rightarrow$ Not just mere profiling!
- We believe **profiling efficiency can be significantly improved**.
  - Reuse of historic profiling data when possible.
  - General profiling optimizations and their combinations.
- **Ongoing and Future work:**
  - Further improving the **efficiency**, **granularity** and **precision**.
  - Support for more **languages**, performance **metrics**, existing **tools**.

Acknowledgements

- In no particular order:
    - Tomáš Fiedor, Tomáš Vojnar, Adam Rogalewicz, Jan Fiedor, Viktor Malík, Martin Hruška, Hanka Šimková, Peter Močáry, Ondřej Míchal, Vojta Hájek, Vladimír Hucovič, Šimon Stupinský, Matúš Liščinský, Martina Grzybowská, Radim Podola, Petr Müller, Jiří Hladký Jan Zelený, Michal Kotoun, and many more.
- Supported by:
    - Red Hat, Inc.
    - Czech Science Foundation Project 20-07487S
    - Czech Science Foundation Project 23-06506S
    - JCMM PhD Talent Scholarship Programme

A Journey Towards Efficient Profiling

Alpine Verification Meeting 2023

Jiří Pavela

E-mail: ipavela@fit.vutbr.cz
Github: `https://github.com/JiriPavela/`
Perun Github: `https://github.com/Perfexionists/perun/`
Paper Demo VM: `10.5281/zenodo.6783242`

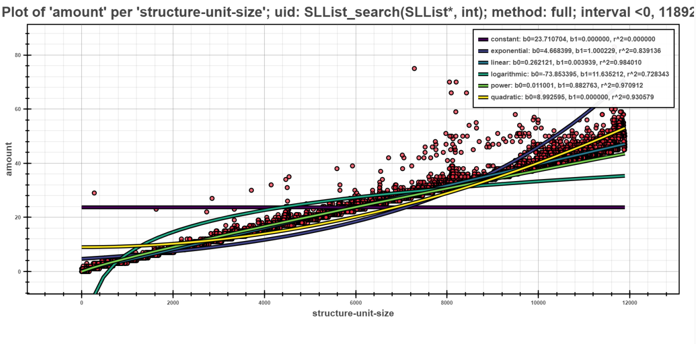
Brno University of Technology, Faculty of Information Technology

Supported by:

Red Hat, Inc.
Czech Science Foundation Project 20-07487S
Czech Science Foundation Project 23-06506S
JCMM PhD Talent Scholarship Programme

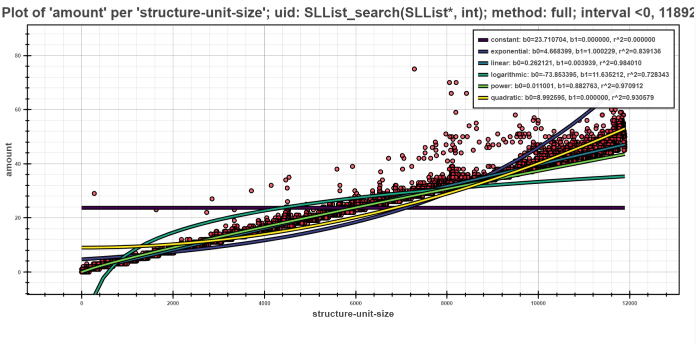
Perun Workflow: Models Example

- Models in Perun are **mathematical functions** of the input size or **statistical summaries** describing the main features of the profile.



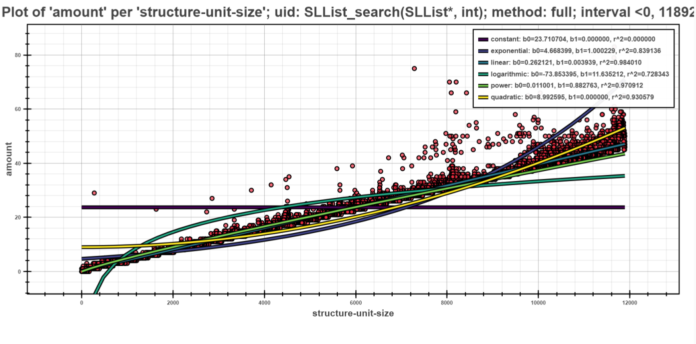
Perun Workflow: Models Example

- Models in Perun are **mathematical functions** of the input size or **statistical summaries** describing the main features of the profile.
- For example, in regression analysis:



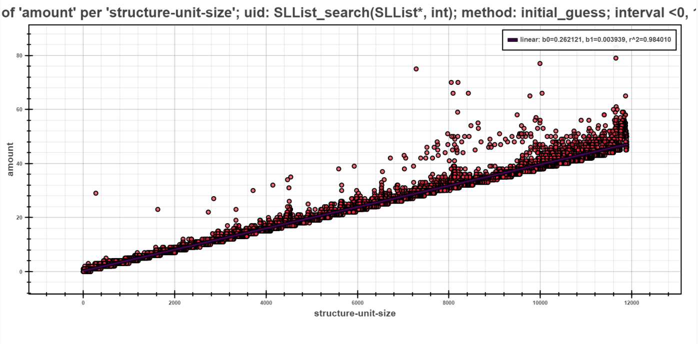
Perun Workflow: Models Example

- Models in Perun are **mathematical functions** of the input size or **statistical summaries** describing the main features of the profile.
- For example, in regression analysis:
  - the curve is described using **function** $y = b_1 f(x) + b_0$,



Perun Workflow: Models Example

- Models in Perun are **mathematical functions** of the input size or **statistical summaries** describing the main features of the profile.
- For example, in regression analysis:
  - the curve is described using **function** $y = b_1 f(x) + b_0$,
  - and the model quality as **coefficient of determination** R^2.



Perun Workflow: Detection Example

- Multiple **detection algorithms** are implemented in Perun:
  - *Best Model Order Equality*
  - *Integral Comparison*
  - ...
  - *Exclusive-Time Outliers*

Perun Workflow: Detection Example

- Multiple **detection algorithms** are implemented in Perun:
  - *Best Model Order Equality*
  - *Integral Comparison*
  - ...
  - *Exclusive-Time Outliers*

Perun Workflow: Detection Example

- Multiple **detection algorithms** are implemented in Perun:
  - *Best Model Order Equality*
  - *Integral Comparison*
  - ...
  - *Exclusive-Time Outliers*
- **Exclusive-Time Outliers:**
  - Based on per-function comparison of exclusive-time **deltas** Δ.
    - *Exclusive-time (self)*: time spent exclusively in a function.

Perun Workflow: Detection Example

- Multiple **detection algorithms** are implemented in Perun:
  - *Best Model Order Equality*
  - *Integral Comparison*
  - ...
  - *Exclusive-Time Outliers*
- **Exclusive-Time Outliers:**
  - Based on per-function comparison of exclusive-time **deltas** Δ.
    - *Exclusive-time (self)*: time spent exclusively in a function.
  - A hierarchy of outlier detection methods determines the **severity**:
    - Modified Z-score $y_i = \frac{x_i - \tilde{X}}{k \cdot median(|x_i - \tilde{X}|)}$

Perun Workflow: Detection Example

- Multiple **detection algorithms** are implemented in Perun:
  - *Best Model Order Equality*
  - *Integral Comparison*
  - ...
  - *Exclusive-Time Outliers*

- **Exclusive-Time Outliers:**
  - Based on per-function comparison of exclusive-time **deltas** Δ.
    - *Exclusive-time (self)*: time spent exclusively in a function.
  - A hierarchy of outlier detection methods determines the **severity**:
    - Modified Z-score $y_i = \frac{x_i - \tilde{X}}{k \cdot median(|x_i - \tilde{X}|)}$
    - IQR multiple $Q_1 - k \cdot IQR < x < Q_3 + k \cdot IQR$

Perun Workflow: Detection Example

- Multiple **detection algorithms** are implemented in Perun:
  - *Best Model Order Equality*
  - *Integral Comparison*
  - ...
  - *Exclusive-Time Outliers*
- **Exclusive-Time Outliers:**
  - Based on per-function comparison of exclusive-time **deltas** Δ.
    - *Exclusive-time (self)*: time spent exclusively in a function.
  - A hierarchy of outlier detection methods determines the **severity**:
    - Modified Z-score $\quad y_i = \frac{x_i - \tilde{X}}{k \cdot median(|x_i - \tilde{X}|)}$
    - IQR multiple $\quad Q_1 - k \cdot IQR < x < Q_3 + k \cdot IQR$
    - Standard deviation multiple $\quad \overline{X} - k \cdot \sigma < x < \overline{X} - k \cdot \sigma$