Efficient Techniques for Program Performance Analysis

Jiří Pavela Adam Rogalewicz

Brno University of Technology, Faculty of Information Technology

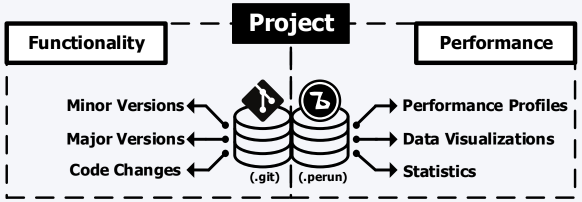
Motivation

- Program Performance Analysis is vital for **detection** and **localization** of performance bugs within a program.
- Similarly to functional program testing, performance should ideally be analyzed after every major change of the source code, e.g., `commits` in the `Git` terminology.
- Large codebases with **hundreds of thousands of LoC** (Lines of Code), thousands of functions and complicated, time-consuming compilation process **pose a challenge** to continuous automated performance analysis.
- Specifically, the problem lies in either **(a)** enormous time and memory overhead when all instrumentation locations are profiled or **(b)** manually selecting the sufficient set of locations to ensure fast but crude analysis.

Goal of the Thesis

- Automated selection of the appropriate subset of available instrumentation points $\mathcal{IP}_i \subseteq \mathcal{IP}$ in order to **diminish the overhead** and avoid the need for manual filtering while still keeping sufficient **precision and granularity** of the profiling data.

Perun Framework

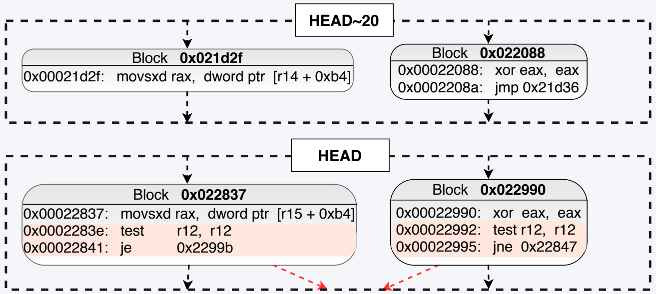


- `Perun` is an open-source lightweight Performance Version System that **manages, analyzes and visualizes** performance profiles of a program.
- Profiling data are obtained via `Collectors`: modules built upon various instrumentation frameworks (e.g. *SystemTap* or *eBPF*) that can **dynamically inject** instrumentation probes to executable files. Thus, there is no need for program recompilation or project source code!

Proposed Optimization Techniques

We designed novel optimization techniques based on **combination of static and dynamic analyses** in the areas of (1) *Semantic information*, (2) *Syntactic information*, (3) *Recency* and (4) *Profiling process*. We propose the following methods:

1. **Static Baseline** exploits existing *formal static analysis of resource bounds* to filter out non-complex instrumentation points (e.g., functions with constant complexity).
2. **Dynamic Baseline** leverages metrics gathered from *previous profiling runs* to filter non-complex instrumentation points that cause the most profiling overhead.
3. **Call Graph Shaping** is a family of *static analysis methods* that prune instrumented functions based on the structure of a program Call Graph.
4. **Diff Tracing** exploits the Call Graph, Control Flow Graph and deep integration of Perun and VCS to instrument only **recently changed code**.



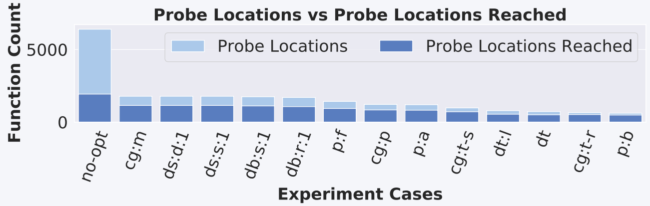
```
Block 0x021d2f
0x00021d2f:  movsxd rax,  dword ptr  [r14 + 0xb4]

Block 0x022088
0x00022088:  xor eax,  eax
0x0002208a:  jmp 0x21d36

Block 0x022837
0x00022837:  movsxd rax,  dword ptr  [r15 + 0xb4]
0x0002283e:  test        r12,  r12
0x00022841:  je          0x2299b

Block 0x022990
0x00022990:  xor  eax,  eax
0x00022992:  test r12,  r12
0x00022995:  jne  0x22847
```

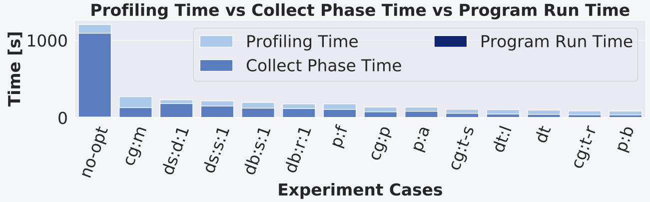
6. **Dynamic Sampling** estimates and iteratively refines how often the probes generate performance data.
7. **Timed Sampling** is similar to Dynamic Sampling, however, the sampling is triggered in specific time intervals.
8. **Dynamic Probing** measures the overhead incurred by the probes at **run-time** and dynamically disables or enables them during the profiling.

Furthermore, we designed the methods such that they can be arbitrarily combined together into `pipelines`. Three pipelines were pre-configured for a better user-experience: (1) *Basic*, (2) *Advanced* and (3) *Full*.

Experimental Evaluation

Total of 37 experiment cases per 4 different configurations and 2 projects (`CPython 3.8` interpreter: 500000+ `C/C++` LoC, $\approx$ 6400 functions, `CCSDS 122.0` image compression standard: 10000+ `C` LoC, 164 functions) were performed. Following barplots show only a small subset of `CPython` measured data corresponding to certain optimization methods.





Conclusion

We were able to achieve a significant degree of optimization for most of the identified `Optimization Criteria` (up to hundreds of % of profiling speedup and data size reduction while **not severely compromising precision**). Thus, we are now able to analyze even projects that would otherwise be **infeasible** due to the enormous time or memory overhead.

In the future, we plan to leverage the proposed methods to **(a)** optimize other dynamic profiling tools (e.g., `Valgrind`) and **(b)** efficiently **hunt performance bugs and degradations** in various real-world projects of all sizes.

The presented work was supported by the ECSEL AQUAS: "Aggregated Quality Assurance for Systems" and Czech Science Foundation SNAPPY: "Scalable Techniques for Analysis of Complex Properties of Computer Systems" projects.