COST AWARE TRACING OF SOFTWARE SYSTEMS

Amir Haghshenas
Naser Ezzati-Jivan
Michel Dagenais
Fall 2023

AGENDA

- Review of first track
- Result and Analysis
- Second track problem statement
- Second track progress
- Next steps

01 - TRACING IS COSTLY

Tracing and logging a software **will** introduce unaccounted costs on the system such as performance overhead or large memory and disk consumption

02 - OBJECTIVE & COST(S)

Every tracing configuration is for a goal and not all of them target similar software regions. Cost of tracing can also be categorized in multiple groups and sub-groups.

03 - COST AWARE FRAMEWORK

Balancing the quality and quantity of the tracing configuration can be done using a cost-aware tracing framework by solving an optimization problem.

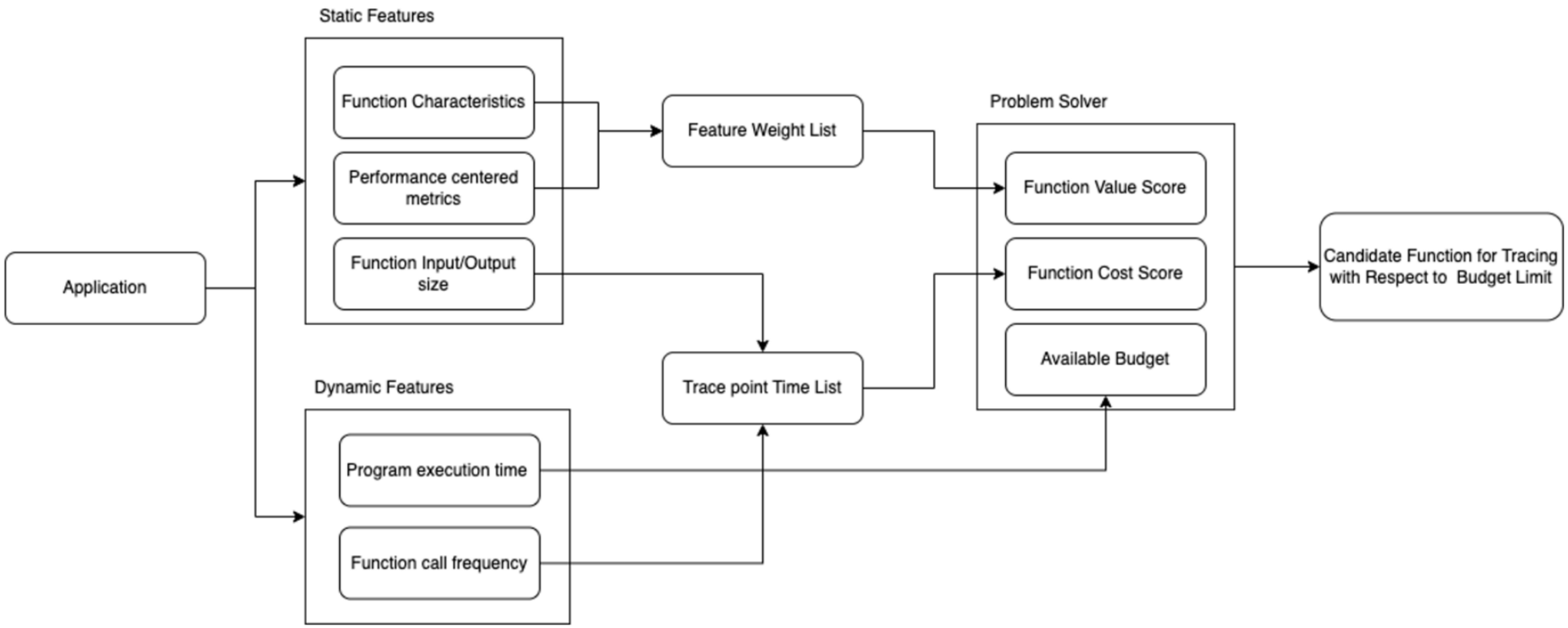
COST FUNCTIONS

- EXECUTION TIME OVERHEAD
- MEMORY OVERHEAD
- DISK OVERHEAD
- CODE SIZE
- OPPORTUNITY COST
- MAINTENANCE CSOT
- ANALYSIS COST
- EGNERGY COST

COST AWARE TRACING FRAMEWORK

1. For selected tracing objective (performance monitoring), identify metrics to calculate effectiveness (value)
2. For selected cost function (execution time), identify metrics to calculate the cost
3. Combine cost and value metrics into one integer in order to map the problem to Knapsack problem
4. Solve the optimization problem to minimize cost and maximize effectiveness
5. Suggest selected functions as initial tracing configuration

PROTOTYPE WORK FLOW



USE CASE

A software with large number of functions as target application for tracing.

Not all functions should be enabled for efficient performance monitoring due to heavy execution time overhead.

Using the cost-aware framework we can select the top functions with uncertain execution time to be initially enabled for tracing.

SECOND TRACK

In this track, we are focusing a new cost function for tracing and logging which is the analysis cost. The goal is to analyze the history of logging in large open source applications, to find the logging effectiveness in the bug fixing process.

PROBLEM

Software systems are usually heavily logged for different purposes. But not all the generated logs are useful. How to reduce the number of logs and keep the most relevant ones.

SOLUTION

Study the historical data of the commits for large open-source applications to understand the relation between logging statements and bug-fixing commits.

CONTRIBUTION

Learn from changed functions and their logging statements and suggest logging decision for new functions.

SECOND TRACK PROGRESS

1. Select popular open-source application in Java and Python
2. Get all the commits from GitHub
3. Filter commits that fixed a bug using a combination of selected keywords
4. Identify the changed file and functions for each commit
5. Extract static metrics from both log statements and functions
6. Using different methods, learn the relation between log statement and bug fixing commit

LOG FEATURE LIST

- LOG PRESENCE
- LOG MESSAGE LENGTH
- LOG LEVEL
- PLACEMENT OF LOG
- CONTEXTUAL INFORMATION
- COMMENTS NEAR LOG
- LOG FREQUENCY

FUNCTION FEATURE LIST

- FUNCTION COMPLEXITY
- CONDITIONAL STATEMENT
- INPUT VALIDATION
- MULTI THREADING
- LOOP STRUCTURE
- SEMANTIC FEATURES

USE CASE

The target application generates large log files due to many log statements that are placed in the code.

Finding the most relevant log lines from the whole log lines generated will take so much effort (analysis cost).

Using the result of the second track, we will be able to reduce the number of logs that are generated while keeping the most relevant log lines.

NEXT STEPS

Learn from the extracted features and suggest log statements that can be removed from the application in order to reduce the analysis cost.

For a new and unknown function, only extracting the syntactic and semantic features should help us determine if the function should be logged on not.

THANK YOU

amir.haghshenas@polymtl.ca