Automated Phase Detection for Adaptive Tracing of Software Systems

Madeline Janecek
mj17th@brocku.ca
Supervisor: Naser Ezzati-Jivan

Department of Computer Science
Brock University

Problem Statement: Adaptive Tracing

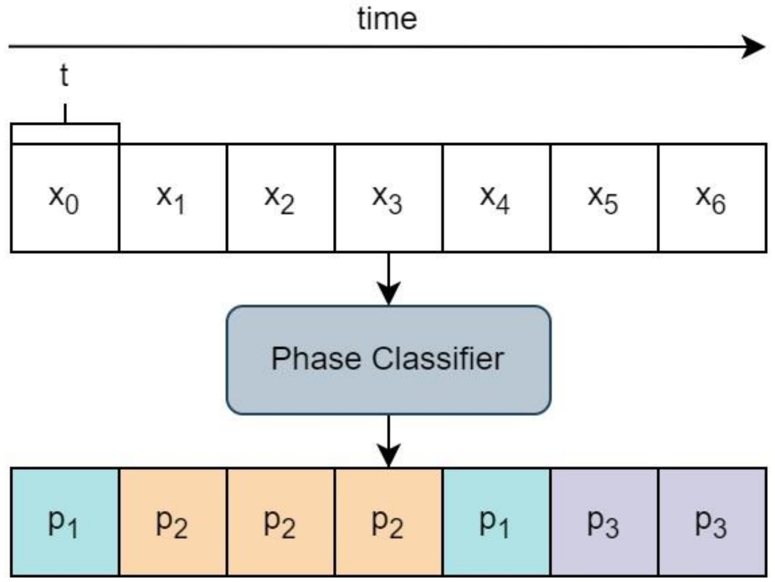
Collecting all possible events:

- Results in extremely large amounts of data
- Introduces unnecessary runtime overhead
- Increases the complexity of the data's analysis

How can we intelligently enable events to skip over redundant information, and only record the most novel aspects of a software's execution?

Application Phases

- *Application phases* are intervals within a software's execution that exhibit similar behaviours and resource requirements
- Phase-based approaches have been used in the past to enhance various tasks, like just-in-time compilation, thread-to-core assignment, resource allocation, and so forth



By identifying a software system's common phases, we can conversely identify its most uncommon behaviours and learn to predict them

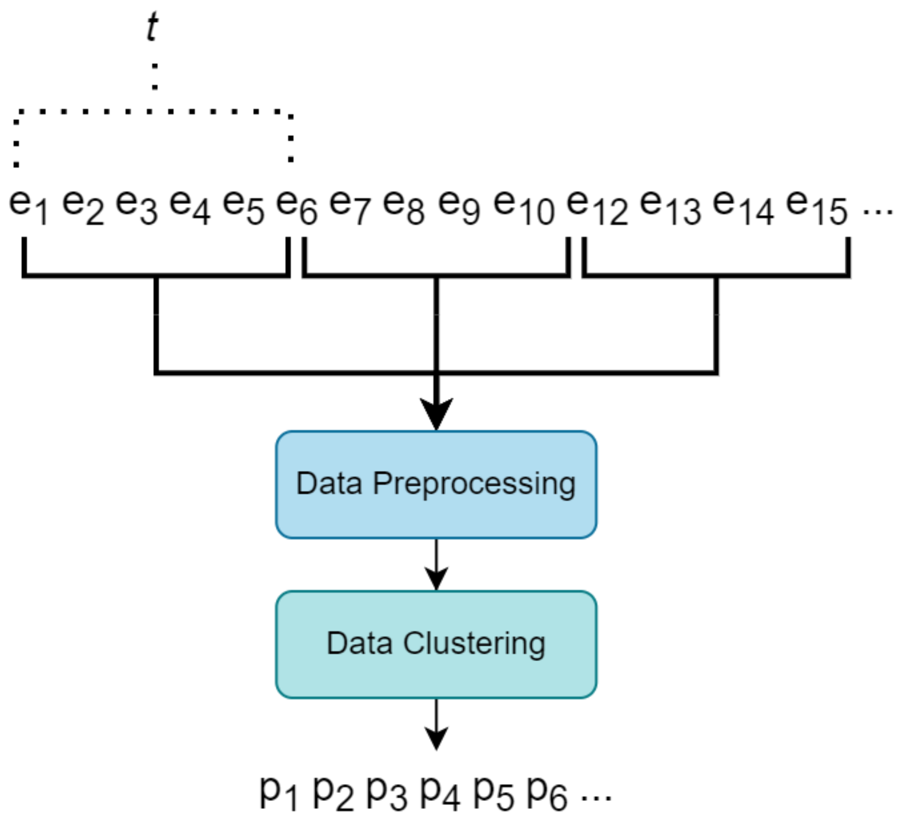
Method Overview

We propose a phase-based adaptive tracing solution that follows three main stages:

1) Phase Identification
   - Employ clustering techniques to identify the software system’s main execution phases, as well as points of interest
2) Phase Prediction
   - Train a prediction model to predict what phase will occur next, and enabled more extensive tracing if it anticipates a point of interest
3) Model Adaptation
   - Check if the prediction results align with what actually happens, and if necessary, update the model

1) Phase Identification

- Collect events using LTTng
  - It is assumed that most events are exhibiting normal behaviour
- Partition events into non-overlapping windows, where each window covers *t* amount of time
- These windows are processed to:
  1. Identify the software's phases, including outlying behaviours of interest
  2. Represent the execution as a sequence of phases for the Phase Prediction stage



1) Phase Identification – Data Preprocessing

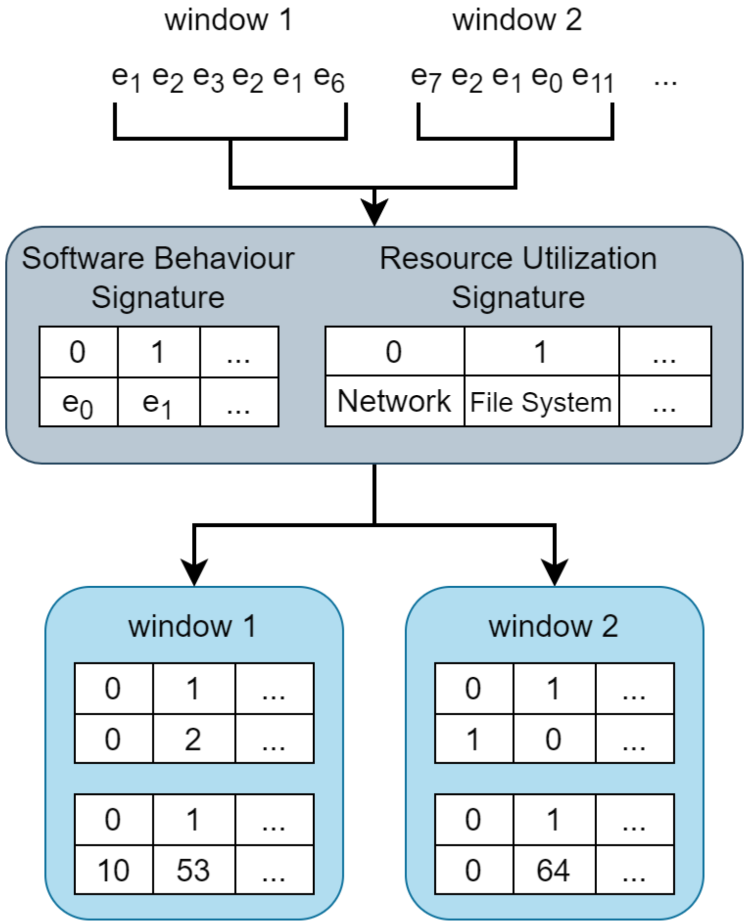
For each window, we generate two different perspectives of the software's execution:

1) Software Behaviour Signatures

- How many times each system call was invoked during the execution window
  - Under identical circumstances, identical code will likely evoke a similar set of kernel events
  - Used infer what code was running during the window

2) Resource Utilization Signatures

- For how much time the thread used different resources to complete its task
  - Used to infer the software's performance, workload, etc.



1) Phase Identification – Data Clustering

We use the two vector formats to group together windows with similar execution behaviours and resource requirements

- In other words, each cluster can be thought of as an application phase

We use Self-Organizing Maps (SOM) in a two-stage clustering approach

- SOMs are a type of artificial neural network (ANN) that rely on competitive learning to gradually learn the underlying data distribution in an unsupervised manner
- SOMs provide several advantages:
  - Robust to noise
  - Faster than other clustering algorithms (e.g. DBSCAN) when given large datasets
  - Capable of identifying clusters with varying shapes and densities

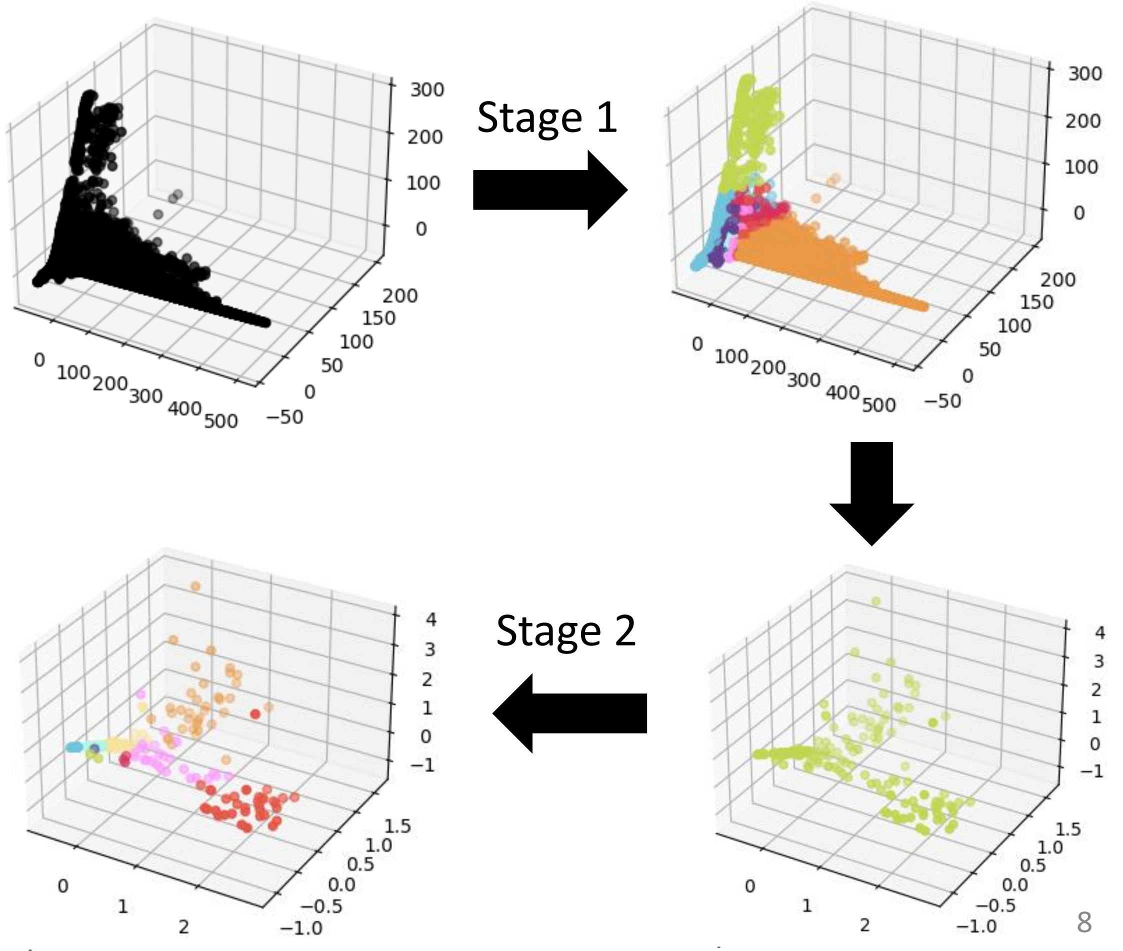
1) Phase Identification – Data Clustering

The data clustering procedure consists of:

Stage 1) Identify windows performing similar tasks by clustering the Software Behaviour Signatures

Stage 2) Identify software phases by taking each cluster from stage 1 and clustering its windows' Resource Utilization Signatures

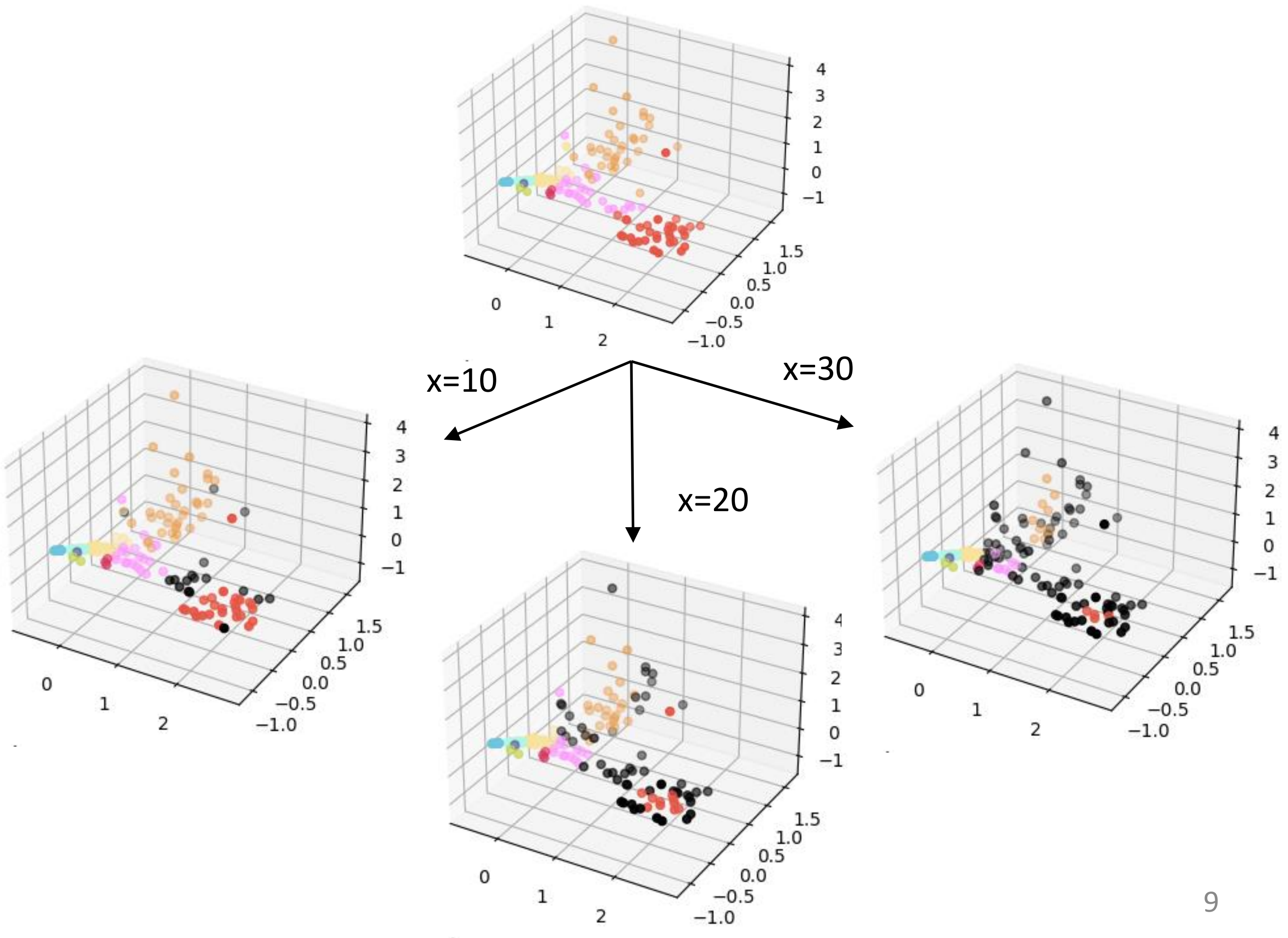
The clusters from stage 2 make up the software system's defined phases



1) Phase Identification – Outlier Identification

Outlier windows, which are the desired target for tracing, are identified in two ways:

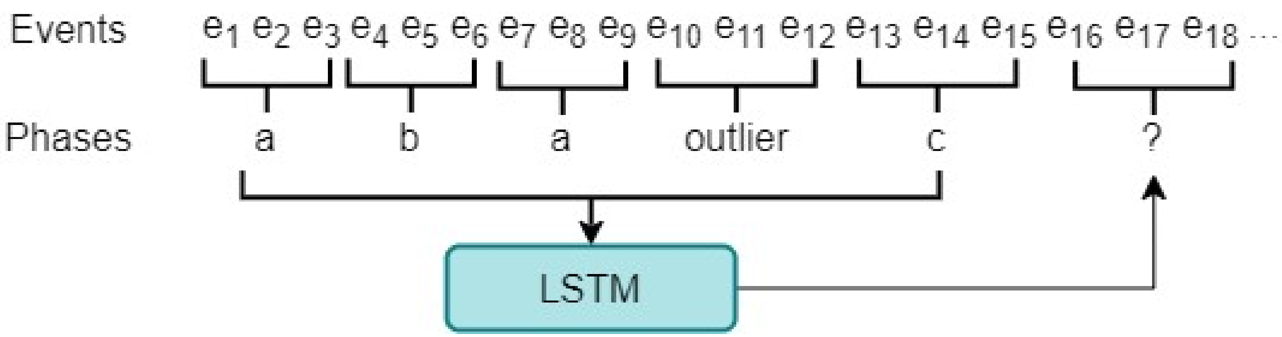
1. Clusters that are too small (e.g. < 0.1% of the data) are marked as outliers
2. The *x*% most outlying windows from each second stage clustering are marked as outliers
   - Higher values of *x* lead to more detailed traces



2) Phase Prediction

The phase sequences are given to an LSTM model, which is trained to predict whether an upcoming phase will be an outlier

- If an outlying window is anticipated, more tracepoints are enabled
- If a window within a phase is anticipated, the additional tracepoints are disabled



3) Model Adaption

To account for dynamic behaviours, we constantly compare a window's predicted label (outlier, or one of the phases) with its assigned label from the SOM models

We define three possible outcomes:

1. True Prediction: The window's predicted label matches its assigned label
2. Incorrect: The window is predicted to be an outlier, but it is assigned to a phase by the SOM models
3. Unknown: The window is predicted to be in a phase, but it is determined to be an outlier by the SOM models

If the number of incorrect windows surpasses a threshold, the prediction model must be retrained

If the number of unknown windows surpasses a threshold, the clusters must be redefined using updated data

Preliminary Results

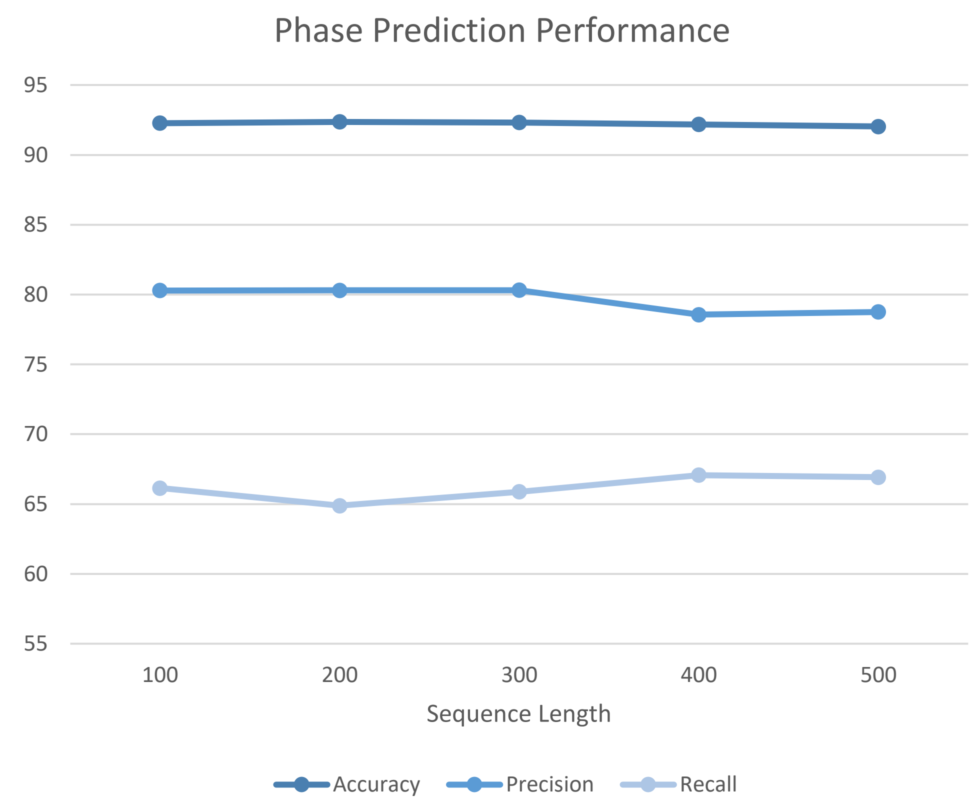
Phase Prediction

Experimental Setup:

- System calls collected with LTTng on an Apache Web Server

When it comes to determining if a window will be an outlier, the LSTM model achieves a:

- 92.035-92.362% accuracy
- 78.568-80.308% precision
- 64.887-67.074% recall



Future Work

- Method is showing promising results for adaptive tracing
- We are actively looking for more specific use cases to further test its potential

Selected References

[1] T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue, “Padla: A dynamic log level adapter using online phase detection,” 05 2019, pp. 135–138.

[2] E. Ates, L. Sturmann, M. Toslali, O. Krieger, R. Megginson, A. K. Coskun, and R. R. Sambasivan, “An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications,” in Proceedings of the ACM Symposium on Cloud Computing, ser. SoCC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 165–170. [Online]. Available: https://doi.org/10.1145/3357223.3362704

[3] M.-C. Chiu and E. Moss, “Run-time program-specific phase prediction for python programs,” in Proceedings of the 15th International Conference on Managed Languages amp; Runtimes, ser. ManLang’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3237009.3237011

[4] M.-C. Chiu, B. Marlin, and E. Moss, “Real-time program-specific phase change detection for java programs,” in Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, ser. PPPJ ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2972206.2972221

[5] E. S. Alcorta and A. Gerstlauer, “Learning-based workload phase classification and prediction using performance monitoring counters,” in 2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD), 2021, pp. 1–6.

[6] E. S. A. Lozano and A. Gerstlauer, “Learning-based phase-aware multi-core cpu workload forecasting,” ACM Trans. Des. Autom. Electron. Syst., vol. 28, no. 2, dec 2022. [Online]. Available: https://doi.org/10.1145/3564929

[7] K. Criswell and T. Adegbija, “A survey of phase classification techniques for characterizing variable application behavior,” IEEE Trans. Parallel Distrib. Syst., vol. 31, no. 1, p. 224–236, jan 2020. [Online].Available: https://doi.org/10.1109/TPDS.2019.2929781

Thank you!

Madeline Janecek
mj17th@brocku.ca
Supervisor: Naser Ezzati-Jivan

Department of Computer Science
Brock University