UNIVERSITY OF CALIFORNIA,
IRVINE

# Visualizing Execution Phases Using Flame Graph

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Information and Computer Science

by

Zichong Li

2021

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

First and foremost, I am extremely grateful to my supervisor, Professor James A. Jones for his continuous support and patience in these two years. He readily agreed to work with me despite my inexperience in his research field. When I encountered difficulties and bottlenecks in my research, he always gave me useful help and suggestions. I also would like to thank Kaj Dreef and Vijay Krishna Palepu for their help with my research. I would like to thank all the authors of the paper *Hierarchical Abstraction of Execution Traces for Program Comprehension*. This project is inspired by their previous work.

I would like to thank my friends Xiaochen Yu, Yang Jiao, and Zhenhan Li for their support in my daily study during these two years. I also thank Wenxuan Guan, Zhen Huang, Yiliu Cai, Yunlong Song, and others for their spiritual encouragement and support.

Finally, I would like to express my gratitude to my parents. Without their tremendous understanding and encouragement, it would be impossible for me to complete my study.

# ABSTRACT OF THE THESIS

Visualizing Execution Phases Using Flame Graph

By

Zichong Li

MASTER OF SCIENCE in Information and Computer Science

University of California, Irvine, 2021

Professor James A. Jones, Chair

Profiling mechanisms are an important dynamic analysis approach for source code learners and maintainers to understand the functions, the dynamic behavior, and the purpose of method calls in the hierarchical abstraction of the program. However, the output provided by program profilers can be verbose, making the mapping between source code and program behavior time-consuming. This paper presents a new approach for processing and visualizing program execution trace to speed up the understanding process, making profilers' output more intuitive to users. This approach first identifies similar execution phases using the tree edit distance (TED) measure and then visualizes the methods and the phases detected based on the flame graph. Compared with other similar studies, our approach reduces the preprocessing time of the profiler's output and presents the execution phases of the software at various levels of abstraction to the user while maintaining the invocation time and the order of the methods in an execution. To verify the effectiveness of this technique, we also conducted case studies on three different Java programs (Jackson, iText, and FingBugs). The results demonstrate that our approach can process the profiler output and detect similar execution phases in a short time. Users can also locate the position of phases in the execution process by finding similar shapes and simple operations in the visualization.

# Chapter 1

# Introduction

Program comprehension is the process of acquiring knowledge about a computer program [35]. With the rapid development of technology and project management, the speed of software updates and iterations is getting faster and faster, which brings new challenges for software development and maintenance personnel to comprehend the programs. To facilitate this process and improve productivity, many strategies of program comprehension have been proposed. Among them, dynamic analysis is an analysis method by analyzing the programs' run-time behavior. Dynamic analysis can help programmers gain a deeper understanding of the program's running flow and method calls, form the mapping between the source code and the program's dynamic behaviors, and facilitate the overall learning process.

Dynamic analysis typically involves instrumenting a program to examine or record certain aspects of its run-time state [4]. However, the trace events of instrumentation-based profiling mechanisms can be verbose, limits the usefulness of the analysis. Besides that, different programmers have different requirements for understanding programs. The raw output provided by those profilers is usually linear and one-dimensional, which is not suitable for programmers at the different cognitive levels to use.

To tackle these problems, researchers have proposed several types of approaches. One method is to reduce the size of the profilers' output by abstraction technique [22, 40, 32]. Those methods summarize and categorize the trace events into structures or phases to help facilitate the comprehension process. Another is to visualize the trace events to make trace events more informative [13, 38]. Those visualizations can usually reveal information that is hard to discover through analyzing profilers' raw output.

Previous research in our lab proposed an advanced approach that combines both methods. It abstracts the execution traces hierarchically to obtain the execution phases of the program under different hierarchies [15]. Based on the original research, this project analyzed its advantages and disadvantages, corrected the existing problems while retaining the advantages, and improved the phases clustering algorithm and the visualization.

By analyzing the experimental results of three case studies, we can conclude that the new approach can find the execution phases under various program hierarchies faster and more accurately. The new visualization method based on the flame graph can also provide more reliable and useful information to users.

## 1.1 Previous Work

The motivation and central idea for this thesis come from the paper by Yang Feng *et al.*: *Hierarchical Abstraction of Execution Traces for Program Comprehension* [15]. We analyzed and summarized the advantages and inadequacies of the approach proposed in this paper and improved the implementation. We will give a more detailed explanation of the previous work in the background chapter.

## 1.2 Motivations

Dynamic analysis could help bridge the cognitive gap between source code and software run-time behavior, which is beneficial for software maintainers and learners to complete their tasks. Dynamic analysis typically comprises the analysis of a system's execution through instrumentation [12]. Developers collect the execution traces produced by the program instrumenters, observe them, and then obtain software running information. However, analyzing the execution traces the developers collected may not be a straightforward task. Literature and papers have pointed out that information overflow is one of the main limitations of analyzing execution traces [17, 12]. The number of execution events that executions produced may be considerably large for a human to understand. Moreover, the execution traces are usually verbose and only contain low-level information, which is hard for the human mind to organize and interpret. Although information overload can be handled by abstraction approaches, we need to be careful while executing those techniques and make trade-offs between precision and quantity of information. The goal of this project is to present a possible workflow to process large amounts of data and present useful information via flame graph visualization. We summarized our motivation for this project as follow:

### 1.2.1 To Process Massive Data Using Phase Detection

Complex software could produce massive data after instrumentation. Reducing the size of execution traces facilitates the understanding process and shortens the data processing time. One abstraction technique for such problem is phase detection [30, 5]. Finding a similar pattern in execution events and grouping them into phases could help developers absorb information more easily. This technique provides developers with the "big picture" of software run-time behavior and ignores the trivial details. Inspired by this kind of approach, our goal is to propose a new phase-grouping mechanism while maintaining other useful information.

### 1.2.2 To Create Multi-level Phase Abstraction

Sufficiently understanding the source code is a key activity for developers. However, "sufficient" can mean different things for developers under different cognitive levels and circumstances. Converting the event traces hierarchically to higher-level and lower-level phases on the basis of needs is beneficial for developers at all cognitive levels.

### 1.2.3 To Present Phases Using Flame Graph

Flame graphs are a recent data representation technique applied to making informative large volumes of information from stack traces listing executed functions of a computer system [17, 9]. Flame graphs were invented by Gregg Brendan in 2011 and used by companies such as Netflix and Google. Although this visualization is mainly for performance analysis and debugging, we believe this new form of representation can also be utilized in the program comprehension field. Our goal is to extend the use of flame graphs and represent the hierarchical phase based on it.

### 1.2.4 To Improve Our Previous Study

This project is the extension of our previous work [15]. We have already proposed a hierarchical abstraction technique to tackle the problems discussed before. However, upon closer analysis, we found that there was much room for improvement in both phase-detection and visualization. We discuss the previous work and its drawbacks in the background chapter.

## 1.3 Thesis Structure

Chapter 7 briefly describes the related work in this project. Chapter 2 contains the background required for a full understanding of the project. This chapter includes a detailed introduction of the paper, tools, and algorithms on which this project is based. Chapters 3 and 4 explain how this new approach detects phases and how the visualization is implemented. Chapter 3 elaborates the source code, as well as the solutions in various cases in detail. Chapter 4 explains the visualization implementation details and the function of the visualization. Chapter 5 devotes to three cases studied that we conducted to verify the effectiveness of our new technique. Chapter 6 discusses the main contributions of the project and how it has improved compared to previous work. Finally, chapter 8 summarizes the thesis and discusses the future work.

# Chapter 2

# Background

This section contains the prerequisite techniques and information for understanding this project. In Section 2.1, we introduced in detail the previous laboratory project on which this project was based. The rest of the sections mainly introduces the algorithms and tools used in this project

## 2.1 Previous Work

The previous paper [15] proposed an automatic approach to present hierarchical abstraction from event traces. The approach also employed Blinky [1] as the instrumenter to produce event traces. Two primary stages remain after retrieving the event traces: data preprocessing and model building. We will give a brief introduction to those stages in this section.

### 2.1.1 Data Preprocessing

The event traces that Blinky outputted contain function-based information. For each method that triggered, Blinky records its (1) method ID, (2) method name, and (3) method call depth. Figure 2.1 shows a possible event trace produced by Blinky.

```
Begin Profiling
ID Name Depth
1 parseFile 0
2 preProcess 1
3 parseASCII 1
5 parseNumber 2
6 parseChar 2
4 parseSpecialCharacter 1
End Profiling
```

Figure 2.1: Event traces

The next step is to generate a preliminary phase tree from the event traces. The preliminary phase tree was detected based on the increases and decreases of call depth. The rules of determining a phase are straight-forward: they traversed all the events, and (1) events with higher call depth is the children of the most recent event with lower call depth, (2) a phase is detected whenever the call depth of the next event is lower or equal to the current event. The phases detected for the event traces above are shown in Figure 2.2.

### 2.1.2 Model Building

After all the preliminary phase tree is detected, a series of actions are performed. The primary goal of model building is to find similar phases and combine them to reduce the total number of phases. Model-building stage will produce a hierarchical execution abstraction for one program execution. This stage consists of several substeps, each of which is explained below.

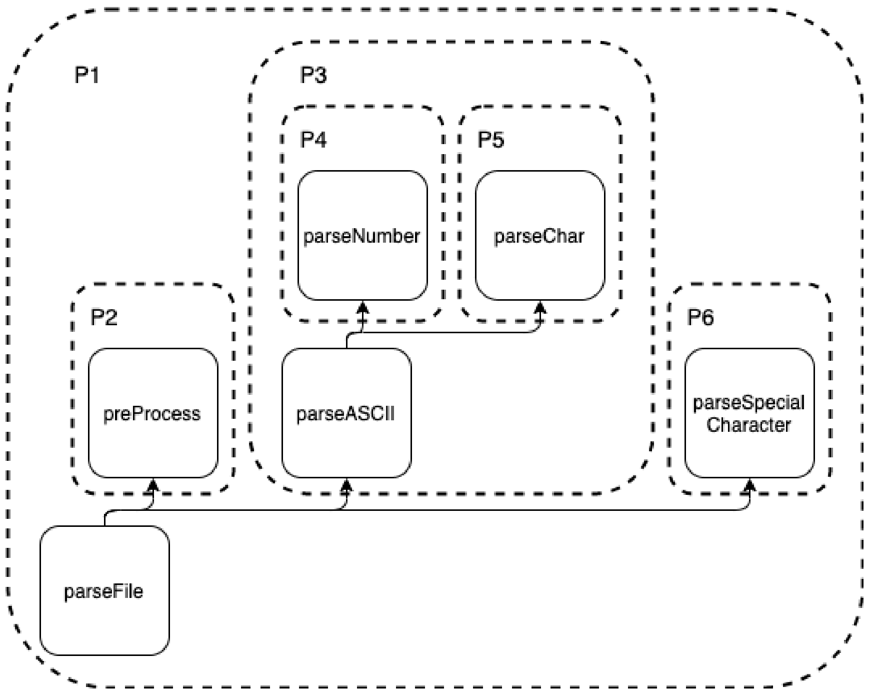


Figure 2.2: Phased detected for the example event traces

**Phase clustering**

They assigned a global key for each preliminary phase tree. The global key is comprised of (1) the depth of the root method in each phase, (2) all the invoked methods in this preliminary phase tree, and (3) the order of the invocations. A method will be included only once even if they were invoked multiple times in this tree. For example, for the preliminary phase tree in Figure 2.3, the global key for *P2*, *P3* and *P6* in the second layer will be "*2: b, e, f*", "*2: g, c, d*", "*2: b, e, k*".

They identified the similar preliminary phase tree on each level by comparing their global key and applied agglomerative hierarchical clustering (AHC) [36] on the unique phases set. To determine the similar tree, they first calculated the *Jaccard Distance* between two trees.

$$JD(P_i, P_j) = 1 - \frac{|M_i \cap M_j|}{|M_i \cup M_j|}$$

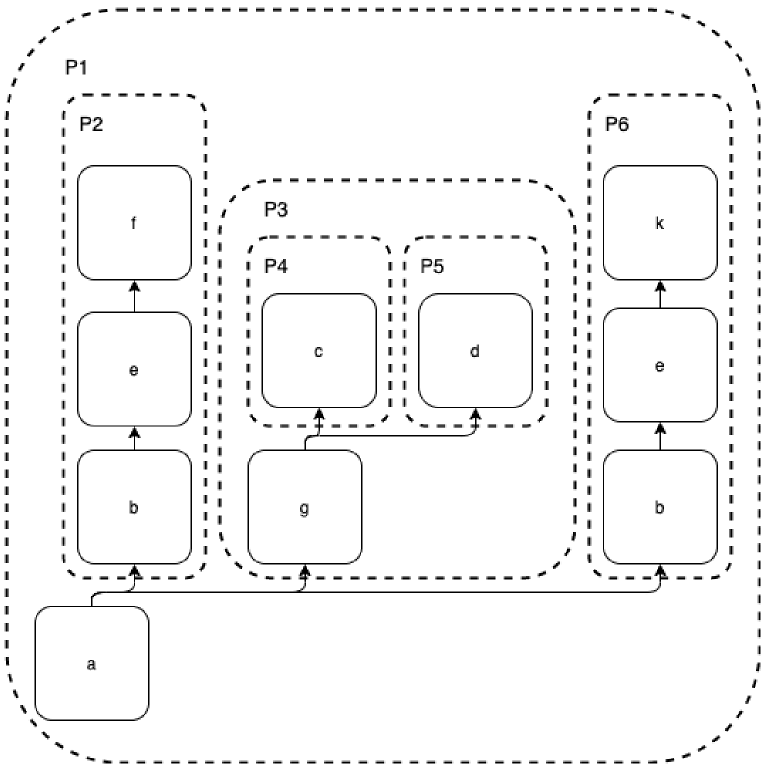


Figure 2.3: Similar preliminary phase tree

$P_i$ and $P_j$ in equation above are the two phase tree that being compared. $M_i$ and $M_j$ denotes the all methods in their global keys. The process iteratively compares each tree with other trees in the same level. For *P2* and *P6* in Figure 2.3, they all have methods $b$ and $e$ in their global keys. So the *Jaccard Distance* of these two tres will be $1 - \frac{2}{3} = \frac{1}{3}$. If the result is below the threshold they set, the two phase tree will be considered similar. In the final step, they provide a unique global identifier for all similar trees to reduce the total phase number.

**Frequent pattern mining**

The last step of abstracting phases is to execute frequent pattern mining over the phase tree. An ordered sequence of methods to accomplish a certain task may appear in multiple places in the program. They could further abstract their program by grouping such method sets as one frequent pattern phase. They perform a frequent pattern mining technique called “sequential pattern mining” (SPAM) [3] on each level of the hierarchical structure. It allows them to detect frequent sequential patterns from large transactional databases. For example, if the sequence of “isEndOfFile, readChar, appendToList” appears multiple times in an execution, SPAM can identify it and label this sequence as a frequent pattern.

### 2.1.3 Drawbacks

The previous work suggested a way of abstracting software programs hierarchically. However, after further analysis, we found some major flaws in this approach that need to be improved or addressed:

1. The previous work firstly uses *Jaccard Distance* to merge the similar phase trees. Although this approach can be considered effective, the comparison process is between the global keys we assigned to each tree. There is no sufficient evidence to indicate the global keys welly maintain the important information of the phase trees they extracted from. Comparing two trees directly for detecting similar phases may be beneficial to the phase clustering.

2. Phase clustering are perform on the phase tree level by level. Since the comparison was never happened globally, similar phases that nest in different level of the hierarchical structure were considered different.

3. Frequent pattern mining was used to identify a group of continuously invoked methods on each level of the phase trees. We found this step may be redundant and can break the correct hierarchical structure of the phase trees. We believe that an effective phase clustering technique will enable us to detect any patterns that found by frequent pattern mining since the majority of the method is similar. Moreover, SPAM may find patterns that consist of phase trees that are derived from different predecessors; this will break the correct hierarchy of the phase trees.

This thesis mainly introduces the improvement and changes over the previous work to tackle the drawbacks above while maintaining its advantages. We used the previous experimental procedures for reference and optimized each step accordingly. We change the data preprocessing method and choose a new technique to detect phases instead of using *Jaccard Distance* and *Frequent pattern mining*. Another improvement is by creating a new visualization for the abstracted program, which will present phases in all layers to users. The detailed explanation of the new approach will be discussed in the following chapters.

## 2.2 Blinky Instrumenter

Blinky [1] is a customizable instrumenter and execution tracer for software systems that compiles to Java Bytecode and targets the Java Virtual Machine for execution. Users can write their own Blinky profilers for their own needs.

Blinky has the capability to log the execution of any source- or byte-code instruction being executed during a software run. It can also log auxiliary execution events for the entry, exit, and completion of method invocations, method declarations, and compile-time source- and byte-code instructions [1]. The information in the output log files of Blinky could be various based on the Blinky profiler each user used. A blank file will be generated by default so users must write their own Blinky profilers for their own needs.

## 2.3 Flame Graph

The flame graph [7] is a new visualization solution for presenting a large volume of software profilers' output. The flame graph could provide users with a clear and easy-to-understand visualization to shorten their burden to comprehend and study foreign software.

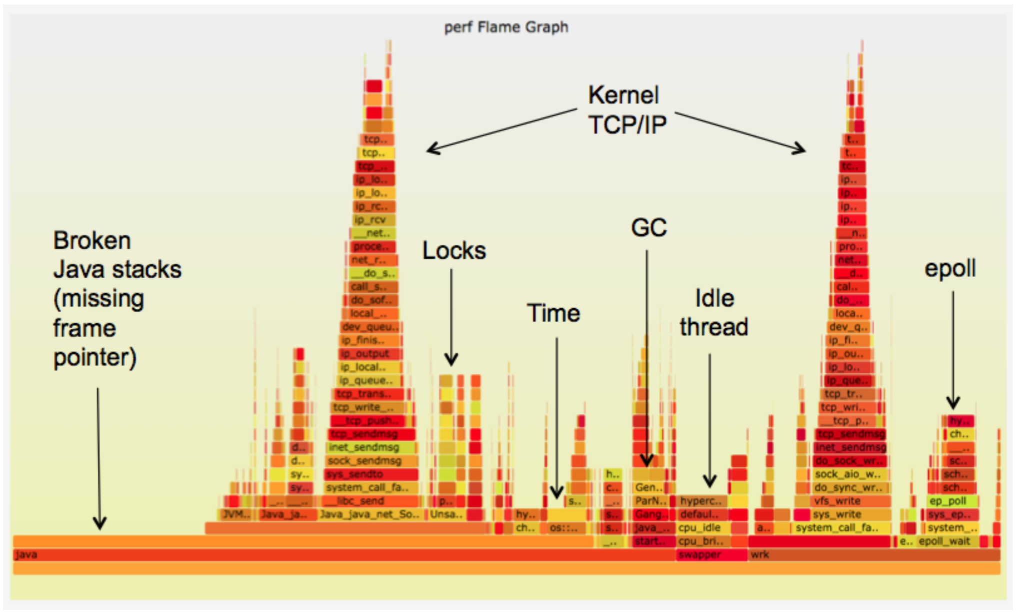


Figure 2.4: An example of the flame graph visualization, from [8]

Figure 2.4 shows us the actual visualization of a CPU profile. We can clearly see the phases as well as the time spent in each function in one execution. Performance engineers can look at this graph and identify which are the functions that may need to be optimized.

The idea of the flame graph is simple and intuitive. The creator of the flame graph Brend Gregg has indicated that the primary use of the flame graph is to visualize CPU profiles. The output of CPU profilers often contains stack traces with the percentage of time that was used in each trace event. The flame graph uses that information to visualize the stack traces that gives users a clear overview of all the CPU events. For example, for the simple Java program below:

```
public class Example {
  void a() {
    b();
    c();
  }
  void b() {
    d();
  }
  void c() {}
  void d() {}
  public static void main(String[] args) {
    Example e = new Example();
    e.a();
  }
}
```

The new instance of Example invoke `a()` function first. `a()` invokes function `b()` and `c()`, and function `b()` invokes function `d()`. If we assume function `a()` to be the root function and each function takes various time to execute, the output of this program after profiling and the flame graph generated will be something like Figure 2.5:

```
Begin Profiling
1: Name: a, Depth: 0, Time: 1000ms;
2: Name: b, Depth: 1, Time: 400ms;
3: Name: d, Depth: 2, Time: 250ms;
4: Name: c, Depth: 1, Time: 200ms;
End Profiling
```

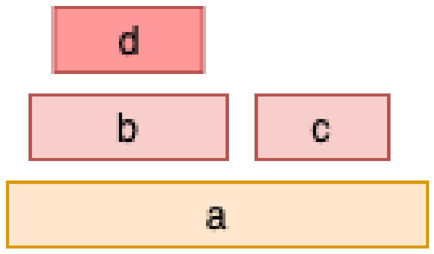


Figure 2.5: Event trace and the generated flame graph

In the original flame graph, the y-axis stands for the stack depth and the x-axis is the alphabetical stack sort to maximize merging. Inspired by flame graphs, Google has integrated a tool called flame charts inside the Chrome DevTools. The events in the x-axis of flame charts show the actual time each event starts and ends rather than an alphabetical sort. Since the order of events is an important aspect of the execution phase, we adopted the idea of flame charts and implement our own visualization.

## 2.4 D3.js Library

We choose to implement the visualization in JavaScript due to its rich library support and its ability to be demonstrates on any platform. Among the related open-source visual libraries, D3 is currently one of the most popular libraries. Although an open-sourced flame graph project based on Perl already exists on the web, it does not give users much room for customization. Therefore, this project takes this open source project as a reference and chooses to use D3.js to realize visualization.

D3 stands for Data-Driven Documents [6]. It allows you to bind arbitrary data to the Document Object Model (DOM) and then apply data-driven transformations to the Document for visual presentation using CSS, HTML, and SVG. The D3 project began in 2011 and grew out of the Protovi project of Stanford University's Visualization Research Group in 2009. D3 can provide a more expressive line framework and provides better visual representation taking into account Web standards. D3 uses SVG to give XML a graphics format used to describe two-dimensional vector graphics. SVG is a resolution-independent graphics format supported by major browser vendors. The D3 version used in this project is 6.0, which was released in August 2020.

# Chapter 3

# Phase Detection

In this chapter, we elaborate on the new approach for addressing the problem we discussed earlier. We first write a customized Blinky profiler to obtain programs' event traces. Then preprocess the event traces to transform text-based information to the tree data structure. Finally, we discuss how we calculate the *tree-edit-distance* for each pair of execution trees to determine if they are similar.

## 3.1 Data Collecting

Blinky can monitor many types of execution events. It requires us to inject a profiler file to produce the event traces we need. A profiler file tells Blinky which events should be monitored and which should be ignored, and what to do with these events.

We write a customized Blinky profiler to collect the raw data for our project. For our purpose, we instrumented the method-enter and the method-exit events. The reason why we added the method-exit event to our previous work is because we believe the length of time the methods runs is a factor to affect phase similarity.

Figure 3.1 shows the Blinky output for the simple Java program below. Blinky first runs a static analysis to detect all the methods in classes before dynamically documenting the program's execution. It assigns a method ID and an event ID for each method and event (enter and exit event) respectively. Each event also contains a method ID to indicate to which method this event is referring. The static analysis also tells us the method name, the class name, and the package name.

The traces that start with "$$$" are the dynamic events of one execution recorded by Blinky. Just as the figure shows, the program starts with the *main* method (Event 14), and then the constructor function (<init>()) is invoked (Event 1). Events are ordered in the actual sequence in which the program is run. We also configured Blinky to let it capture the *call depth*, the *timestamp* and the current *thread ID* for our future analysis.

```java
package test;
class Demo{
  void One() {
    Two();
    Three();
  }
  void Two() {}
  void Three() {}
  public static void main(String[] args) {
    Demo demo = new Demo();
    demo.One();
  }
}
```

```
$$method$$ <init>()V test/Demo ID=1
$enter$ 1 1
$return$ 2 1

$$method$$ One()V test/Demo ID=2
$enter$ 3 2
$return$ 7 2

$$method$$ Two()V test/Demo ID=3
$enter$ 8 3
$return$ 10 3

$$method$$ Three()V test/Demo ID=4
$enter$ 11 4
$return$ 13 4

$$method$$ main([Ljava/lang/String;)V test/Demo ID=5
$enter$ 14 5
$return$ 18 5
Instrumented Class = test/Demo

true
Starting tracing
$$$ $enter$ EventId=14 CallDepth=1 Timestamp=29085 Thread=1
$$$ $enter$ EventId=1 CallDepth=2 Timestamp=1533439 Thread=1
$$$ $exit$ EventId=2 CallDepth=2 Timestamp=1617215 Thread=1
$$$ $enter$ EventId=3 CallDepth=2 Timestamp=1706525 Thread=1
$$$ $enter$ EventId=8 CallDepth=3 Timestamp=1797765 Thread=1
$$$ $exit$ EventId=10 CallDepth=3 Timestamp=1911502 Thread=1
$$$ $enter$ EventId=11 CallDepth=3 Timestamp=2006953 Thread=1
$$$ $exit$ EventId=13 CallDepth=3 Timestamp=2098716 Thread=1
$$$ $exit$ EventId=7 CallDepth=2 Timestamp=2174432 Thread=1
$$$ $exit$ EventId=18 CallDepth=1 Timestamp=2255811 Thread=1
endProfiling
```

Figure 3.1: Event traces

## 3.2 Event-Trace Preprocessing

Once we acquire the event traces, we need to parse them into tree data structures to compare. We will explain how we write a program to help us to convert event traces into trees. The input of the program should be a complete output file of Blinky.

We create a new class called "*FunctionNode*". The instances of this class represent method invocations in an execution. The *FunctionNode* contains all the information we need for comparing the similarity including the *method name*, *the class name*, *the children list*, *the start timestamp*, *the end timestamp*, and the *callDepth*. We should notice that one method could have multiple *FunctionNode* instances if it being called more than once. The code for the FunctionNode class is shown below. The *groupId* property is related to phase clustering and will be explained in the next section.

```java
public class FunctionNode {
  public String name;
  public String className;
  public int groupId = 0;
  public List<FunctionNode> children = new ArrayList<>();
  public long startTime;
  public long endTime = -1;
  public int depth;
}
```

We also create a *layerMap* to store all the dynamic event traces. The key of the map is the *call depth*, the value of the map is a *ArrayList* that stores all methods called (*FunctionNode*) at that depth in order.

The program reads the input file line by line and processes the traces depending on their types:

1. For lines that start with “$$method$$”, we create a mapping between *method ID* and *method name*.
2. For lines that start wieh “$<event name>$”, we create a mapping between *event ID* and *method ID*.
3. A new *FunctionNode* instance will be created for each enter-event start with “$$$”. We first find its corresponding method name based on its *event ID*, and combined it with other information in the event trace to create the *FunctionNode* instance. The new instance created will be added to the *layerMap* based on its *call depth*.
4. For exit-events starts with “$$$”, we change the *end timestamp* of its corresponding node. We then need to add this node as the child to its parent node (the last node of the layer below).

For the event traces in Figure 3.1, the *layerMap* generated is shown in Figure 3.2. This diagram shows the basic structure of the *layerMap*, with the rounded rectangle on the right representing the *ArrayList* and the rectangles inside the rounded rectangle representing the instances of *FunctionNode*.

### 3.2.1 Overhanging Method

Under normal circumstances, all nodes in the *layerMap* should have a direct parent node. The method overhang is the situation where a node in the upper layer does not have a direct parent, like Node *A* in Figure 3.3.

The method overhang happens when instrumenting the overloaded constructor functions and when instrumenting the rewritten system methods in a class that extends the system class.

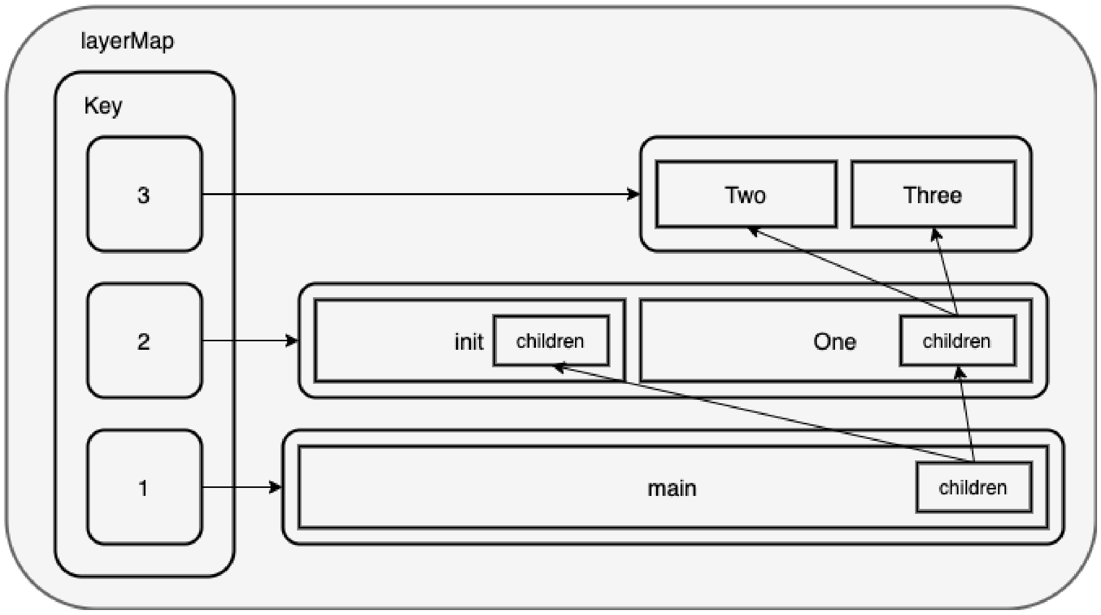


Figure 3.2: The layerMap after processing the event traces.

This cannot be prevented due to the Blinky configuration and the internal working sequence of the JVM, so we manually added their missing ancestors for all the overhanging nodes, as shown in Figure 3.4. All the added ancestors shares the same timestamp as the overhanging node.

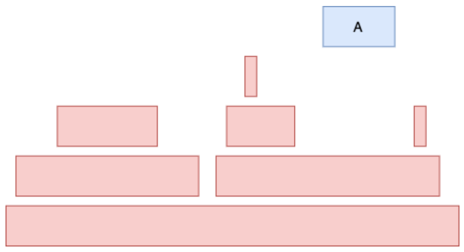


Figure 3.3: The overhanging node

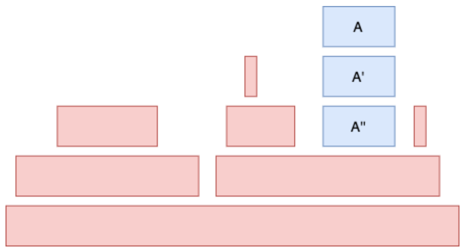


Figure 3.4: Added ancestors

## 3.3 Tree Edit Distance

In the last step, we get a *layerMap* that contains *FunctionNodes*. Each node has a *ArrayList* called *children* which contains all other nodes it invokes. We can consider each node as a *tree* structured data. We perform the phase clustering process by determining the similar *trees* in the *layerMap*. Since the comparison is between real trees, our approach uses the *tree edit distance* to calculate the tree similarity. Although we use an open-sourced library developed by Database Research Group [25] for this project, it is also essential to understand the idea of the *tree edit distance* algorithm for designing the comparison workflow and future improvement. In this section, we introduce the basic concept of the algorithm and the reason why we use it as our phase clustering method.

### 3.3.1 Definition

The *tree edit distance* is defined as the shortest sequence of elementary operations which transforms one tree into another. The elementary operations we consider are:

1. Deletion: Delete a node and connect its children to its parent.
2. Insertion: Insert a node between a parent node and all its consecutive children.
3. Replacement: Change the name of one node to another.

The idea of *tree edit distance* comes from the similar operations we perform when comparing two string (*string edit distance*). For example, to transfrom the word "fun" into the word "soon", we need to perform one insertion and two replacements. If we assume that each operation cost 1, then a minimum edit distance between these two words is 3. The *tree edit distance* can be considered the generalization of the *string edit distance.*

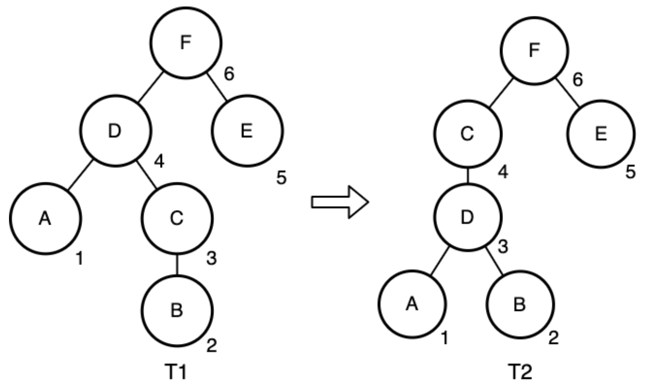
### 3.3.2 Zhang-Shasha Algorithm

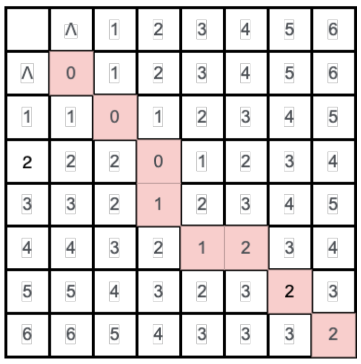
Zhang-Shasha [42] is one of the first algorithms for solving the tree edit distance. Just like most algorithms to calculate string edit distance, this algorithm also uses the dynamic programming paradigm. Since the APTED algorithm [28, 29] that we used in the open-sourced library is an improvement on the Zhang-Shasha, we briefly introduce the idea of Zhang-Shasha.

Equation 3.1 is the general equation of calculating the tree edit distance. We can calculate the edit distance once we specify the cost of each base operation. The tree distance is the minimal cost among the three subproblems.

$$td(\overset{\circ}{\triangle\triangle}, \overset{\bullet}{\blacktriangle\blacktriangle}) = \min \begin{cases} td(\triangle\triangle, \overset{\bullet}{\blacktriangle\blacktriangle}) + Cost(delete) \\ td(\overset{\circ}{\triangle\triangle}, \blacktriangle\blacktriangle) + Cost(add) \\ td(\triangle\triangle, \blacktriangle\blacktriangle) + Cost(replace) \end{cases} \tag{3.1}$$

For example, The dynamic planning table for transforming tree *T1* into tree *T2* is shown in Figure 3.5. Cell (m, n) is the minimum distance to transform the tree that rooted at $m$ on the left to the tree that rooted at $n$ on the right. After the computation, the bottom right cell stores the distance between *T1* and *T2*. The red path is the most efficient path to transform two trees. In this example, one replacement and one deletion are needed to change tree *T1* to tree *T2*. If we assume the cost of both two operations are 1, the tree edit distance of these two trees is 2.





| | Λ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Λ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 2 | 2 | 0 | 1 | 2 | 3 | 4 |
| 3 | 3 | 2 | 1 | 2 | 3 | 4 | 5 |
| 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 |
| 5 | 5 | 4 | 3 | 2 | 3 | 2 | 3 |
| 6 | 6 | 5 | 4 | 3 | 3 | 3 | 2 |

Figure 3.5: Distance table

## 3.4 Duplicate Event Detection

In the worst case, to check the similarity between trees globally, each tree in the *layerMap* has to be compared with all other trees. Since this process runs in $O(n^2)$, we can notice that a CPU cannot handle all the event traces in an acceptable time if the number of events exceeds a certain amount.

In project experiments, we found that even if the number of traces exceeds a million, the number of different methods invoked will not exceed 10,000. Repeated and identical function calls contribute most of the size of the trace file. To find similar phases globally, the first step after data preprocessing is to detect the duplicate events and exclude them in the following comparison steps.

To identify duplicate events, we create a map with method name as the key and a list of *FunctionNodes* as the value. We traverse the entire Map, keeping only the *FunctionNode* that are different from each other in each list. Since recurrent behaviors are common in program execution, the number of total events to be compared globally will reduce immensely after this step.

## 3.5 Phase Clustering

After the preprocessing, we convert the event traces to a *layerMap*, which is easier for programmers to analyze and check information. The goal of phase clustering is to find the collection of elements in the *layerMap* that have similar functionality and group them. By performing phase clustering, we can significantly reduce the trivial information in each layer and facilitate the comprehension process.

The previous work was just to compare nodes layer by layer, which meant that similar trees could never be found at different call depths. To globally detect similar nodes, each node must be compared to all other nodes. In the last section, we mentioned that the phase detection process is based on the *tree edit distance* algorithm. However, the time complexity of the algorithm is too high to compare all possible node combinations. The classical Zhang-Shasha runs in $O(n^4)$ time and the *APTED* algorithm that we used runs $O(n^3)$ time [27]. Although the improved algorithm significantly reduces the time complexity of calculating the *tree edit distance*, it is still not practicable to compute the distance of all node pairs when we have millions of event traces. Thus, strategies that help to reduce the total comparison times are needed. In Figure 3.6, preliminary screening reduces the total comparison times, trimmed node comparison speeds up the *tree edit distance* calculations, and phase labeling groups similar phases together for the later visualization. Steps enclosed with a rectangle reduce the total processing time of phase clustering.

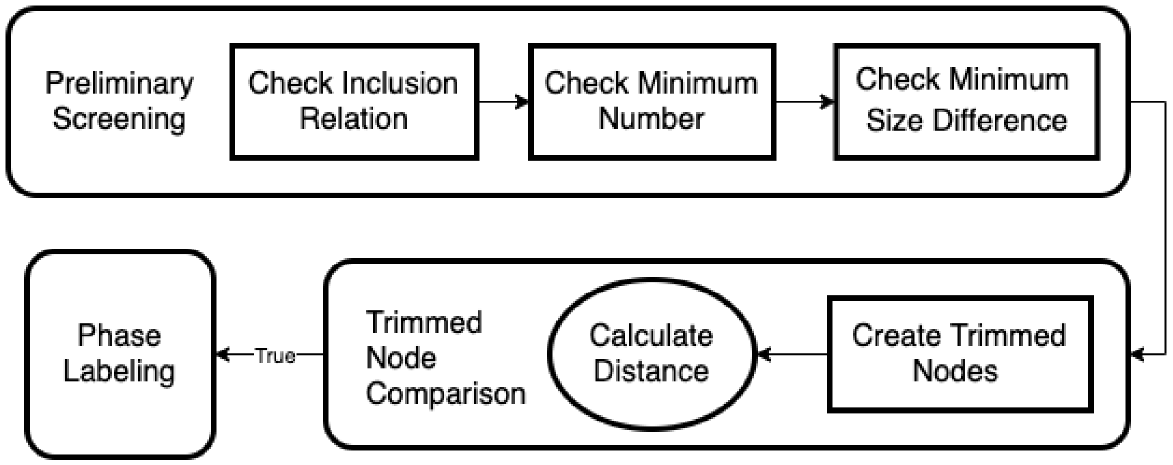


Figure 3.6: Phase clustering workflow

### 3.5.1 Preliminary Screening

The screening workflow that reduces unnecessary comparisons consists of three steps: check inclusion relation, check minimum number, and check minimum size difference, as shown in Figure 3.6. The main purpose of these steps is to sift out tree pairs that are unlikely to be similar or not logically correct to be compared.

### Check Inclusion Relation

To detect similar trees globally, one tree has to be compared with all other trees in the *layerMap*. However, intuitively, one tree should not be compared to its ancestor. We write code to detect whether the two trees to be compared have an inclusion relationship, and if the answer is true, the comparison is aborted. We will not compute the *tree edit distance* for the tree $A$ and tree $B$ in Figure 3.7.

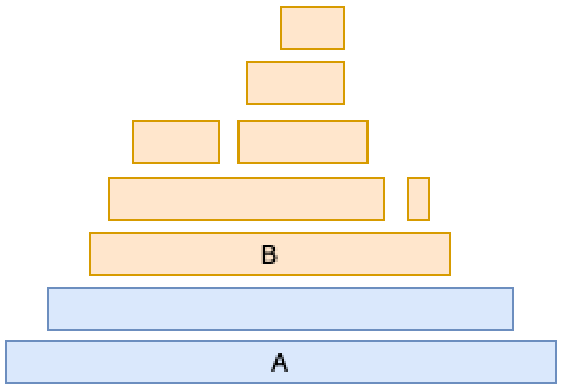


Figure 3.7: Inclusion relation

### Check Minimum Number

Trees that contain few nodes are not worth comparing because most of them are located in the farthest layer. The reasons for ignoring those nodes include: (1) the number of them is so large that they will increase the computational time, and (2) trees with small size do not represent siginificant “phase” execution. We set a parameter to determine the minimum number of nodes in a tree that will proceed to be compared. For example, if we set the parameter value to 5, the tree in Figure 3.8 will be ignored.

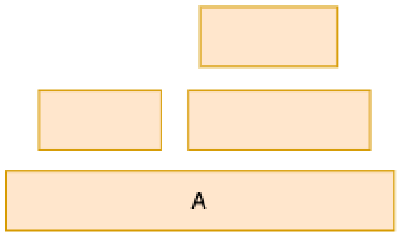


Figure 3.8: Tree with four nodes

### Check Minimum Size Difference

Since the *tree edit distance* is calculated between two trees. We can firstly consider factors that affect the calculation results. In this step, we consider the size difference between the two trees. The size difference is the absolute value of the number of nodes in the first tree minus the number of nodes in the other. From the definition of the *tree edit distance*, we can recognize that the size difference between trees is the minimum possible value of their edit distance. Just like Figure 3.9, we can sift out tree pairs with high size differences to avoid unnecessary calculation.

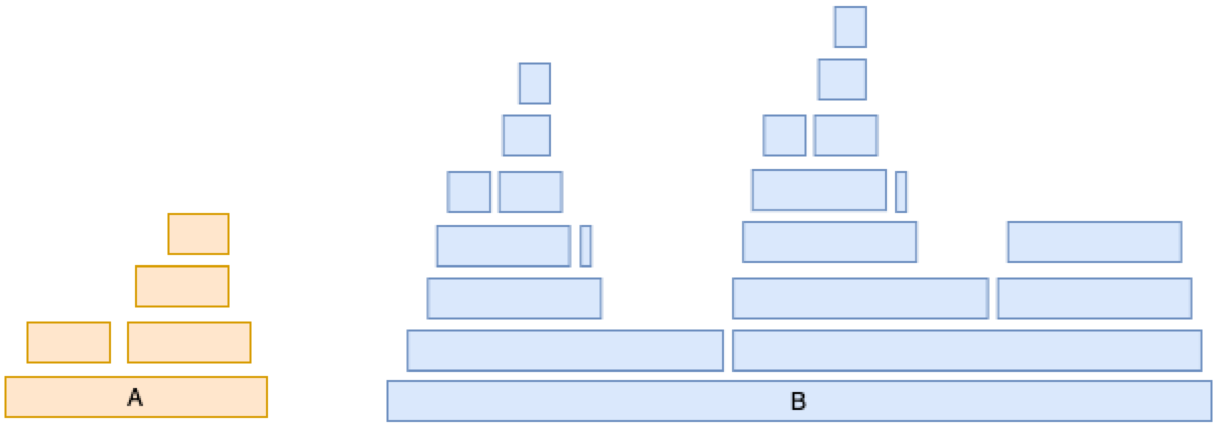


Figure 3.9: Trees with high size difference

We used the event traces of JsonConverter project to test the workflow. For the 13752 nodes in the *layerMap*, the total comparison number reduce from $1.9 \times 10^8$ to 229213.

### 3.5.2 Trimmed Node Comparison

*Tree edit distance* calculates the exact minimum number for one tree transforming to another. However, the algorithm run in $O(n^3)$, and the time for calculating distance for trees with hundreds of thousands of nodes will be too long. After analyzing the key factors that affect the calculation results, we design a technique to trim the nodes for each comparison.

Firstly, most of the time is wasted acquiring the exact edit distance. We chose to intercept the lower layers of the tree to represent the whole tree for two reasons: (1) nodes in the lower layer have a greater weight in describing the method behavior; (2) nodes in the upper layer contribute most to the increase in edit distance. For example in Figure 3.10, trees $C$ and $D$ will be compared instead of $A$ and $B$. In the course of our research, we also found that the functions with longer running time were better at summarizing the tasks performed in the current phase of the program. When creating the trimmed nodes, we also chose to ignore nodes with too short running time.

By applying these techniques, the tree sizes will always be under a constant, so that the time complexity on the total time cost of clustering is controlled to the minimum.

### 3.5.3 Phase Labeling

If the edit distance is under the threshold, a similar phase is detected. We label the phase by assigning them a unique *groupId* if they do not belong to any group. If two similar trees

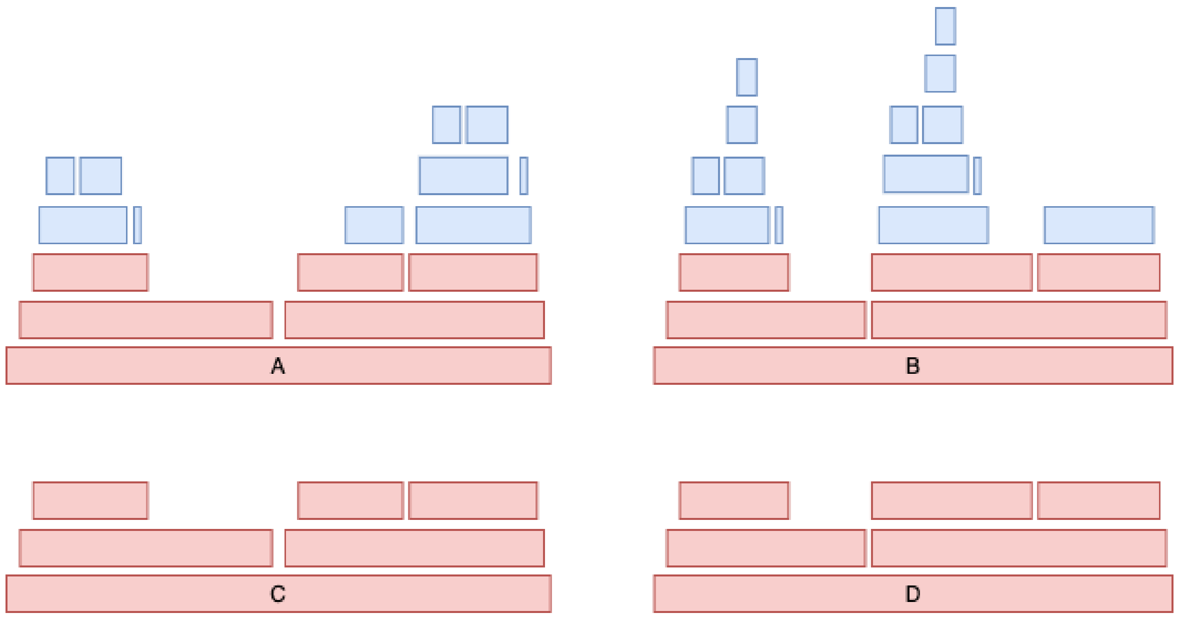


Figure 3.10: Trimmed node comparison

already have different *groupId* (which means the tree is similar to another tree that was analyzed before), we use the idea of *Union Find* and change their IDs to the smaller one.

## 3.6 Conclusion

In this chapter, we purposed a new approach for handling event traces for software dynamic analysis. It consists of a series of steps including event trace prepossessing, phase clustering, and phase labeling. We utilized the APTED algorithm to determine the similar trees in the *layerMap* and designed a workflow to reduce the time cost of phase clustering. In the next chapter, we introduce how we create the flame graph visualization based on the generated *layerMap*.

# Chapter 4

# Phase Visualization

The phase clustering step can transform the large event traces into phases and group similar phases together. It turns complex and trivial instrumenters' output into a data structure that contains hundreds or thousands of execution phases. However, it is still inefficient for source code maintainers to comprehend the functional behaviors in this execution just by examining the texted output data structure. We need a more intuitive technique to help present all information to users more directly and intuitively.

Visualization is considered to be an approach to convey large amounts of information to humans. We add phase information in the trace events and visualize the events based on the flame graph. The visualization assists in software comprehension that is required for certain software maintenance tasks. We introduce the functions and the implementation of the visualization in the following sections.

## 4.1 Design Choice

Research in software visualization has found out that the effectiveness of visualization is difficult to assess. The fact of lacking characterization criteria in this field makes it hard to evaluate the visualization [12]. One possible way to evaluate information visualization is a visual inspection by experts [16]. However, most of the evaluation happened after the visualization was built and is not much help for designing a visualization for a new propose.

To maximize the possibility of designing a software visualization that is useful in industry contexts, we chose to create the visualization base on the flame graph. The flame graph is a new visualization solution for presenting a large volume of profilers' output. This visualization technique has been proven to be effective on several industrial platforms including Netflix, Google, AWS and JetBrains [7]. Although the flame graph is primarily used for performance analysis, we explore the use of flame diagrams to present the phases for software comprehension.

## 4.2 Visualization Introduction

The goal of our visualization is to present the information of event traces as well as the similar phases to the users. The events in the x-axis of the original flame graph are ordered alphabetically. In each layer, all methods with the same method name are treated as the same and the flame graph will group them as one method with a longer execution time. While this solution is appropriate in performance analysis, it is not suitable when presenting phases because we want to maintain the chronological order of method calls. We modify the original flame graph to provide the timestamp information and let users know what method or phase is being executed in each time frame.

After the modification, the visualization can provide the information we obtained in the earlier steps. The y-axis represents the call depth. The method in the bottom layer is the entry point of this execution. The x-axis represents the time. The scale of the X-axis is determined by the total running time. Each rectangle in the visualization represents a method call. The length of the rectangle gives a direct representation of the running time relative to the entry method. Users can hover over the visual interface to browse through all the detected phases. Similar phases can exist in different call depths and are shown in red (Figure 4.1).

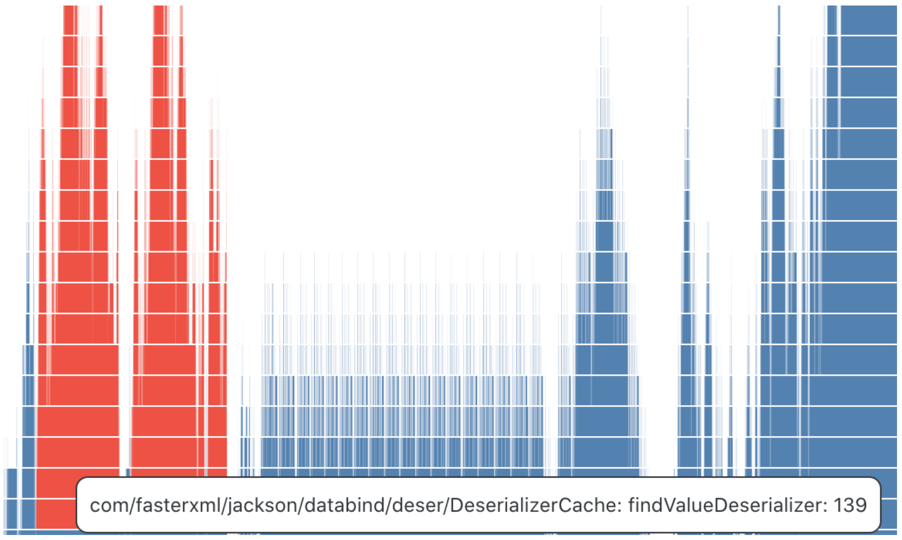

com/fasterxml/jackson/databind/deser/DeserializerCache: findValueDeserializer: 139

Figure 4.1: Screenshot of the flame graph implementation

### 4.2.1 Interactions

Users can retrieve information of event traces by interacting with the visualization. After selecting the execution from the “Choose Project” button, users can explore the visualization by zooming, dragging, hovering, and resetting.

### Click or Scroll to zoom

Different tasks require source code maintainers to focus on different levels of details of execution. The default behavior of our implementation can give programmers the method call situation below the first 30 levels of call depth. We set the number 30 bases on the height of each box and the normal size of modern personal computers. Layers over 30 will be hidden by default at the top of the visualization frame and displayed by interacting with the system. To assist users to analyze all phases or methods with different running times, we add two ways to enable zooming on the timeline (Figure 4.2).

Users can zoom in or zoom out by scrolling their mouse. Doing so will proportionally scale up or down the entire visualization. One possible scenario of this function is to minimize the visual image so that the whole outline can be displayed on a screen. The outline can tell us the basic information about this execution, such as the major phases and the maximum depth of the call.

Other way to zoom is by clicking the method rectangle. This feature can help users to horizontally zoom in on events of interest. When a box is clicked, the flame graph is scaled horizontally. This reveals more detail, and the submethod box of the method being clicked will also scale up.

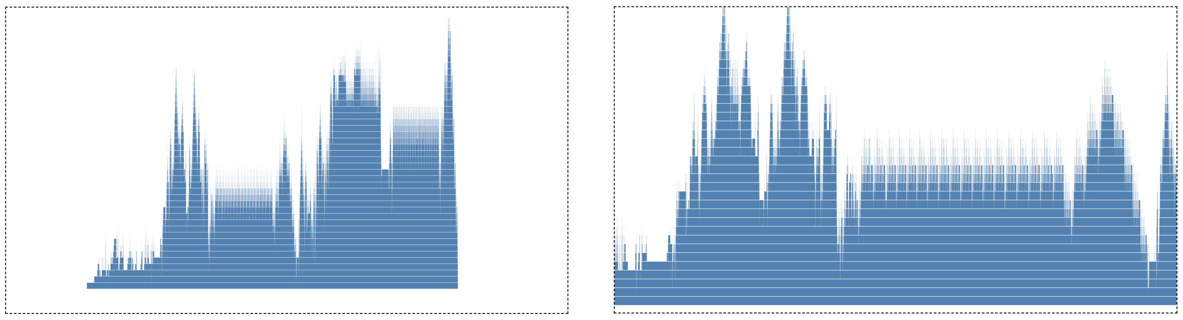


Figure 4.2: Zooming in and zooming out

### Drag to Move

The visualization supports using the mouse to drag the visual image to achieve displacement. Users can drag the visual image to check for hidden boxes.

### Mouse-over for Information

On mouse-over of boxes, an informational tooltip displaying the class name, method name and the groupId will show up. The tooltip we designed can display the whole method name and its package path fully. Apart from that, if the mouse is over a phase detected earlier, all similar phase will light up simultaneously. Users can deepen their understanding of the program based on where each phase responds in the visualization (Figure 4.3).

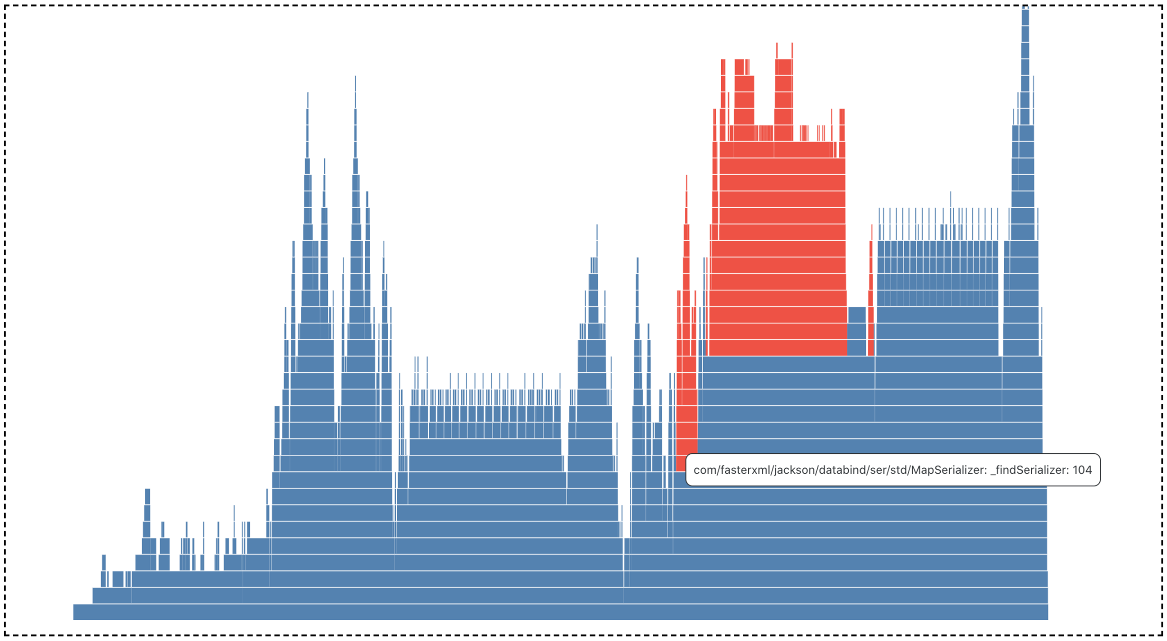

com/fasterxml/jackson/databind/ser/std/MapSerializer: _findSerializer: 104

Figure 4.3: Displayed the toolbox when the mouse is over the box

**Reset the Image**

Zooming visualizations often suffer from the problem of the user "getting lost" — being zoomed too far into an empty area, or being too far zoomed out to the point where the visualization is no longer visible. As such, to make it easier for the user to reposition the view, we added a Reset button to the visualization that, when pressed, returns the position and scale of the view to its default value.

### 4.2.2 Multithread Visualization

In the multithreaded program, the timestamps of methods from different threads are going to overlap, which means there will be multiple events at a certain point if we visualize methods from different threads in a single view. Thus, we separate methods from different threads and implement a feature to switch between threads. In our phase clustering step, we also calculate and compare events from different threads separately. Methods take turns running in multithreading, the time difference between entry and exit of a method does not represent its true running time. So for programs relying heavily on multithreading, the length of each event box will be inaccurate, which is one of the drawbacks of our implementation.

## 4.3 Flame Graph Implementation

The visualization was built by multiple web technologies including TypeScript, React, and D3.js. We use TypeScript to implement because it is more reliable and easier to refactor. We combine D3.js with React to create the web page. D3.js enables us to manipulate SVG based on the data. React gives us the ability to record the status of each operation and change the web contents accordingly.

### 4.3.1 Generate Input File

The *layerMap* obtained by clustering contains arrays of program call events at each call depth. Each call event contains basic information such as the time name, class name, phase ID, and timestamp. Since the parsing of JSON data is supported by D3.js, we choose to use tools to convert the *layerMap* data structure into JSON and make it the input file of the visualization.

### 4.3.2 Create Flame Graph

Once an execution was selected, the JSON input file will be added to the current React state. D3 will create event boxes recursively until all method events were created. We use the linear scale provided by D3 to determine the length of each box, which is proportional to its running time. All the interactive features were also attached during the flame graph creation.

### 4.3.3 Phase Prestore

The visualization lights up all similar phases once the mouse is over the root of the phase. To provide hierarchical phase information, children of the detect phase root do not have the same *groupId*. Thus, retrieving the same phase ID throughout the visualization and lights up all of its sub boxes simultaneously is time-consuming. To tackle this problem, we create a map with phase id as its key and the boxes array as its value. We store the correct information in it at the beginning of the visualization creation.

# Chapter 5

# Results

In the previous chapters, we introduced the main work of this project. We first design the workflow of similar phases while the program is running and then describe how to design a visual implementation that represents the information of these phases based on flame diagrams. To verify the effectiveness of these efforts, we present case studies on three different Java projects in this chapter. We first run our project in three Java projects and made a quantitative analysis of the running results of each software in this project. Then, we analyzed the indicators such as running time, number of similar stages, and characteristics of similar stages. Finally, we provided a qualitative analysis of the visualization in one of the applications and showed how visualization helps users understand the runtime information more quickly.

## 5.1 Experimental Design and Introduction

We selected three open-source Java projects for this case study. The three projects are Jackson, IText, and FindBugs. Jackson is a Java JSON library that provides multiple approaches to working with JSON including parsing JSON to Java objects and vice versa. iText is also an open-source library for creating and manipulating PDF files in Java. FindBugs is a Java application for static analysis to look for bugs and possible vulnerabilities.

Java libraries and applications have so many features that it is almost impossible to analyze every possible execution flow. In dynamic analysis, we often keep only a trace of the program's execution in a simple task. Thus, we selected only a part of the functions of each project for analysis.

For the first two projects, we choose to write an executable file that showed the basic functionality of both toolkits. For Jackson, we first create a JSON file, then call the methods in Jackson to read the JSON file and store the read structure in an object when the read is complete. Finally, we call Jackson's API again to turn the object we just assigned into a new JSON file. For IText, we call its API to create a new PDF file with a line of text into our local machine. For the FindBugs, we instrument its function of opening an XML file and displaying the analysis results.

In order to quantitatively analyze the validity and implementation efficiency of the detection in the phase, we run the three generated trace files in our project respectively. To obtain the result data, we counted the total number of methods, the total number of comparisons, the number of events, and other indicators in each project, and compared the changes of each indicator under different thresholds.

## 5.2 Experimental Results

For each experimental project, we calculated the results under different similarity thresholds through experiments, and the specific experimental results are shown in Table 5.1 to Table 5.6. The instrumenter's output for FindBugs has 16 threads. Since we choose to process event traces of different threads separately, we present the thread with the most event number in the result tables.

The first column of Table 5.1 represents the size of the output obtained after instrumentation for the projects; The second column is the number of events in the output file; The third column shows the number of different methods invoked during this execution. For Table 5.2 to Table 5.4, each row in the table represents a Java project, and each column shows project experimental results. Column 1 to 4 are the specific results of the experiment, where *Phases* represents the number of events detected into Phase, which means that each event has a non-null groupID. We should note that in the *Phases*, Events can have the same groupID. Group represents the number of different phases in this execution. *Phases I* and *Groups I* are the results when only compare the event with same root name, whereas *Phases II* and *Groups II* are the results when the comparison of events with different name added. Table 5.5 is the total time of parsing the event traces and detecting the similar phases.

By examining the results in Table 5.5, we can verify the effectiveness of our approach of reducing the processing time in Chapter 3. The processing time of the three projects is 3 seconds, 31 seconds, and 15 seconds respectively. We can see that even with the number of events in the hundreds of thousands, our approach can complete text parsing and similar phases lookup in a relatively short time. After all the steps before calculating the *tree edit distance*, the time complicity of finding similarity globally has dropped from $O(n^5)$ to $O(mn)$, where m is the number of the different methods being called in this execution.

From Table 5.3, we can see how our approach performs under different projects. When the similarity threshold is set to 0.7, the average proportion of events detected as phases across the three projects was 87%, 94%, and 98%, which indicates that the program runs consistently across projects. The events that are not phases are likely to be the methods that are called only once in this execution.

We define the similarity threshold as $\frac{tree \quad edit \quad distance}{number \quad of \quad nodes \quad in \quad the \quad smaller \quad tree}$. The data in Table 5.2 to Table 5.4 are the experimental results with thresholds set to be 0.6 to 0.8, respectively. We found that when the threshold increased, the data under the Phases increased correspondingly, and the data in the Groups column decreased correspondingly. This result indicates that more events are classified as similar phases with the decrease of the threshold value, which verifies the correctness of stage abstraction in this method to a certain extent.

The "I", "II" in Table 5.2 to Table 5.4 is the results under two different circumstances. "I" columns are the results when the phase detector only compare events with same method names, and "II" are the result when comparing events with events with different names.

The results show that only a small number of new stages were found in "II". Most of the similarities come from the events with the same name, which is under our expectations. Table 5.6 is the list of some of the method names verified to be similar in "II" in iText at the threshold of 0.7. The first column represents their *groupID*, the second and the third column are the root names of the trees. By comparing the method names of the left and right columns of Table 5.6, we can further demonstrate that our approach successfully detects similar phases of program execution.

| Projects | Size | Events | Methods |
|---|---|---|---|
| Jackson | 2.9 mb | 13752 | 4899 |
| IText | 27.3 mb | 188765 | 2980 |
| FindBugs | 31.6 mb | 230814 | 4981 |

Table 5.1: Basic information of the three projects

| Projects | Phases I | Groups I | Phases II | Groups II |
|---|---|---|---|---|
| Jackson | 11985 | 629 | 12365 | 659 |
| IText | 177942 | 408 | 178053 | 424 |
| FindBugs | 226751 | 1289 | 226850 | 1321 |

Table 5.2: Results when similar threshold = 0.6

| Projects | Phases I | Groups I | Phases II | Groups II |
|---|---|---|---|---|
| Jackson | 11942 | 638 | 12003 | 662 |
| IText | 177918 | 423 | 177983 | 438 |
| FindBugs | 226739 | 1312 | 226836 | 1345 |

Table 5.3: Results when similar threshold = 0.7

| Projects | Phases I | Groups I | Phases II | Groups II |
|---|---|---|---|---|
| Jackson | 11907 | 645 | 11982 | 667 |
| IText | 177904 | 424 | 177952 | 433 |
| FindBugs | 226721 | 1327 | 226823 | 1360 |

Table 5.4: Results when similar threshold = 0.8

| Projects | Time I | Time II |
|---|---|---|
| Jackson | 1076 ms | 3539 ms |
| IText | 20113 ms | 39933 ms |
| FindBugs | 12968 ms | 17923 ms |

Table 5.5: Results when comparing events with different root name

| Group ID | Methods with same ID | |
|---|---|---|
| 109 | applyDestination | applyLinkAnnotation |
| 109 | applyDestination | applyAction |
| 109 | applyAction | getBorders |
| 126 | retrieveMinHeight | retrieveMaxHeight |
| 128 | restoreState | endText |
| 137 | endElementOpacityApplying | endRotationIfApplied |
| 144 | writeFloat | writeDouble |
| 146 | writeXrefTableAndTrailer | writeToBody |
| 150 | isOverflowProperty | getPropertyAsTransparentColor |

Table 5.6: Different methods with same group ID

## 5.3 Visualization Analysis on Jackson

In this section, we perform a qualitative analysis on the visualization ouput image of Jackson to check if the functional pattern could be clearly present. The execution consists of two main parts, which are parsing a JSON file to a object and converting the object back to a JSON file. The image generated by our implementation is in Figure 5.1

By hovering the mouse over the two longest boxes on the second floor (Figure 5.2), we can see that the two main jobs of this program are to read values and write values, which is corresponding to the source code.

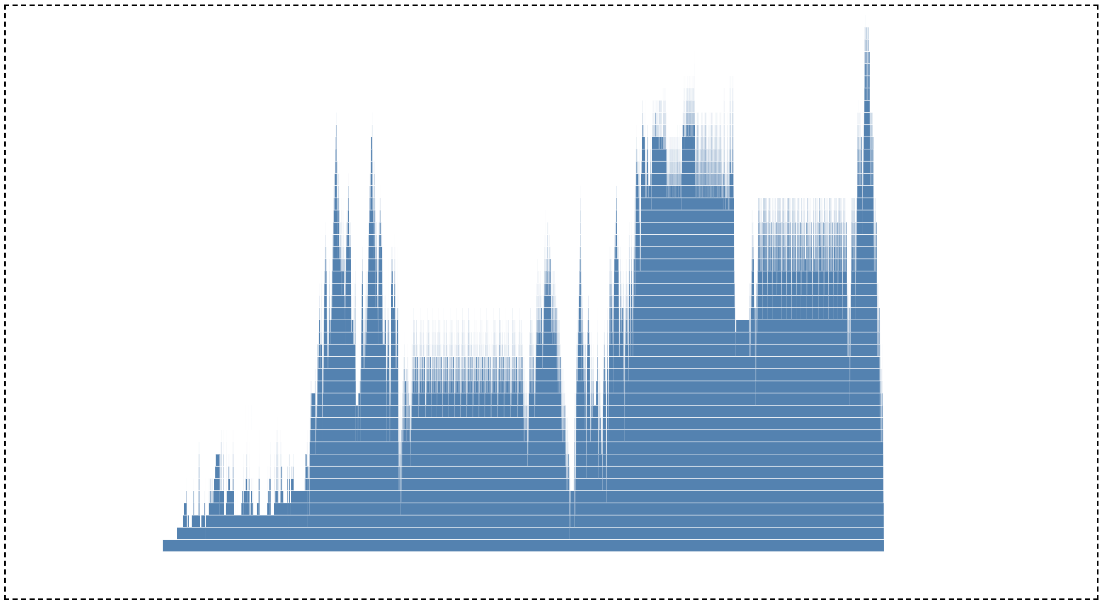


Figure 5.1: Screenshot of the execution

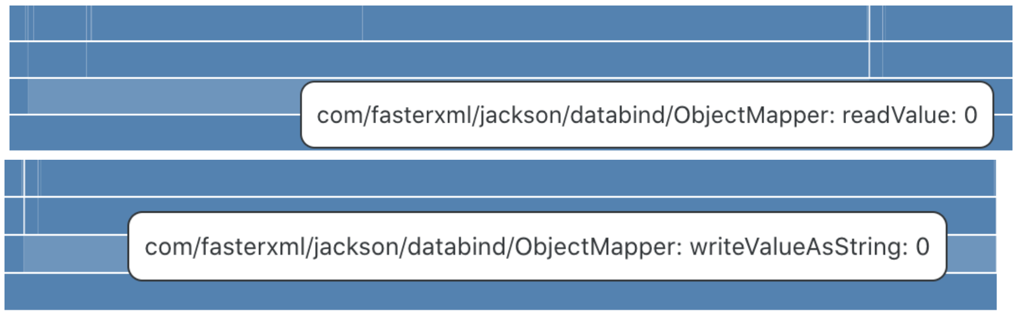

com/fasterxml/jackson/databind/ObjectMapper: readValue: 0
com/fasterxml/jackson/databind/ObjectMapper: writeValueAsString: 0

Figure 5.2: Two main phases

We can use the interaction to explore the subphases in the deeper call depth.For example, we can hover over our mouse on the *readValue* box to see the subphases. The two longest box above the *readValue* box are method *_findRootDeserializer* and *deserialize*. From the root name, we can be informed that the program first needs to create or find a object called Deserializer and deserialize the input. We can keep doing these operations combined with other interactive features to explore the whole execution.

The highlighted phases can tell more information of the execution. For example, inside the *deserialize* method, there are two similar phases that have quite different running times and shapes (Figure 5.3). Since they are similar, we know that they are basically doing the same thing. Since they are similar, we know they are basically doing the same thing. This kind of information cannot be obtained without the phase detection and it reduces the burden of program understanding.

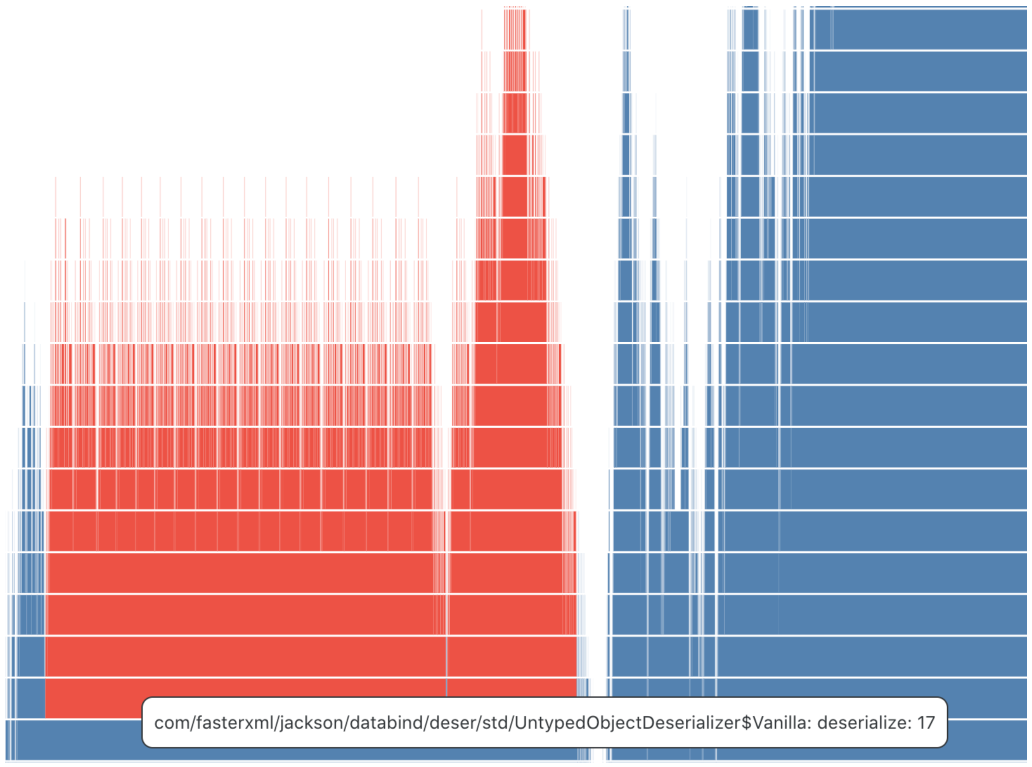


Figure 5.3: Similar Phases with different shape and running time

# Chapter 6

# Discussion

In the previous chapters, we discussed how our project to process massive data using phases detection. We also introduced our flame graph implementation for presenting multi-level phase abstraction. In this chapter, we mainly discuss the improvements of the project on the basis of our previous work in several aspects. In the last section, we list some limitations this project has.

## 6.1 Processing Time

The newly designed workflow has reduced the time of phase detecting substantially. Unlike the previous work, we did not perform frequent pattern mining in each call depth. We choose to use the *tree edit distance* algorithm to determine the similar phases. Also, the algorithm works in $O(n^3)$, and the trimming technique we used has limited the number of nodes in comparison to a constant level. By comparing nodes with the same names and deleting duplicate nodes, our approach is able to process files with millions of event traces in a short time.

## 6.2 Phase Detection Mechanism

We considered the limitations our previous work has and improved the phase-detection mechanism to support finding more phases globally. In our previous study, phase clustering and frequent pattern mining were performed layer by layer, which means nodes and events in different call depths can never be linked and analyzed together. In our approach, similar phases can be detected from different call depths. Revealing similar phases at any point of execution provides more information on the programs' functionalities gives users more correct phase detection results.

## 6.3 Visualization's Comprehensibility

This visualization, based on the flame map, provides more information about the execution than our previous project. As shown in Figure 6.1, the visualization of the early project only holds the phases in one hierarchy. Users need to click the one box to see its subphases. It is hard to check phases in the deeper layer as well as the relationship between them. The flame graph provides users with a clear execution structure and shows all execution events from the bottom to the top layers. In our implementation, users can combine flame map thumbnails with interactive features to focus on the parts of their interest more quickly. Moreover, in the previous implementation, the length of each box does not represent the running time, but the number of sub-phases within it. That design does not follow the user's intuition and may has negative impact for their comprehendsion. We use the timestamp to accurately describe the running time of each phase in the visualization.

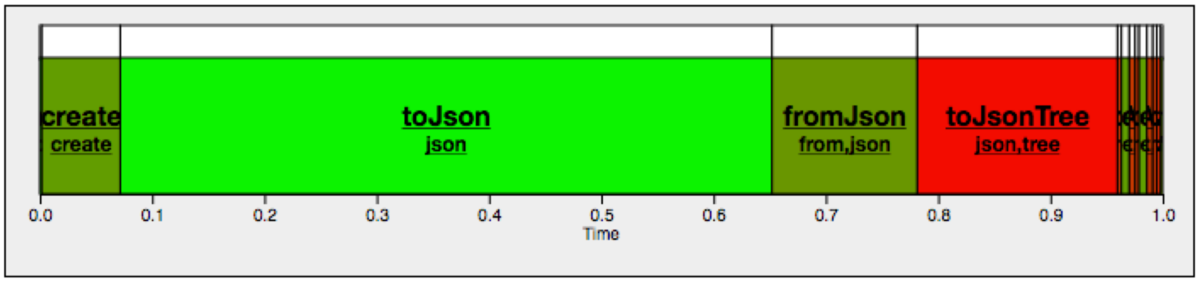


Figure 6.1

## 6.4 Limitations of the Model

Although our new approach has improved our previous model and addressed some problems its has, there are still some limitations that may affect the results and performance.

### 6.4.1 Phase Labeling

In the phase labeling step, we assigned each phase a *groupId*, however, this number is only useful for the program to recognize and highlight similar phases. The effect of program comprehension depends on how well the source code developer named its methods. In Table 5.6, we can notice that similar phases have names that could reflect their basic functionality. If the methods are poorly named, users may not understand the relationship between phases.

### 6.4.2 Data Capacity of Web Application

Although we reduce the number of the event to be compared in our phase detection workflow, the number that after phase clustering is not reduced. This means that the JSON file our program produced has all the events that exist earlier. Due to the data capacity of the web browser, the visualization will get choppy after the number reached its limit. For larger trace files, techniques of enabling web browsers to render larger amounts of data are needed.

### 6.4.3 Inapplicable Circumstances

We employ *tree edit distance* on this project because it describes the similarity between trees. While this solution is working with most of the cases, there are some extreme cases where the distance does not represent the phase similarity correctly. For example in Figure 6.2, since function $A$ called function $B$, $C$ and $D$ multiple times, they should be considered similar to the second tree. However, our solution will not consider them similar due to their large tree edit distance.

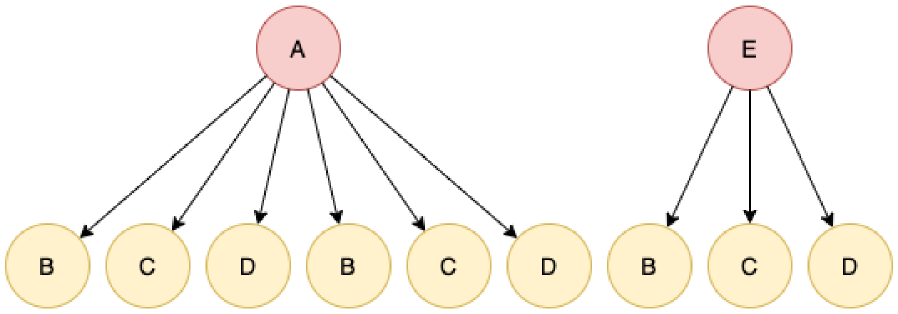


Figure 6.2: Limitation of tree edit distance in phase clustering

# Chapter 7

# Related Work

Dynamic analysis is an approach to analyze the data gathered from a running program, which can provide users with the information of programs' run time behavior and help them to better comprehend the software. The picture of dynamic analysis can range from function-level to high-level architectural views [34, 37, 39]. Trace analysis and visualization are typically used for lower-level analysis. This kind of analysis can often give the user more information about how the program is running but is limited by the scalability issues. In this chapter, we summarize other work that is related to this project.

## 7.1 Trace Compaction and Abstraction

The massive size of the event traces is one of the challenges that hinder the comprehension process. To address these scalability issues, many approaches for trace compaction and abstraction have been proposed.

Pattern summarization is the main approach for reducing the size of the traces. Watanabe *et al.* [40] proposed to find phase in object-oriented programs based on the objects' creation and destruction. Zaidman *et al.* [41] separate the event trace into event clusters by using a heuristic approach. Other trace compaction techniques are offered by Reiss and Renieris [33] and Hamou-Lhadj *et al.* [20, 19, 21]. They grouped the events that appear repeatedly to reduce the total number of events.

Another approch to reduce the size of event traces is by using metric-based filters. This approach filtered out events based on the metric we set. Hamou-Lhadj *ed al.* [18] proposed a way to filter out low-level components and keep the high-level components, resulting in a much smaller execution trace. The work of Cornelissen *ed al.* [11] also indicates that limiting the number of stacks depth can be an effective way to address the problem.

## 7.2 Execution Trace Visualization

Visualization techniques are a popular approach for conveying program information for software comprehension. Multiple visualizations have been developed for dynamic analysis to provide different types of properties of the software system.

Many visualization tools organize events by time and call depths [14, 23, 31]. This kind of visualization is suitable for forming the mapping between source code and execution trace. To reveal more runtime behavior, Cornelissen *et al.* [10] and Palepu *et al.* [26] implement visualizations that shows the repeated patterns. The patterns and phases that were present were detected in the earlier process. The studies indicate that phases and sub-phases are nested in the event traces hierarchy.

David Lo and Shahar Maoz [24] also layered the execution trace to find different levels of granularity and present them to the user as a real-time sequence chart. Different from Palepu's work, their hierarchical structure is derived from the package structure and the real-time sequence chart.

Other visualization tools focus on providing high-level functionality of the program. They the phases and behaviors are shown in these visualizations are usually independent and have no hierarchical nature. For example, Alimadadi *et al.* [2] mapped the low-level behavior to a higher-level behavioral model by examining the runtime behavior of web applications. Wim De Pauw and Steve Heisig [13] visualized the high level of abstraction that distilled from system behavior.

# Chapter 8

# Conclusion and Future Work

## 8.1 Conclusion

In this paper, we proposed a new approach for creating hierarchical phase abstraction of an execution trace and created a visualization to present the phase information based on the flame graph.

The workflow of our approach contains several steps. The new approach first converts event traces to a *layerMap* data structure. Then we perform a duplicate event detection to reduce the number of the total events. In phase clustering, we utilize the *tree edit distance* for determining the similarity between trees. We also designed a series of steps to reduce the phase detecting time while maintaining the accuracy. We consider two trees as one similar phase if the distance is under the threshold value.

We also introduced a visualization that helps developers to receive clearer information about the event traces and the phases we detect earlier. It has several interactive features that enable uses to explore the event traces in different hierarchical levels. The flame graph also shows the outline of the execution and the running time of each function call to users directly.

The quantitative analysis of three Java projects verifies that this project can detect similar phases in event traces successfully. It also verifies the reasonableness of the steps in Figure 3.6 to shorten the processing time. The results indicate that while most of the phases are coming from the method of the same name, there are still a small number of events with different root names were grouped into one phase.

The case study of visualization on Jackson demonstrates the visualization's ability to reveal the system's run time behaviors and its ability to provide execution information at multiple levels of granularity to users.

Finally, we discussed the improvements and optimization of this project over our previous one in terms of running time, phase detection algorithm, and visualization implementation.

## 8.2 Future Work

### 8.2.1 Flame Graph Implementation

Changing the implementation of the flame diagram might be a possible solution to the problem we mentioned in the 6.4.2. Events with short running time can be ignored in the first rendering to make the browser more responsive. For larger JSON files, we might need a server that returns a portion of the data you are interested in, which means events that place outside our interest will not be on the HTML page.

The presentation of phases could also be improved in the future. The current visualization requires that the user "mouse over" the flame graph to find groups. Future work could use color to show all groups without needing to mouse over.

Furthermore, more interactive features could be added to the web project. For example, one possible direction is to add correspondence between visual elements and source code to let users read source code directly on the webpage.

### 8.2.2 Phase Detection Across Multiple Executions

The prior work also identified similar phases across multiple executions. The current project can only analyze traces from one executions. Our goal in the future is to update the project to support the identification of common subtrees that occur in multiple executions.

### 8.2.3 Multi-threaded Handling

At this moment, our approach works best for single-threaded applications. This project only provides a preliminary solution for handling multi-threaded programs. We consider each thread as a standalone output and perform phase clustering within the thread. The visualization also visualizes events of different threads separately. This may be a viable solution at this stage, but the vision is to achieve similar phase detection across threads. The visualization could also be updated to shows similar phases across threads on one page.

### 8.2.4 Process Automation

In the future, we hope to add new code to make the wiring of the steps in this approach more automated. At present, the three steps of this project: (1) obtaining trace file; (2) phase detection; and (3) using JSON file to generate visualization are grid-independent. The user needs to manually import each file into the appropriate location and run the program to proceed to the next step. We want to implement an automated system that allows analysis and visualization of programs with minimal effort.

### 8.2.5 User Study

Finally, we will design a user study in the future to further verify that our phase detection workflow and visualization implementation can help developers more effectively familiarize themselves with the dynamic behaviors of the source code.

# Bibliography

[1] spideruci/blinky-core. `https://github.com/spideruci/blinky-core/wiki`.

[2] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding javascript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 367–377, 2014.

[3] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435, 2002.

[4] T. Ball. The concept of dynamic analysis. In *Software Engineering—ESEC/FSE'99*, pages 216–234. Springer, 1999.

[5] O. Benomar, H. Sahraoui, and P. Poulin. Detecting program execution phases using heuristic search. In *International Symposium on Search Based Software Engineering*, pages 16–30. Springer, 2014.

[6] M. Bostock, V. Ogievetsky, and J. Heer. D$^3$ data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[7] Brendan Gregg. Flame graphs. `http://www.brendangregg.com/flamegraphs.html`, 2012.

[8] Brendan Gregg. Java in flames. `https://netflixtechblog.com/java-in-flames-e763b3d32166`, 2015.

[9] A. BUTLER and K. YOSHIMOTO. Large scale semantic representation with flame graphs. 2015.

[10] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. Van Wijk, and A. Van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 49–58. IEEE, 2007.

[11] B. Cornelissen, A. Van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 213–222. IEEE, 2007.

[12] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[13] W. De Pauw and S. Heisig. Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*, pages 143–152, 2010.

[14] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Software Visualization*, pages 151–162. Springer, 2002.

[15] Y. Feng, K. Dreef, J. A. Jones, and A. van Deursen. Hierarchical abstraction of execution traces for program comprehension. In *Proceedings of the 26th Conference on Program Comprehension*, pages 86–96, 2018.

[16] C. M. D. S. Freitas, P. R. G. Luzzardi, R. A. Cava, M. Winckler, M. S. Pimenta, and L. P. Nedel. On evaluating information visualization techniques. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '02, page 373–374, New York, NY, USA, 2002. Association for Computing Machinery.

[17] B. Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[18] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 112–121. IEEE, 2005.

[19] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 181–190. IEEE, 2006.

[20] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55, 2004.

[21] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu. Challenges and requirements for an effective trace exploration tool. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 70–78. IEEE, 2004.

[22] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):249–267, 2008.

[23] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 320–329. IEEE, 2006.

[24] D. Lo and S. Maoz. Mining hierarchical scenario-based specifications. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 359–370. IEEE, 2009.

[25] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance. `http://tree-edit-distance.dbresearch.uni-salzburg.at`, 2018.

[26] V. K. Palepu and J. A. Jones. Revealing runtime features and constituent behaviors within software. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 86–95. IEEE, 2015.

[27] M. Pawlik and N. Augsten. Rted: a robust algorithm for the tree edit distance. *arXiv preprint arXiv:1201.0230*, 2011.

[28] M. Pawlik and N. Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)*, 40(1):1–40, 2015.

[29] M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[30] H. Pirzadeh, A. Agarwal, and A. Hamou-Lhadj. An approach for detecting execution phases of a system for the purpose of program comprehension. In *2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications*, pages 207–214, 2010.

[31] S. P. Reiss. Jive: visualizing java in action demonstration description. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 820–821. IEEE, 2003.

[32] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the third international workshop on Dynamic analysis*, pages 1–6, 2005.

[33] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 221–230. IEEE, 2001.

[34] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 13–22. IEEE, 1999.

[35] S. Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.

[36] W. S. Sarle. Algorithms for clustering data, 1990.

[37] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.

[38] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th international symposium on Software visualization*, pages 133–142, 2010.

[39] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. *ACM SIGPLAN Notices*, 33(10):271–283, 1998.

[40] Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 8–14, 2008.

[41] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 329–338. IEEE, 2004.

[42] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.