Università della Svizzera italiana

Faculty of Informatics

Bachelor Thesis

May 30, 2017

Data Analysis for Performance Annotations

The (optional) subtitle

Irene Jacob

Abstract

Identifying and resolving performance issues in large software systems can be a difficult task. In particular, it is difficult to know to what extent a piece of code affects the overall performance of a system by simply analyzing the source code. It is even harder to understand why that happens. In order to support software engineers in these tasks, we want to gain more information about the performance characteristics of a piece of code. In traditional performance profiling, one measures and associates aggregate performance metrics (e.g., execution time) with various parts of the code (e.g., methods). In this project we aim to go a bit further by relating performance metrics of specific executions with relevant features of the input or the state of the system within those executions (e.g., relating the size of the input with the execution time). We then perform a statistical analysis of the performance and feature data, which might lead to further analysis of the code and possibly more indicative features. The ultimate goal of this analysis is to formulate synthetic performance annotations for the system and its components.

Advisor
Prof. Antonio Carzaniga
Assistant
Daniele Rogora

Advisor's approval (Prof. Antonio Carzaniga): Date:

1 Introduction

1.1 Large Software systems and their performance issues

Maintaining large software systems is difficult. A common problem in software engineering is maintaining a good quality for the performance of a system. For simple algorithms, we can logically deduce the space and time complexity. Thus if the algorithm's performance behaves [weirdly] at run-time, it may be relatively easy to locate the cause of this performance issue and solve it. This is not the case in large software systems with multiple large linked sections. Thus we need a different approach in finding the cause of performance issues in large software systems. This is done through program analysis...

Static program analysis is ...

Dynamic program analysis is...

Thus performance testing can be used effectively to...

Profiling is one of the most commonly used instrumentation techniques to maintain the execution time of a section of code or a system as a whole [...]

This information in itself provides us great insight into the inner workings of the system. However, simply looking at the data collected to see if it meets expected performance requirements (and later discarding this data) is useful/ not enough. In this paper, we aim to go further with the data collected, by finding correlations with features of the piece of code instrumented.

1.2 Motivation

Performance problems in large software systems can be difficult to resolve. In particular, identifying a performance issue in the source code of a large system can be an arduous process, with little guarantee of success. The idea of performance annotations is to provide valuable information regarding the performance characteristics of a piece of code. For example, a performance annotation might relate the running time or space complexity of a function to, say, the size of an input data structure. This information can then be used to optimize a system. With proper analysis, the data received from these annotations could reveal the parts of the system causing performance problems.

1.3 Goal

There are a number of potential uses of performance annotations. Namely:

1. They can be used as performance assertions, to ensure that a particular function follows its performance requirements. In particular, an annotation may serve as an oracle for tests to detect performance problems.
2. Annotations may also provide useful information in debugging performance problems. For example, a performance annotation may provide information about the memory usage of a function. This information could reveal a possible memory leak, which may otherwise have been difficult to identify by simply examining the code.
3. They can also be seen as a design tool, where systems are designed and implemented to meet certain performance requirements. The annotations of a method, combined with the structure of the code of the system in which this method is used, might allow the developer to deduce the overall performance of the system.

The goal of this project is to derive performance annotations from the execution of a system. In essence, this amounts to recording and analyzing relevant data logged by instrumenting a subject system.

1.4 Project Description

In this project we will instrument a subject system, based on some workload or application scenarios. For example, we could use profiling as an instrument to track the runtime of some function in the system. The performance results received would be recorded and later analyzed using various statistical techniques, and perhaps machine learning. In particular, we will attempt to find a correlation between the performance results and some independent variables (for example, the size of the thread pool). The correlations found would then help derive performance annotations for the subject system (for example an optimal thread pool size).

2 General Description

2.1 Using DiSL

We have two jvms: the server and the jvm where we run the system that we want to instrument. We start the server and give it the instrumentation classes we want to run (from the Manifest file) and then we start a jvm, with the plugin that sends classses to be instrumented to the server. We run the program we want to instrument and analyse on it. Since we have the compiled version of the systems to be instrumented, we simply have to run the jar file containing the compiled classes.

2.2 Basic techniques used to instrument methods and find features

Instrumenting a system can be a time consuming process, if we hace to look at the source code and understand how each method or object interacts with the rest of the system, in order to pinpoint the reason why a section of the systems performance is affected by something such as the size of an array. This mammoth task becomes impossible for large systems, where subtle interactions between two objects in a method might cause the memory leak that you are inverstigating. Therefore, understanding how the system works in depth is not an option. Instead, we use the steps described below to find interesting features of the system and how they relate to the execution time of the section of code we are investigating.

We begin by looking at popular methods of the system, i.e. methods that are called most often when we run the system with some input. The reason for this is simple: if a method that is called many times has a small performance issue, it may ultimately pose a bigger performance issue in general than some method with a bigger lag which is only called a few times, simply because of the number of times this method is called. To count which methods are called most often, we use the following very basic instrumentation technique:

After exiting a code region marked in the scope of the DiSL snippet (which at this point would be as general as possible for the system), we record the Full name of the method/code region (with the scope/path to its class) in a Map where the name of the method is a key and the value is the count of how many times we have called this method so far, which is incremented each time we see the same method. [At the end], we sort the method names by value in descending order and print the results. It should be noted that we are not interested in methods that are not called very often, because the lack of sufficient data points would mean that we would not be able to definitively say whether or not a feature of the code truly affects the execution time of that code region.

```
@After(marker = BodyMarker.class, scope = "org.sunflow.*.*")
static void afterMethodExit(MethodStaticContext msc){
   BasicProfiler.addMethod(msc.thisMethodFullName());
}
```

[Example of DiSL code snippet]

The popular methods provide us insight into interesting scopes for further analysis. Next, we look at one such scope, and calculate the pearson corellation coefficient values for methods in this scope. We look at simple features initially such as collection sizes, value of integer parameters or the length of a string parameter. [explain the process] [explain why execution time]

The next step is where we extract the data, look at the source code of the method we are instrumenting, check to see if the feature is more subtle than we initially thought and possibly do further analysis on the method and create graphs to show the relation between the feature and the execution time.

Popular methods - scope - correlations - extracting data - graphs - analysis

2.3 Features

We mentioned earlier that once we find promising scopes, we search for methods that show a good correlation between their execution time and some feature of that method. In particular, we look at the following six features to begin with: length of a string parameter, value of an integer parameter, the method receiver, size of a collection parameter, size of an array parameter, and an unknown feature. The unknown feature is used mainly when we look at the source code, and are trying to find a feature with a parameter of unknown type in the particular method. Similarly, we have to look at the source code to see if there are any hidden features in the method receiver. We will look at some graphs which show these features and their correlation to the execution time of the method in the case studies that follow.

3 Case Studies

3.1 Dacapo Benchmarks

3.1.1 About the system

3.1.2 Why I selected this system as a case study

3.1.3 Lucene - Luindex

Apache lucene is a text-search engine library written in java. There are two benchmarks dedicated to lucene in the daCapo benchmark suite: luindex, which uses lucene to index a set of documents and lusearch, which uses lucene to do a search for keywords. This benchmark is more interesting because we can find features for methods that are perhaps more interesting with regards to indexing than searching and vice versa.

We begin by looking at some popular methods:
table of interesting scopes we found using the data from this popular method search:
Table 1 shows methods of lucene that were called most often when running the luindex benchmark. (Appendix ?) Let us look at the function setTermBuffer in the Token class. [Preliminary analysis based on FeatureValueCorrelation.java]...

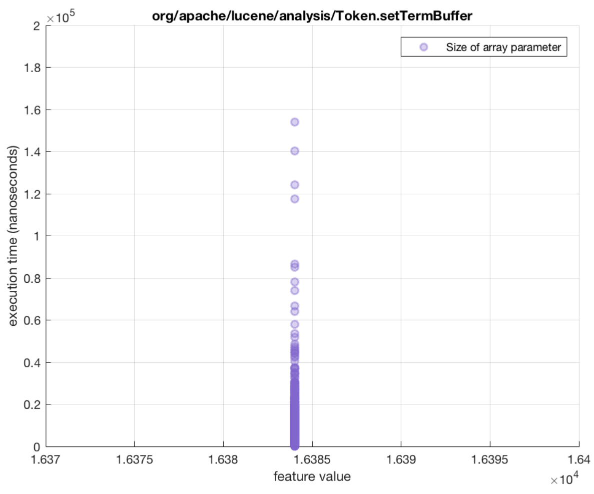
Method name	**Method scope**	**Number of times method was called**
initTermBuffer	org/apache/lucene/analysis/Token	11543371
termLength	org/apache/lucene/analysis/Token	6458130
termBuffer	org/apache/lucene/analysis/Token	5078621
writeByte	org/apache/lucene/store/BufferedIndexOutput	4272541
writeVInt	org/apache/lucene/store/IndexOutput	3246925
readVInt	org/apache/lucene/store/IndexInput	2977629
readByte	org/apache/lucene/store/BufferedIndexInput	2144989
equals	org/apache/lucene/analysis/CharArraySet	1528222
writeVInt	org/apache/lucene/index/TermsHashPerField	1395025
setTermBuffer	org/apache/lucene/analysis/Token	1381976
⋮	⋮	⋮

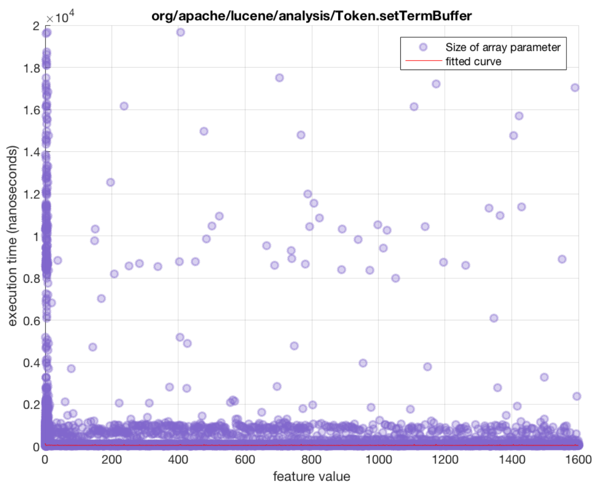
Table 1. Caption of the table

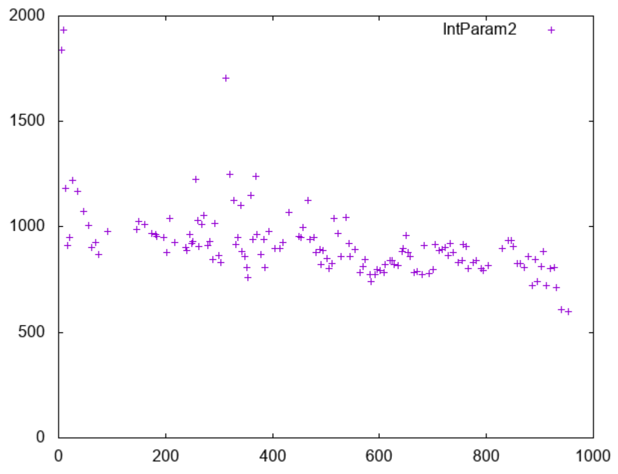
There are 3 basic features for this method: the size of the array parameter, and the value of 2 integer parameters. (Then we searched for methods that show a good feature value correlation:)

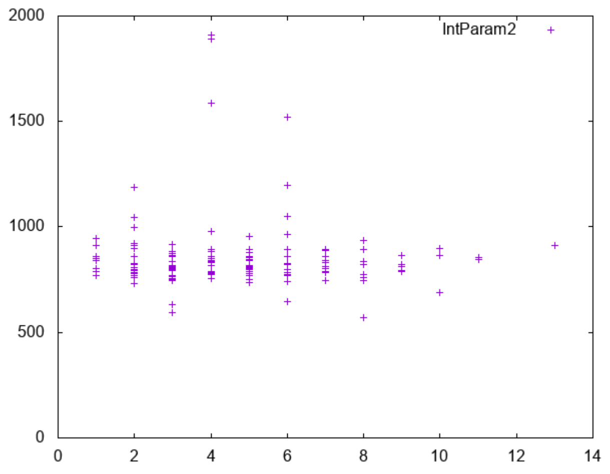
The following is the signature of the method, which tells us what these parameters are

```java
  public final void setTermBuffer(char[] buffer, int offset, int length) {
      termText = null;
      char[] newCharBuffer = growTermBuffer(length);
      if (newCharBuffer != null) {
        termBuffer = newCharBuffer;
      }
      System.arraycopy(buffer, offset, termBuffer, 0, length);
      termLength = length;
  }
```









Clearly the array graph is not very interesting because the feature does not show us anything about the execution time. The graph of the integer parameter shows us that there is something interesting however, the feature is too random for us to be able to decisively say anything. Note that there are a lot more data points in this graph as well. This is because, we recorded 2 integer parameter values for each execution time, so it looks [heavier]. Therefore, we looked into the two integer parameters individually to see if they would give clearer graphs. The result is 2 graphs which both show a constant feature-execution time pattern. [how did u make this graph? - Record results - Profile with feature]

However a constant pattern can be rather boring, so let us look at the feature value correlation to give us more interesting graphs.

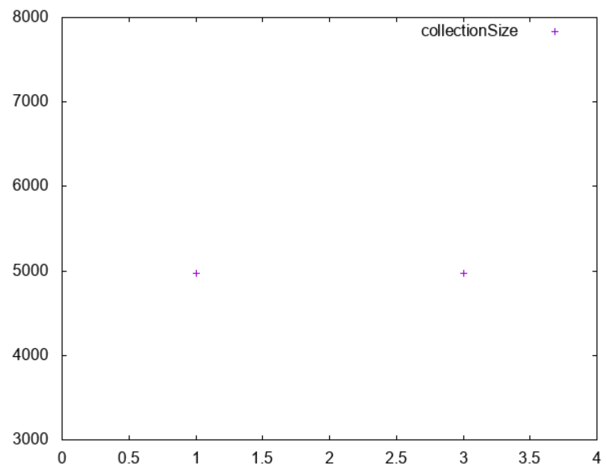
We then used the print_value method to look at promising method, and identify the feature which would give a graph with good correlation: Table of graphs with promising correlations their feature types, and their correlation **Table 2** shows how to insert tables in the document. Given below is one such graph with a 1.0 correlation:

Method name	**Method scope**	**PCC value**
initTermBuffer	org/apache/lucene/analysis/Token	11543371
termLength	org/apache/lucene/analysis/Token	6458130
termBuffer	org/apache/lucene/analysis/Token	5078621
writeByte	org/apache/lucene/store/BufferedIndexOutput	4272541
writeVInt	org/apache/lucene/store/IndexOutput	3246925
readVInt	org/apache/lucene/store/IndexInput	2977629
readByte	org/apache/lucene/store/BufferedIndexInput	2144989
equals	org/apache/lucene/analysis/CharArraySet	1528222
writeVInt	org/apache/lucene/index/TermsHashPerField	1395025
setTermBuffer	org/apache/lucene/analysis/Token	1381976
⋮	⋮	⋮

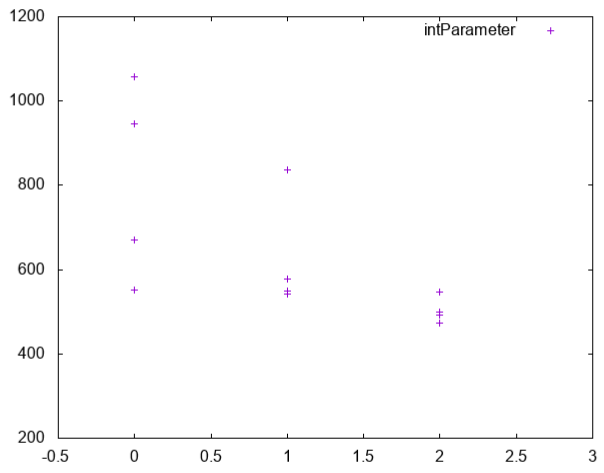
Table 2. Caption of the table

org/apache/lucene/index/LogMergePolicy.findMergesForOptimize
PCC value of 0.9999979750249041

Initial feature analysis shows that Collection size could be a promising feature.



[Graph with a fairly simple feature value relation]
org/apache/lucene/index/SegmentInfo.hasSeparateNorms



[Graph with multiple parameters of the same feature type]
already done above

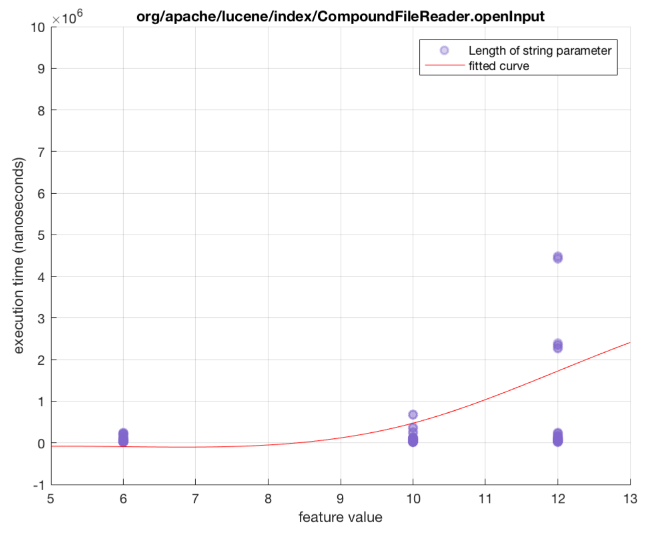
[Graph with decent features, but not enough variety in their values]

[Graph with receiver as feature type]

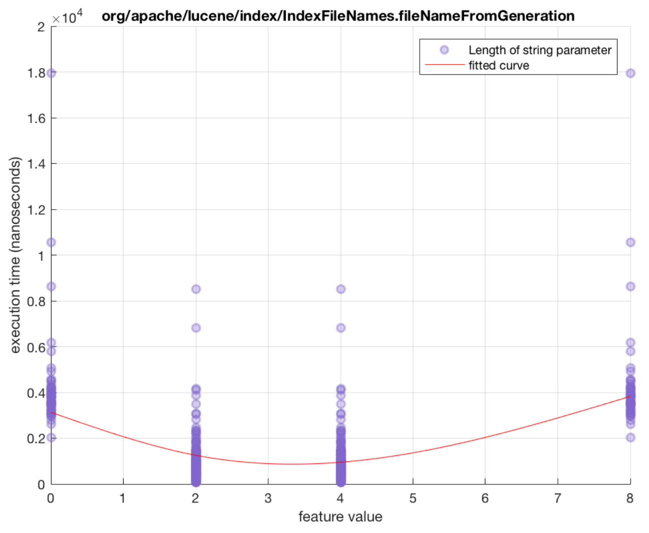
3.1.4 Lusearch

What were the promising methods in lusearch?

graph of the same method, same feature, - how does it change depending on the benchmark?



org/apache/lucene/index/IndexFileNames.fileNameFromGeneration



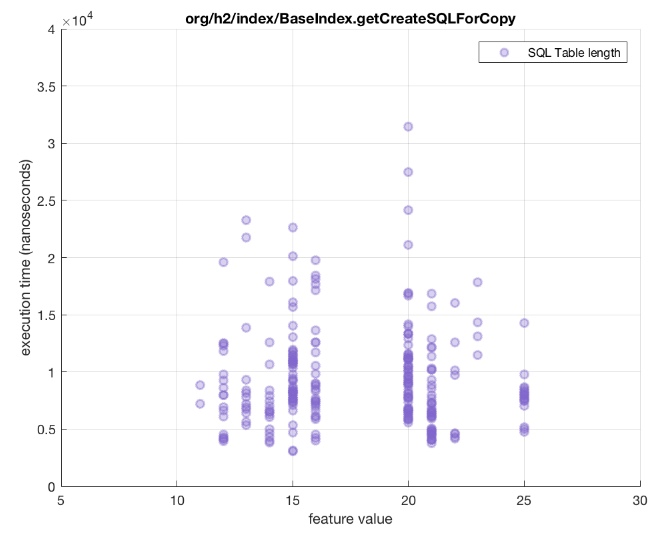
...

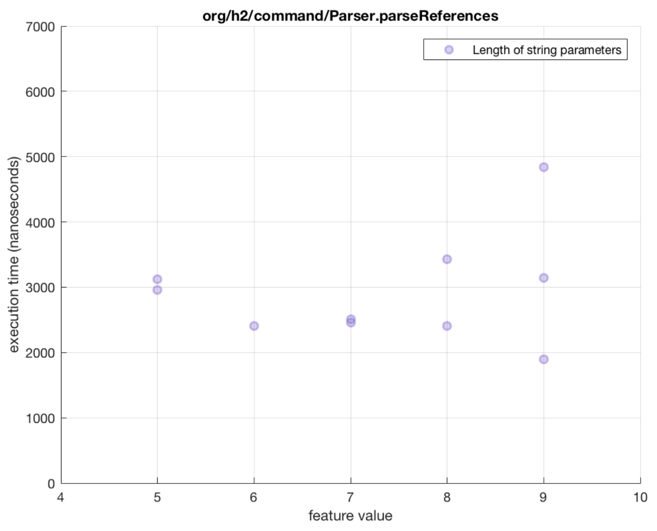
3.1.5 H2

H2 is a database management system written in java. The daCapo benchmark for this system gives a number of transactions in a model banking application as an input.

[scopes from popular methods]

[promising methods, their feature types and correlation values]





[graphs of promising methods and their analysis]

[Graph with unknown feature type]

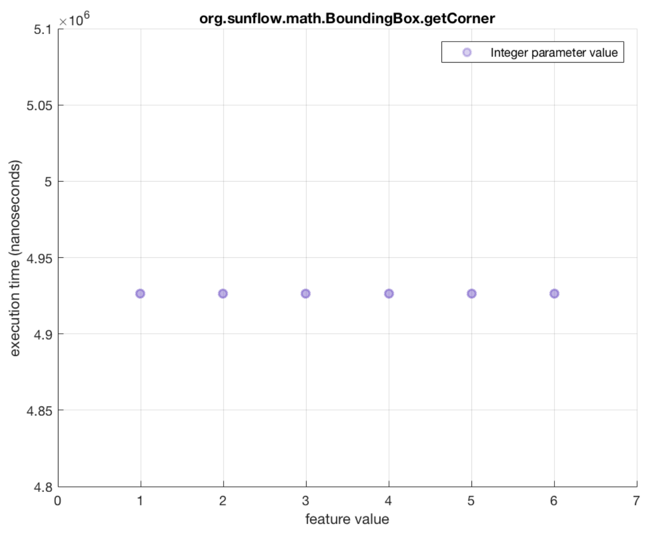
3.1.6 Sunflow

why choose sunflow: what is sunflow? possibility of more features [scopes from popular methods]

[promising methods, their feature types and correlation values]

[graphs of promising methods and their analysis]

org.sunflow.math.BoundingBox.getCorner



3.1.7 Issues

DaCapo is not a performance benchmark suite:
Lack of Data Points:
At this point, I would like to [clarify] that DaCapo was in fact, not the first system that I instrumented (Closure compiler was the first). The reason I presented the case stuies in this order and not the chronological order can be seen from the graphs presented above. Dacapo benchmarks provides its own input for the benchmarked systems. Since DaCapo is not a performance benchmark suite (it is a . . .), the methods that are called most often (most data points) with the given input, do not have a great correlation. The graphs that do show an interesting pattern, dont have enough data points for us to convincingly say that there is a pattern. Thus, I decided to revisit the closure system (and check the ScalaBench benchmarks) since I could provide my own input for closure(. . .).

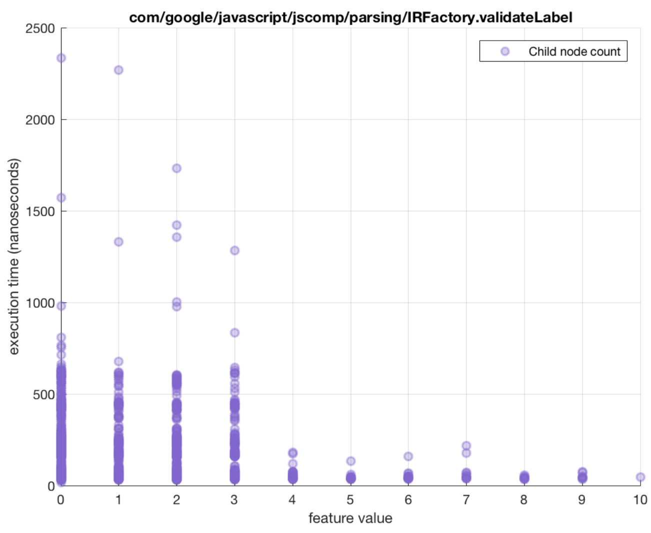
3.2 Closure Compiler

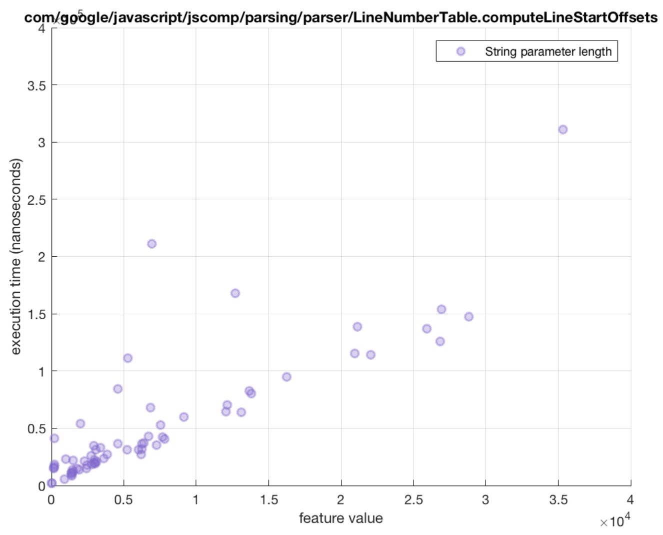
3.2.1 About the system

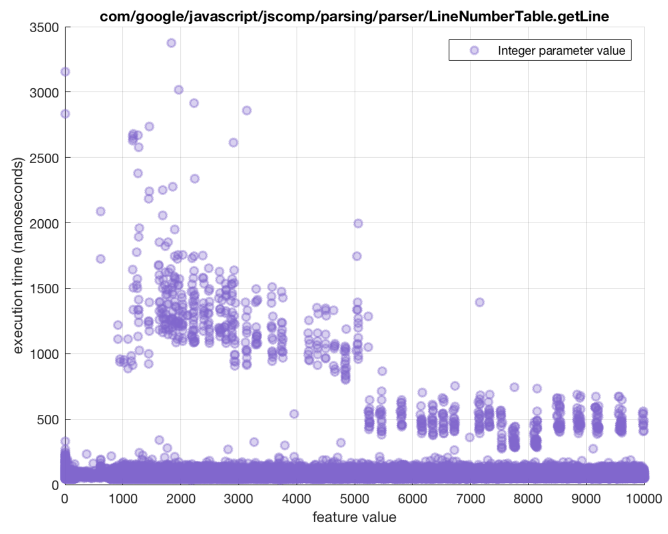
3.2.2 Why I selected this system as a case study

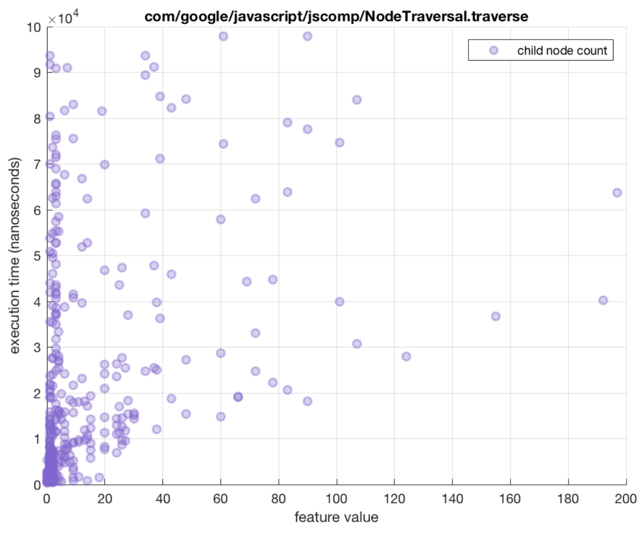
As mentioned earlier, I provide the input files for this system. I used javascript files taken from various websites to try and replicate real world scenarios.

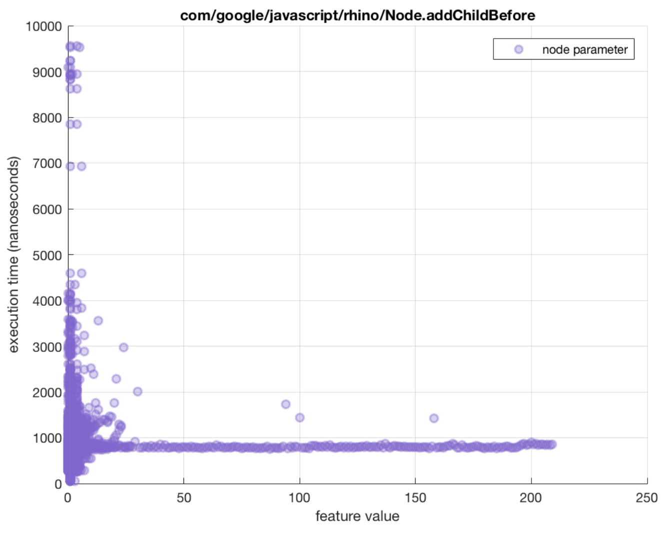
[scopes from popular methods]
[promising methods, their feature types and correlation values]
[graphs of promising methods and their analysis]

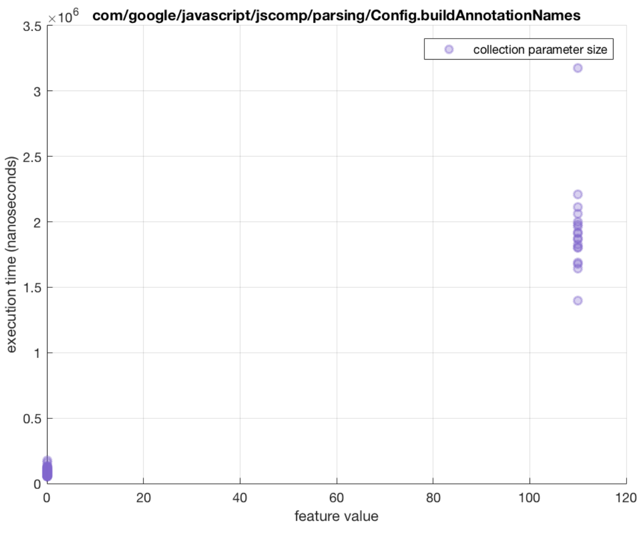


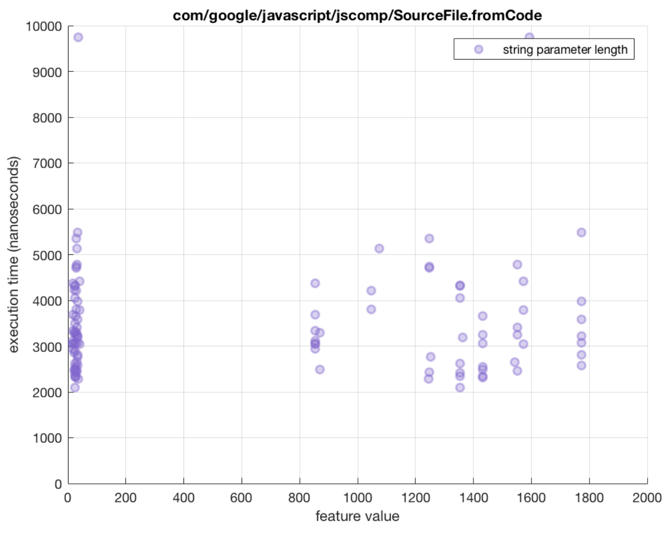


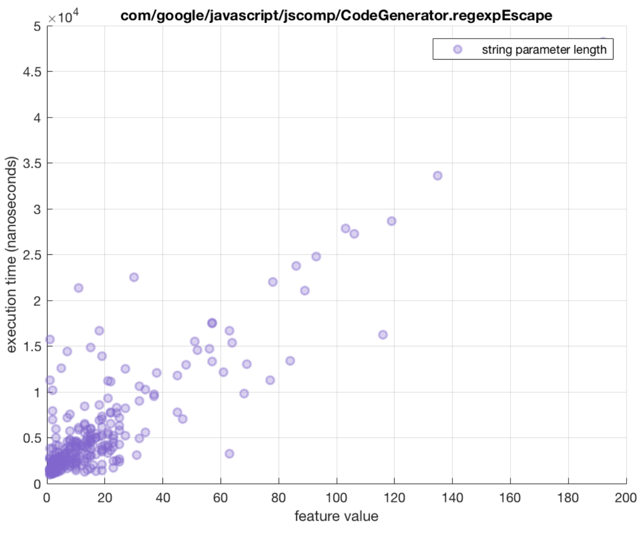

com/google/javascript/jscomp/parsing/parser/LineNumberTable.getLine
Integer parameter value
execution time (nanoseconds)
feature value

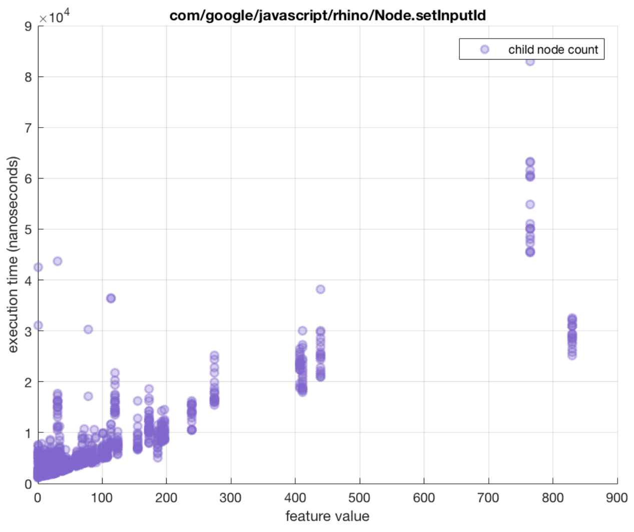

com/google/javascript/jscomp/NodeTraversal.traverse
child node count
$\times 10^4$
execution time (nanoseconds)
feature value


com/google/javascript/rhino/Node.addChildBefore
node parameter
execution time (nanoseconds)
feature value


com/google/javascript/jscomp/parsing/Config.buildAnnotationNames
collection parameter size
$\times 10^6$
execution time (nanoseconds)
feature value


com/google/javascript/jscomp/SourceFile.fromCode
string parameter length
execution time (nanoseconds)
feature value


com/google/javascript/jscomp/CodeGenerator.regexpEscape
string parameter length
$\times 10^4$
execution time (nanoseconds)
feature value



3.3 ScalaBench Benchmarks

3.3.1 About the system

3.3.2 Why I selected this system as a case study

The reason I chose this benchmark is because it is somehow similar to dacapo. However, it uses different systems and therefore different inputs. I wanted to see how graphs would look for a different benchmark suite.

FOR EACH BENCHMARK AS IN DACAPO:

[scopes from popular methods]
[promising methods, their feature types and correlation values]
[graphs of promising methods and their analysis]

4 Results

5 Conclusion

References