Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 20, 2017

Data Analysis for Performance Annotations

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 this 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. Traditional performance profiling provides a good starting point where one measures and associates aggregate performance metrics (such as execution time) with various parts of the code (e.g., methods). In this project we aim to go further. We want to relate performance metrics of specific executions with relevant features of the input or the state of the system within those executions (for example, relating the size of the input with the execution time).

We start by instrumenting one or more systems and focus on some part of the code that affects performance in a significant way, for which we then identify relevant features of the input (or state). Next, we run the system with various inputs, and collect performance and feature data. We then perform a statistical analysis of that 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:

Contents

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 behavior affects the overall system performance in a significant way, 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...

[Work in progress]
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 instrument a subject system, based on some workload or application scenarios. In practice, we use instrumentation techniques developed for, and commonly used in profiling to measure and record the running time of some function in the system. We then collect the measured performance results (i.e., the dependent variables) together with a set of corresponding input or state features (i.e., the independent variables) for an off-line analysis. For this off-line analysis we can use various statistical techniques, including machine-learning techniques and classifiers. 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 use DiSL, an aspect oriented programming (AOP) language, to instrument the systems in our case studies. Fundamentally, we use DiSL as an instrumentation tool to create an execution time profiler for the dynamic program analysis of selected systems. Given below is a basic description of DiSL's instrumentation process:

We have two JVM processes, one for the observed system and the other one is the instrumentation process (DiSL seperates the two in order to reduce the burden on the observed system while measuring performance metrics). We start the DiSL server and pass any instrumentation classes we want to run to the server. At this point DiSL initialises any markers, argument processors, static contexts etc, declared in the instrumentation classes. Then we start a JVM process with a plugin that sends the classes being instrumented to the DiSL server. Finally, we run the observed system with some input and gather data about the features and the chosen performance metric.

2.2 Basic techniques used to instrument methods and find features

Pinpointing performance problems is difficult. In particular, it is already difficult to gain a good understanding of an entire code base, and it is even more difficult to understand the interactions between the various components of the system, and the feature of the input that trigger or control such interactions. This approach is not practical for large software systems, where subtle interactions between two objects in a method might cause the memory leak that you are inverstigating. Thus, we need a better process to identify performance issues in a system. In order to deal with this complexity, we first focus on hot spots in the code, and then look for interesting and promising features that correlate with the performance of those hot spots. We now detail this high-level process.

We begin by looking at popular methods of the system, that is, methods that are called most often when we run the system with some input. The reason for this is simple: even if the performance issue in a method is small, if the method is executed many times, the issue could escalate to major performance issues for the system. To count which methods are called most often, we use the following very basic instrumentation technique:

```
1          @After(marker = BodyMarker.class, scope = "org.sunflow.*.*")
2          static void afterMethodExit(MethodStaticContext msc){
3             BasicProfiler.addMethod(msc.thisMethodFullName());
4          }
```

FIGURE 1. DiSL code snippet for logging method names to find popular methods.

After exiting a code region marked in the scope of the DiSL snippet (which at this point would be very general), we record the full name of the method 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. This values is incremented each time we see the same method. In 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.

The popular methods found, provide insight into interesting scopes for further analysis. The next step would be to look at one such scope, and calculate the pearson corellation coefficient (PCC) values for methods in this scope. Initially, we look at simple features such as collection sizes, value of integer parameters or the length of a string parameter. Then, we pick methods with promising PCC values, and for features of these methods, we collect the value of the feature and the value of the execution time of that method, which is our chosen performance metric. We chose execution time as the metric because it was fairly simple and allowed us to focus on the relation between the feature and the metric more. Instead, having memory used as a performance metric would have led to complications due to java's garbage collection. After collecting data with basic features, we look at the source code of the scope to try and more unique and indicative features such as isolating the values of a specific parameter, or looking at the method receiver.

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: the length of a string parameter, the value of an integer parameter, some featuer of the method receiver, the size of a

collection parameter, the size of an array parameter, and any unknown feature. The unknown feature is used mainly to identify features in non-primitive type parameters of a emthod and requires closer inspection of the source code of the method. In the case studies which follow, we will investigate some methods which show a good correlation between their values for the selected feature types and the execution time of that method.

3 Case Studies

3.1 DaCapo Benchmarks

3.1.1 About DaCapo

The DaCapo benchmark suite was one of the first benchmark suites created to [evaluate] Java and is still used by many researchers for Java benchmarking. In particular, we look at version 9.12 of the benchmark suite (released in 2009). The systems in this suite are all open-soure systems written in Java. This is particularly beneficial for us, as we can look at the source code of the system we instrument to try and find any hidden features. Input for each benchmark is provided in three sizes (small, default, large). The small and large sized inputs are not provided for all systems and thus we use the default input size for the sake of regularity. In particular, we look at three systems (four benchmarks): lucene (luindex and lusearch), h2 and sunflow.

3.1.2 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. We are interested in looking at both of these benchmarks to see how the system performs, depending on which functionality of the system we observe.

The first step in our instrumentation process is to look at the "popular" methods in the system. We run the system with the given input and log the names of the methods that were called most often along with the total number of times that the method was called with the given input. The reason we do this is because a lack of data points would cause problems in either identifying a pattern within the feature-value pairs, or if there is a pattern but not enough data points, we would not know the uncertainty in our results and therefore, could lead to false conclusions about the performance of the method. **Table 1** shows ten methods of lucene that were called most often when running the luindex benchmark.

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. Popular methods in lucene called during an execution of the luindex benchmark

From **Table 1**, we can see that "org/apache/lucene/analysis/Token" is quite a popular scope. Therefore we look into this scope further to calculate the PCC value for methods called from this scope with the features we have chosen to explore. The results from this [process] will point out methods in this scope with a good linear correlation between the values of the features explored and the performance metric we measure (execution time in nanoseconds).

Table 2 shows methods in the scope "org/apache/lucene/analysis/Token" that have an absolute PCC value greater than 0.6.

The PCC values in **Table 2** tell us that there are no features to be explored in this scope, even though many of the popular methods are from this scope. However, the function **setTermBuffer** still stands out, because it definitely has a parameter with some feature-type that we are exploring, and although the initial PCC value shows that the correlation of the feature and the execution time is not very good, we could look further into the method to see if the feature we are looking for is more specific than what is covered by our initial inspection. The basic features in this method were the size of an array parameter and the values of two integer parameters. We then collect the relevant

Method name	PCC value
setEndOffset	-0.007195126852575053
setTermLength	-0.0044103176313241114
reinit	-0.17796321065767187
setTermBuffer	-0.005546296464117803
setType	-0.0013907834015398784
setStartOffset	-0.005177540284207936
setPositionIncrement	0.0
growTermBuffer	0.0028720228569128576

TABLE 2. methods in the scope "org/apache/lucene/analysis/Token" and their PCC values

data (value of chosen feature-type and execution time) and plot the graphs for these two features. **Figure 2** shows these two graphs.

Both graphs in **Figure 2** confirm that the there is no real feature-value correlation as we hypothesised from the PCC value. Clearly the array graph is not very interesting because the feature does not tell us anything about the execution time. Likewise, the graph with the integer parameters as the feature does not show any reasonable pattern. However, we can look at the source code of this method to try and find any interesting feature that our initial search may have missed.

```
1       public final void setTermBuffer(char[] buffer, int offset, int length) {
2           termText = null;
3           char[] newCharBuffer = growTermBuffer(length);
4           if (newCharBuffer != null) {
5             termBuffer = newCharBuffer;
6           }
7           System.arraycopy(buffer, offset, termBuffer, 0, length);
8           termLength = length;
9       }
```

The code snippet above shows us that initially, we graph both the offset and length as two integer parameters (with the same execution time) in the same graph. Perhaps plotting the two parameters seperately as individual features would result in graphs with better feature-value correlations.

Figure 3 clearly shows that merging the two graphs would result in the the graph with integer parameter as a feature that we saw in **Figure 2**. However, unlike what we hoped for, it seems that neither parameter is a feature of this method. There seem to be small areas within each graph that seem to follow a pattern, however the majority of the data-points in both graphs are random and thus we conclude that neither of the parameters are useful features.

3.1.3 Lucene - Lusearch

We look at the constructor of a class called **FieldInfo** in lucene. The method has a PCC value of -0.677 which shows a negative linear correlation. A feature of this method is the length of its string parameter, and **Figure 4** shows the correlation of this feature with the execution time of this method.

Figure 4 tells us that although there are enough data points in this graph, there is limited variety with the independent variable (the length of the string parameter is always one of two values). This means that even though the PCC value might tell us that there is a negative linear correlation between the feature and the performance metric, in reality this information is not very useful for us. All we know from the data collected is an estimate of the execution time for two values of the feature, which shows a negative correlation. We cannot make any conclusion about whether this is a linear relation or not (we cannot estimate the execution time for any length of the string parameter between those two points). Let us take a look at the source code of method to see if there are any other interesting features to be explored.

```
1       FieldInfo(String na, boolean tk, int nu, boolean storeTermVector,
2           boolean storePositionWithTermVector,  boolean storeOffsetWithTermVector,
3           boolean omitNorms, boolean storePayloads, boolean omitTf) {
4           name = na;
5           isIndexed = tk;
6           number = nu;
7           this.storeTermVector = storeTermVector;
```

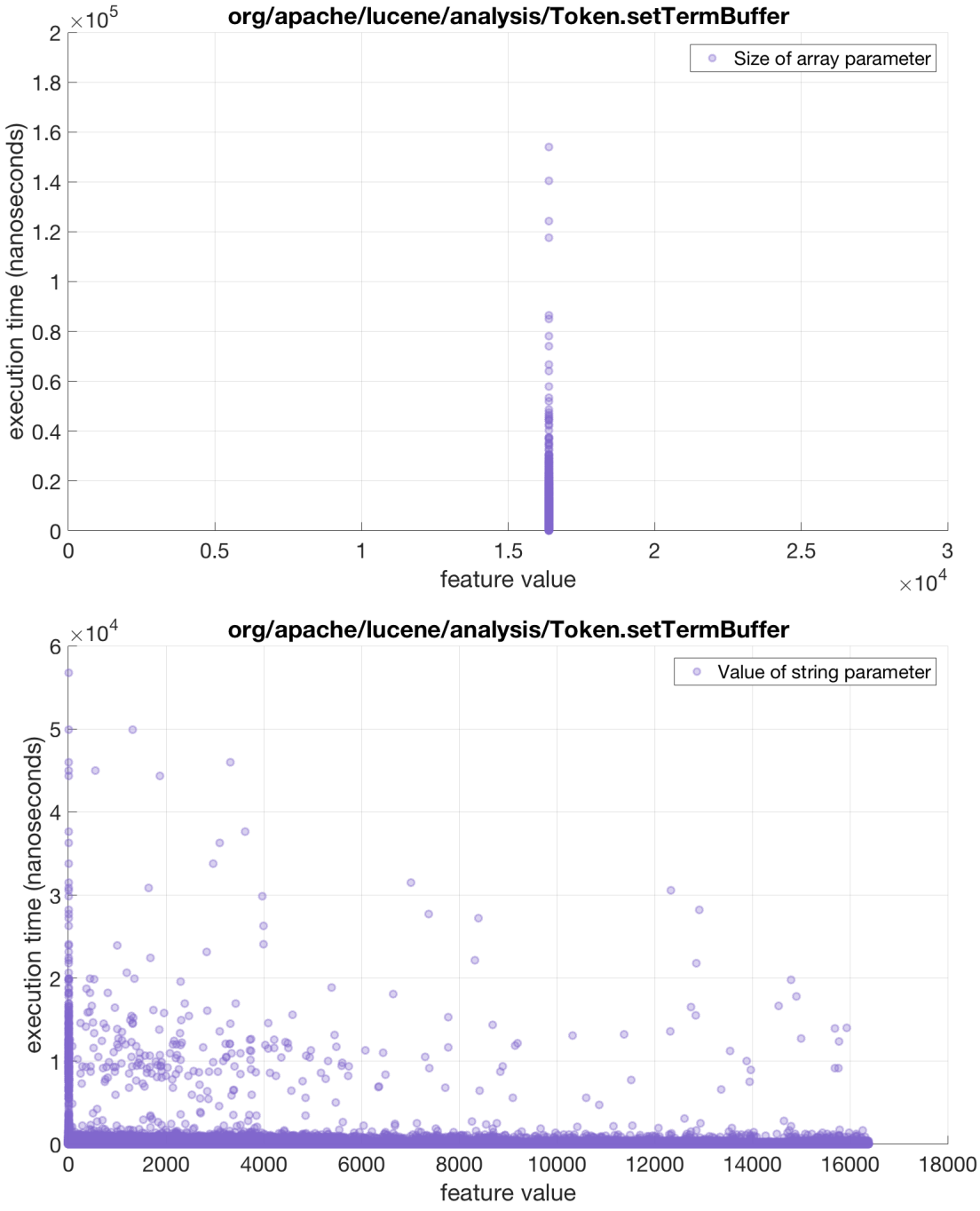

FIGURE 2. Graphs of method setTermBuffer for the feature types discovered in the initial inspection

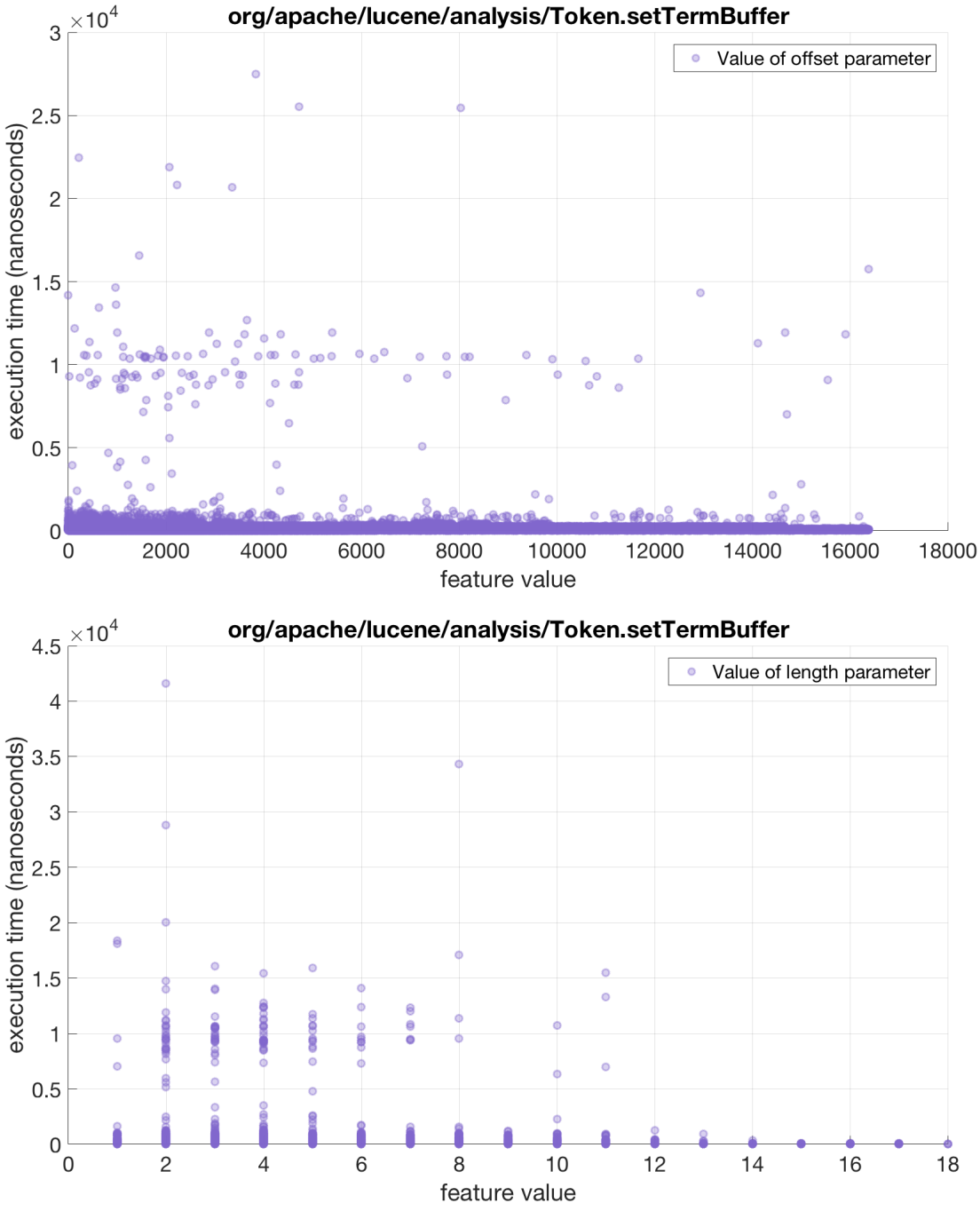

FIGURE 3. Graphs of setTermBuffer with the two integer parameters as seperate features.

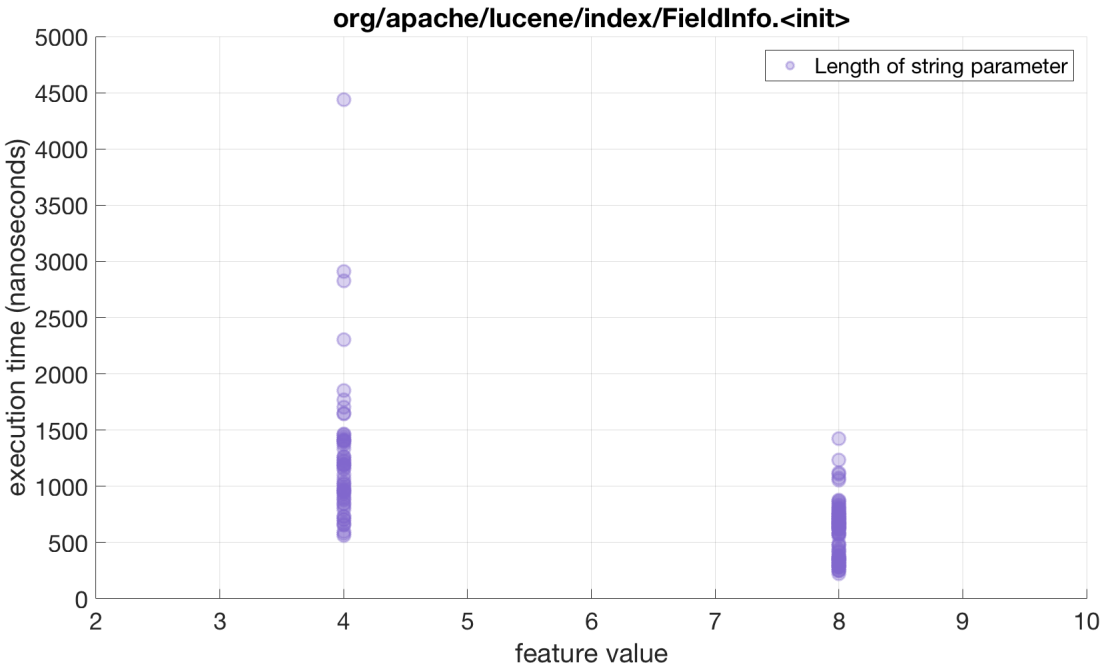

FIGURE 4. Graph of the method FieldInfo with the length of a string parameter as its feature type.

```
        this.storeOffsetWithTermVector = storeOffsetWithTermVector;
        this.storePositionWithTermVector = storePositionWithTermVector;
        this.omitNorms = omitNorms;
        this.storePayloads = storePayloads;
        this.omitTf = omitTf;
    }
```

From the source code of the method, we can see that although there is an integer parameter which could be a possible feature, this constructor method does not do anything particularly interesting. Both the string parameter and the integer parameter are simply stored in some variable and thus are not likely to be very interesting in terms of performance.

3.1.4 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.

Table 3 shows ten methods of H2 that were called most often when running the h2 benchmark.

Method name	Method scope	Number of times method was called
getValue	org/h2/result/Row	293521951
compare	org/h2/util/MathUtils	126119557
compareTypeSave	org/h2/value/Value	125153590
compareTypeSave	org/h2/table/Table	125144577
compareValues	org/h2/index/BaseIndex	125096324
compareSecure	org/h2/value/ValueShort	82418363
compareRows	org/h2/index/BaseIndex	58652798
checkClosed	org/h2/jdbc/JdbcConnection	49874510
get	org/h2/util/ObjectArray	44185377
compareSecure	org/h2/value/ValueInt	43484239

TABLE 3. Popular methods in H2 called during an execution of the h2 benchmark

Let us take a look at the method **getCreateSQLForCopy** in the scope "org/h2/index/BaseIndex". In particular, we want to look into the length of an SQL Table which was a parameter in this method, as the feature-type. This is interesting, because in our initial PCC calculation, we would only record this parameter as an unknown feature type and we would not know what the value of this feature was until closer inspection. Therefore, we would look at this

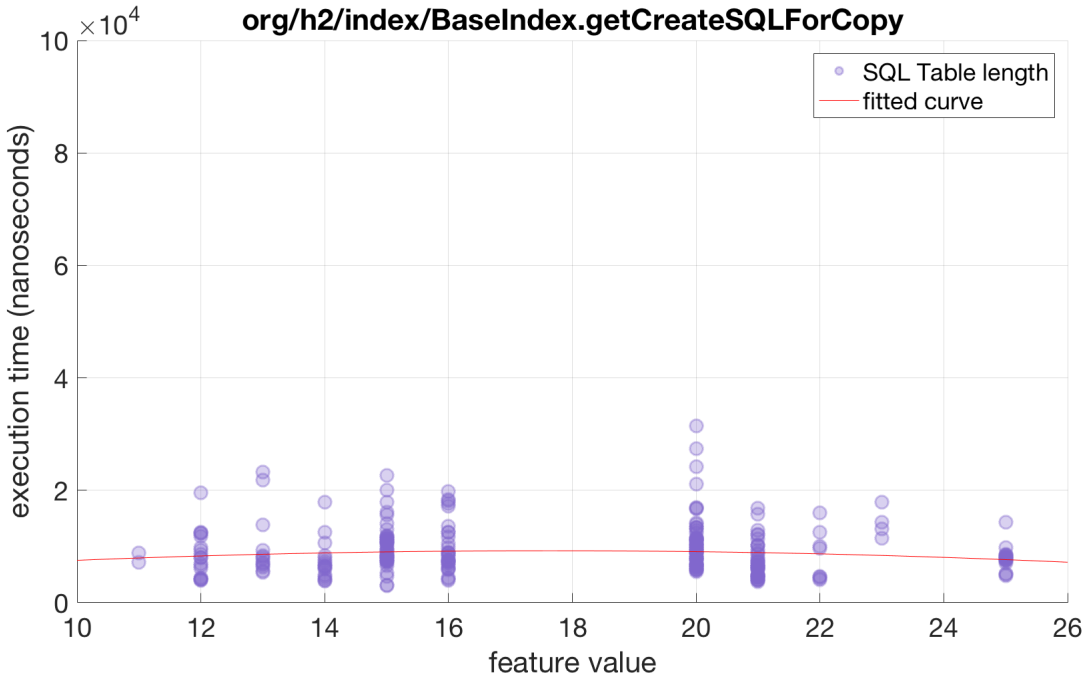

FIGURE 5. Graph of method getCreateSQLForCopy

feature-value pair without first looking at its PCC value. **Figure 5** shows the graph of the this feature-type against the execution times of this method.

Figure 5 shows that there is some correlation between the feature type and performance metric. However, there is a gap in the between $x = 16$ and $x = 20$ for which we do not know the behaviour of the graph. Currently, the fitted curve suggests that there is a quadratic relation between the feature-type and performance metric. However, the lack of data means that there is some uncertainty with these results.

Unknown feature types are some of the more common and more interesting features in such systems as these object parameters are often defined and called in other parts of the system and could reveal more about the way two parts of the system interact.

Let us take a look at another method in the h2 benchmark that which seems to have a quadratic feature-value relationship. **Figure 6** shows the quadratic relation between the length of a string parameter and the execution time in the method **parseReferences**.

Figure 6 shows that although there seems to be a quadratic relation between the feature and the execution time, there are not enough data points to measure the uncertainty in our results.

3.1.5 Sunflow

Sunflow is an image rendering system which uses ray tracing to draw photo-realistic images. Similar to the other benchmarks in DaCapo, sunflow is an open-source system written in java. The reason why we chose this system was because it showed the real world application of performance annotations in a different light. We looked at how performance annotations can be used to understand the performance of large database systems, and text indexing and searching. Now we look at image rendering in computer graphics, which is an area where improving the performance of the system would cause a drastic increase in the demand for the system.

Rendering time is a major bottleneck in many computer graphics projects, especially in game development. Having to render realistic images could take a long time, and so games opt for speed over quality of rendering and use techniques such as rasterisation instead of ray tracing, which is more efficient but the images are not as photorealistic as those created by ray tracing.

After looking at the popular methods in the system for this benchmark, we decided to look further into the scope "org.sunflow.core.∗.∗"

Table 4 shows methods in this scope with a good correlation between its features and the performance metric (execution time in nanoseconds).

Let us take a further look into the constructor of the class **BuildTask**, as this was one of the methods we recognised

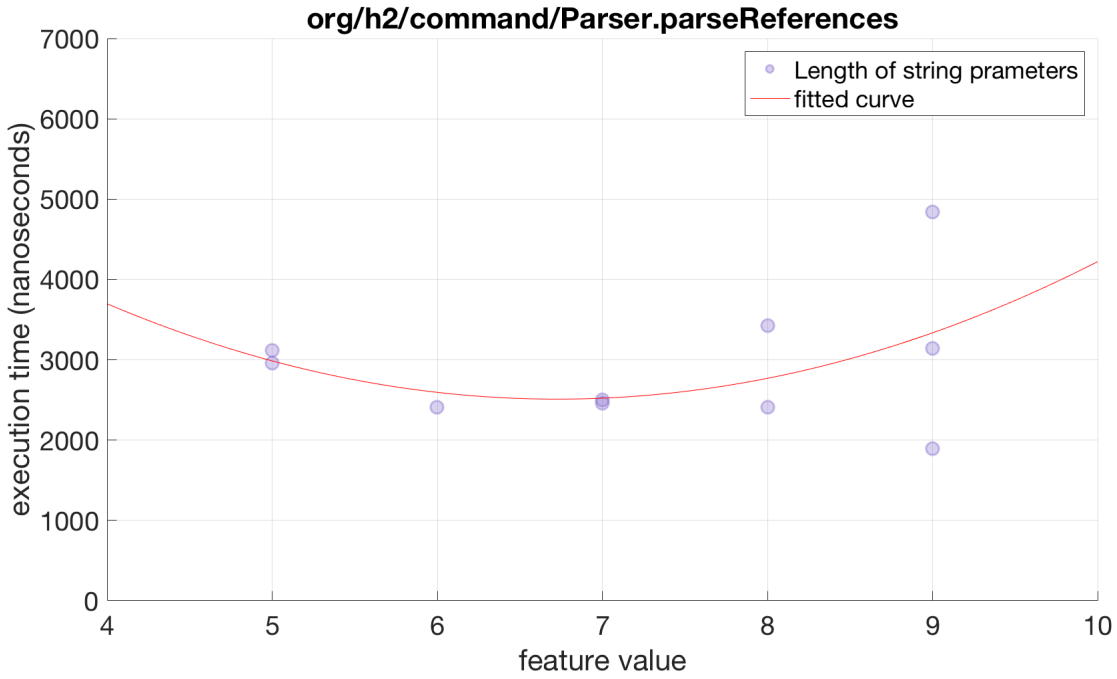

FIGURE 6. Graph of the method parseReferences with the length of a string parameter as its feature-type

Method name	Scope	PCC value
addFloat	org/sunflow/core/ParameterList.addFloat	0.9195098289521854
constructor	org/sunflow/core/ParameterList$InterpolationType.<init>	-0.7859884871860157
addPoints	org/sunflow/core/ParameterList.addPoints	-0.9904143581193274
createNode	org/sunflow/core/accel/BoundingIntervalHierarchy.createNode	-0.880663340905176
getNumPrimitives	org/sunflow/core/InstanceList.getNumPrimitives	-0.6144959888457919
constructor	**org/sunflow/core/accel/KDTree$BuildTask.<init>**	**0.8764541684463605**
getBoolean	org/sunflow/core/ParameterList.getBoolean	0.7637467566304426
subdivide	org/sunflow/core/accel/BoundingIntervalHierarchy.subdivide	-0.9007371578525583

TABLE 4. methods with a good PCC value in the scope "org.sunflow.core.∗.∗"

from the search for interesting scopes. By looking at the source code, we realised that the constructor was overloaded in this class, and that the signature for both methods had an integer parameter called *numObjects*. Therefore, we decided to try and see if this integer parameter would be an interesting feature for this method.

From **Figure 7**, we can clearly see that our intuition (based on the PCC value) was right and that the integer parameter was indeed positively correlated to the execution time of the method. In other words, the higher the number of objects, the longer it took to make BuildTask objects.

3.1.6 Issues

One of the major issues in trying to find features for the systems in the DaCapo benchmark suite was that many of the benchmarks were made of outdated versions of the systems. Thus it was not possible to run all of the benchmarks in the suite with the current version of the JRE, which consequently meant that those benchmarks could not be instrumented. This also meant that finding the source code for the particular version of the system was not always easy, even though DaCapo uses open source systems.

Another issue was that DaCapo was not designed as a performance benchmark. It is a Java benchmark suite and therefore, the input provided for each of the systems is aimed at evaluating Java and not the performance of the system. This means that not all the data collected would be useful in showing interesting relations between the performance of the piece of code and the feature we explore. Since the input of the system is provided in the benchmark suite, this meant that sometimes there was a good feature-value correlation, but a lack of data points meant that we could not measure the uncertainty in the data. Chronologically, Closure compiler was the first system that we instrumented in this project. However, the input for the Closure compiler system was provided by us (we took javascript files from random websites to try and simulate realistic executions in the system). Thus, we present the systems in this order to address this issue and to illustrate the difference in quality of the data collected.

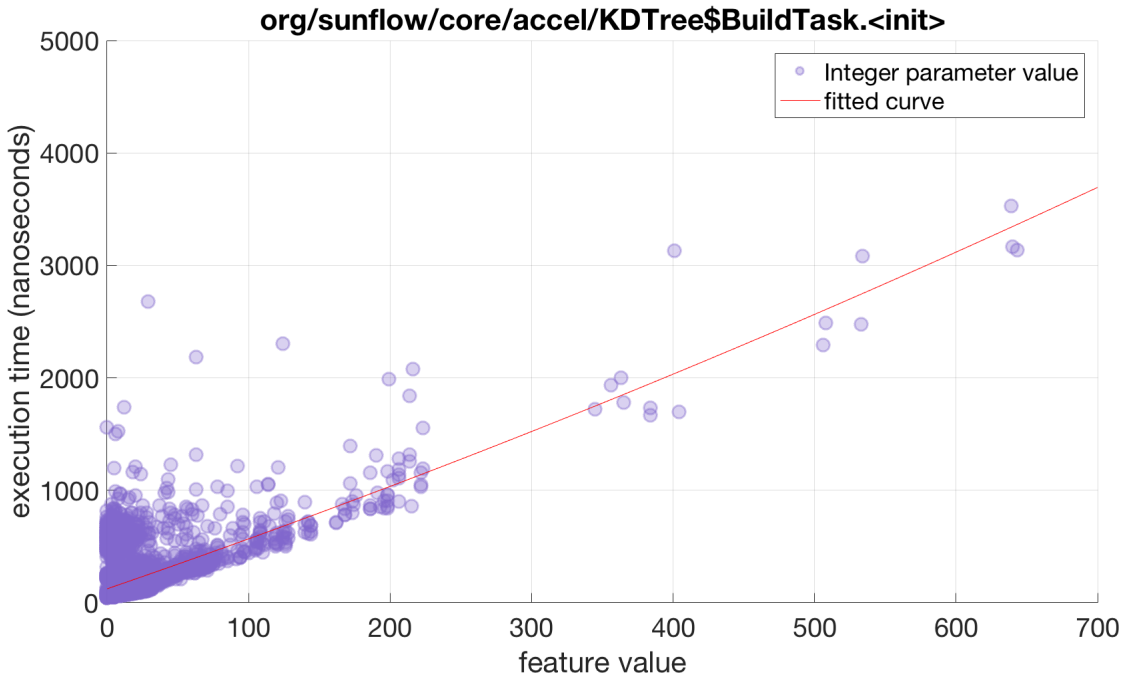

FIGURE 7. Graph of the constructor method from the BuildTask class

3.2 Closure Compiler

3.2.1 About the system

Closure compiler is an open-source project by Google which aims to make Javascript run faster. In particular, we instrument the command-line application, written in Java. The compiler takes a Javascript file and minimises the size of the file and makes it more efficient by removing uneccessary whitespace and dead-code, checks for dubious code and prints warnings for the same. The compiler then outputs the minimised Javascript file which can be used instead of the input file, as the code in the output file would be more efficient. The reason why we look at this system is because, unlike the benchmarks we have seen before, the input for this system was not provided by a third party. We picked the input files from various websites to have a realistic simulation of the methods that the compiler would call. This is interesting because unlike the benchmark suite which shows how a system runs in its best or worst case, by providing our own input, we are building a realistic average-case.

3.2.2 Results

Table 5 shows some of the most executed methods from closure compiler.

Method name	Method scope	Number of times method was called
getToken	com/google/javascript/rhino/Node	10837001
getFirstChild	com/google/javascript/rhino/Node	4200793
getNext	com/google/javascript/rhino/Node	3728802
isValidIndex	com/google/javascript/jscomp/parsing/parser/Scanner	3092498
traverseBranch	com/google/javascript/jscomp/NodeTraversal	2861340
traverseChildren	com/google/javascript/jscomp/NodeTraversal	2747789
createsBlockScope	com/google/javascript/jscomp/NodeUtil	2351386
peekChar	com/google/javascript/jscomp/parsing/parser/Scanner	2245487
getParent	com/google/javascript/rhino/Node	2137254
isFunction	com/google/javascript/rhino/Node	2035733
⋮	⋮	⋮

TABLE 5. Caption of the table

Let us take a look at the scope “com/google/javascript/jscomp/parsing” and methods in this scope that have a good correlation between their execution times and their feature values.

The buildAnnotationNames function looks promising with a PCC value close to 1. Now as we can see from the above graph, the reason why the method has such a good PCC value is because the independent variable is only ever one of two values. The lack of variety in the feature values means that there is not sufficient data for us to make any

Method name	Method scope	PCC value
buildAnnotationNames	com/google/javascript/jscomp/parsing/Config	0.9998530711820874
extractList	com/google/javascript/jscomp/parsing/ParserRunner	0.7001973844053915
setOffset	com/google/javascript/jscomp/parsing/parser/	0.6059009913106439
validTypePredicate	com/google/javascript/jscomp/parsing/TypeTransformationParser	0.6388008461376518
parse	com/google/javascript/jscomp/parsing/ParserRunner	0.60689093292811
toIntArray	com/google/javascript/jscomp/parsing/parser/LineNumberTable	0.8863126616091507
outputCharSequence	com/google/javascript/jscomp/parsing/parser/util/format/SimpleFormat	-0.7121693836707618
computeLineStartOffsets	com/google/javascript/jscomp/parsing/parser/LineNumberTable	0.8038954869182353

TABLE 6. Caption of the table

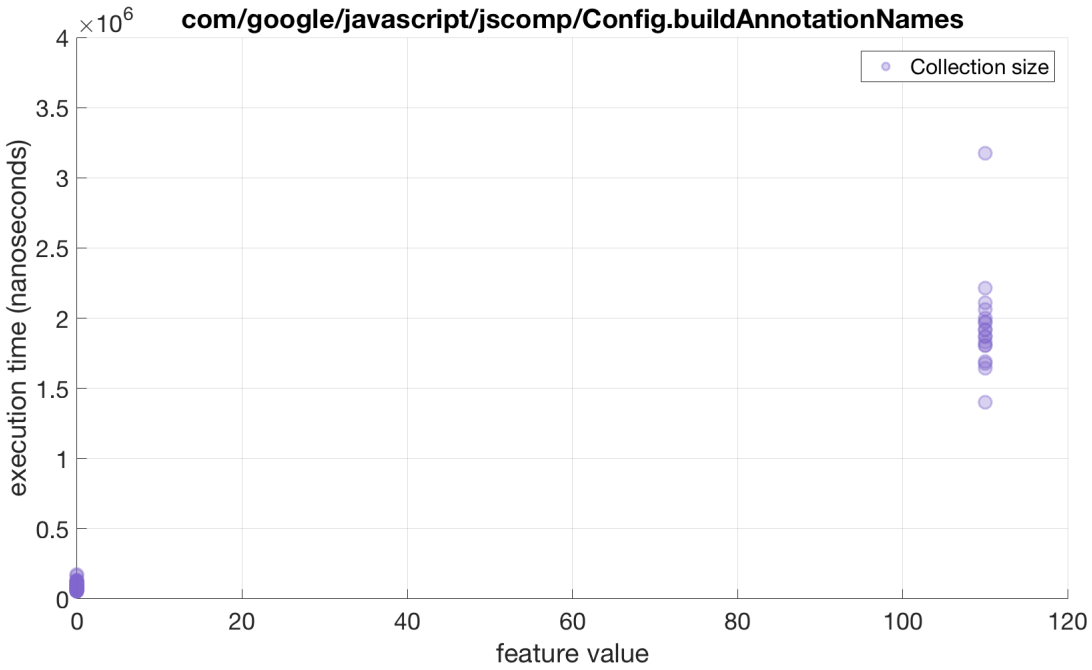

FIGURE 8

detailed analysis of the method. We know that within the interval, there is a positive correlation between the size of the collection and the execution time. However, we cannot say if this is a linear relation or a quadratic relation due to the lack of data points in between these two points. Thus we can retrieve basic information from this graph, however, it is not as useful as we would have liked.

Let us look at the method computeLineStartOffsets next. From **Table 7** we can see that computeLineStartOffsets has a slightly lower PCC value than buildAnnotationNames. However, the graph for computeLineOffsets is much more interesting than that of buildAnnotationNames, because of the variety in the feature values. The PCC values for this method also show a strong positive correlation, much like that of the method buildAnnotationNames. But the variety in the values of the independent variable, gives us a much more detailed picture for this graph. We can clearly see a linear correlation between the length of the string parameter and the execution time of the method.

Let us look at an example of the object receiver being the feature type. The node class in closure compiler provides an excellent example of this.

When we try to look for methods with good PCC values in the scope "com.google.javascript.rhino.Node.∗", the results come back empty. But this class was one of the most popular ones we found in our search for interesting scopes. Thus we look at the object receiver for interesting features.

```
1       Object rec = apc.getReceiver(ArgumentProcessorMode.METHOD_ARGS);
2           if (rec instanceof Node) {
3               Node n = (Node)rec;
4               Measurement m = new Measurement();
5               m.arg_idx = arguments.length;
6               m.ft = Measurement.FeatureType.FT_RECEIVER;
7               m.fv = n.getChildCount();
8               m.value = duration;
9               ProfileExecutionTime.addValue(msc.thisMethodFullName(), m);
```

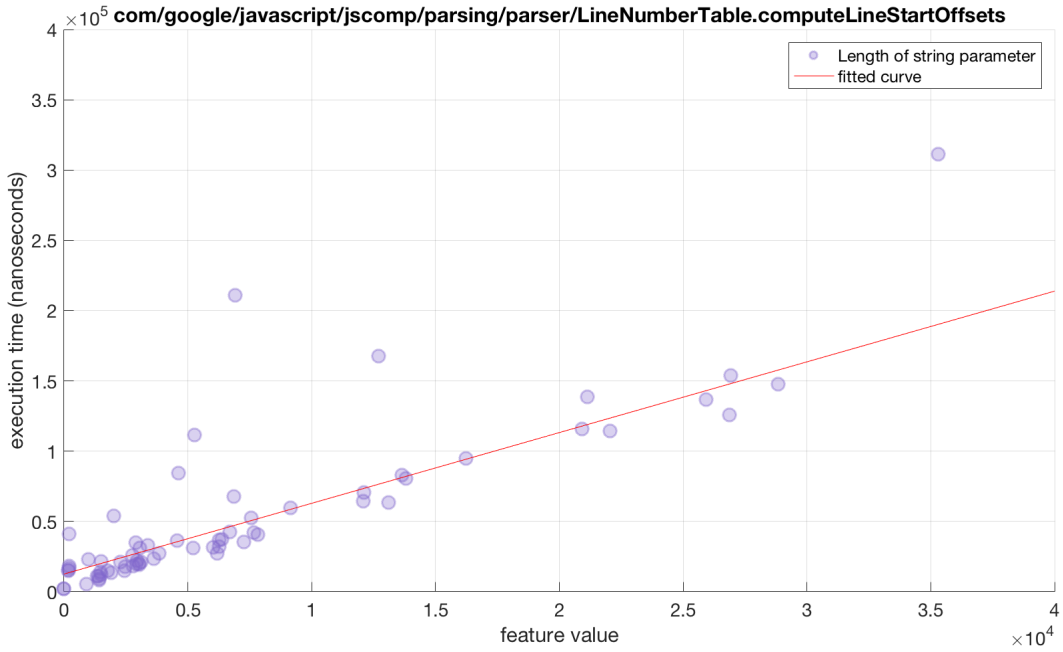

FIGURE 9

```
10          }
```

The code snippet above, shows how we looked at the Node object, and specifically at the child count of a node as a possible feature. **Table 7** shows methods in the scope "com/google/javascript/rhino/Node" that have an absolute PCC value greter than 0.6 after adding the child node count of the receiver object as a feature in the search. setInputId

Method name	PCC value
setInputId	0.9578214065615162
addChildBefore	0.8379390951395574
useSourceInfoIfMissingFromForTree	0.6127997297953337
removeFirstChild	0.6261839263374583
addChildAfter	0.6299331195689717
useSourceInfoFromForTree	0.6879775206073537

TABLE 7. methods in the scope "com/google/javascript/rhino/Node" and their PCC values

seems to be a very promising method, because it is in the list of the 500 most executed methods and with the receiver feature type, it seems to have an excellent positive correlation with the execution time.

The graph clearly reflects our intuition, and a clear linear correlation can be seen. Let us take a look at the source code of the setInputId method.

```
1     public void setInputId(InputId inputId) {
2       this.putProp(INPUT_ID, inputId);
3     }
```

Although we see that the object receiver is being called in this method, it is still not easy to see how the number of children a node object affects the execution time of this fairly simple setter method. This graph shows one of the most fundamental benefits of performance annotations. In particular, it shows us that features of a code region may be more subtle than what can be immediately seen from the source code. The code of the setInputId method seems very simple and does not look like it would have any interesting features at first. [...]

Finally, let us take a look at the method traverse in the scope "com/google/javascript/jscomp/NodeTraversal". The source code of the method is given below.

```
1       public void traverse(Node root) {
2           try {
3             setInputId(NodeUtil.getInputId(root), "");
4             curNode = root;
```

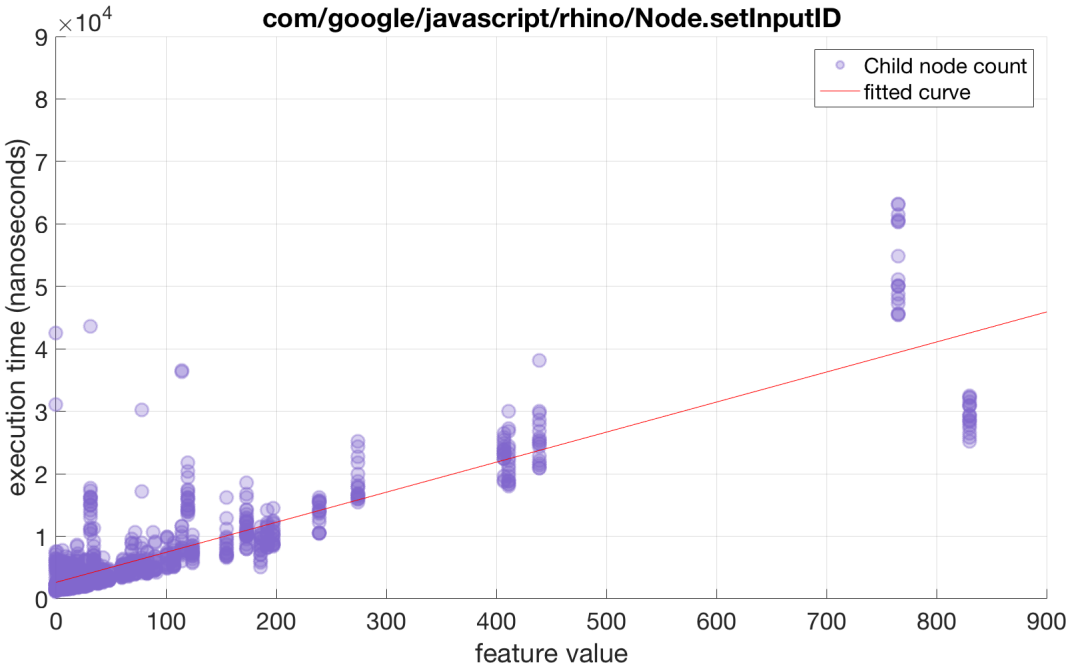

FIGURE 10

```
5            pushScope(root);
6            // null parent ensures that the shallow callbacks will traverse root
7            traverseBranch(root, null);
8            popScope();
9          } catch (Exception unexpectedException) {
10           throwUnexpectedException(unexpectedException);
11         }
12       }
```

We look at the node parameter as a possible feature for this method. In particular, we look at the child count of a node as the feature. Please note that in the method setInputId, the feature was the node object that called the method, in contrast to this method where the node object is passed as a parameter.

At first glance, this graph does not seem to show any useful correlation between the feature value and the execution time. Most of the the data points are clustered near zero and the data points for higher feature values are sparse and do not show any real pattern. However, we are more interested in where the data points are not in this graph. In other words, this graph is interesting because it hints at a possible lower-limit in the execution time, depending on the feature value. The graph does not clearly prove the existance of such a limit, however it does open the question up to whether the number of children a node has, could affect the lower limit of the execution time of that method. This is a plausible theory, but it would require further investigation to be confirmed. We are more interested in the possibility of being able to detect such information from these graphs. This information, if stored as performance annotations could help developers understand the way the system works practically. In particular, this would help a maintainance programmer identify why a method is taking too long to be executed much faster (without having to again profile and collect data and [stab in the dark]) because we provide the meta-data ourselves for the most interesting code-regions in the system.

4 Conclusion

Developers and software engineers are always looking for ways to improve, maintain and monitor the performance of software systems. This is a particularly difficult task when the softare in question is large with many interconnected components. In this project, we look at two case studies (four systems) for which we create and follow an instrumentation process, where we find methods with an interesting feature-value correlation. In particular, we explored some basic features such as string parameters, integer parameters, array sizes e.t.c. We also looked at some subtle features, which were more difficult to identify such as an object receiver or some non-primitive type parameter of a method. Aside from the visual analysis (the graphs with the fitted curves) ajd the value of the pearson correlation

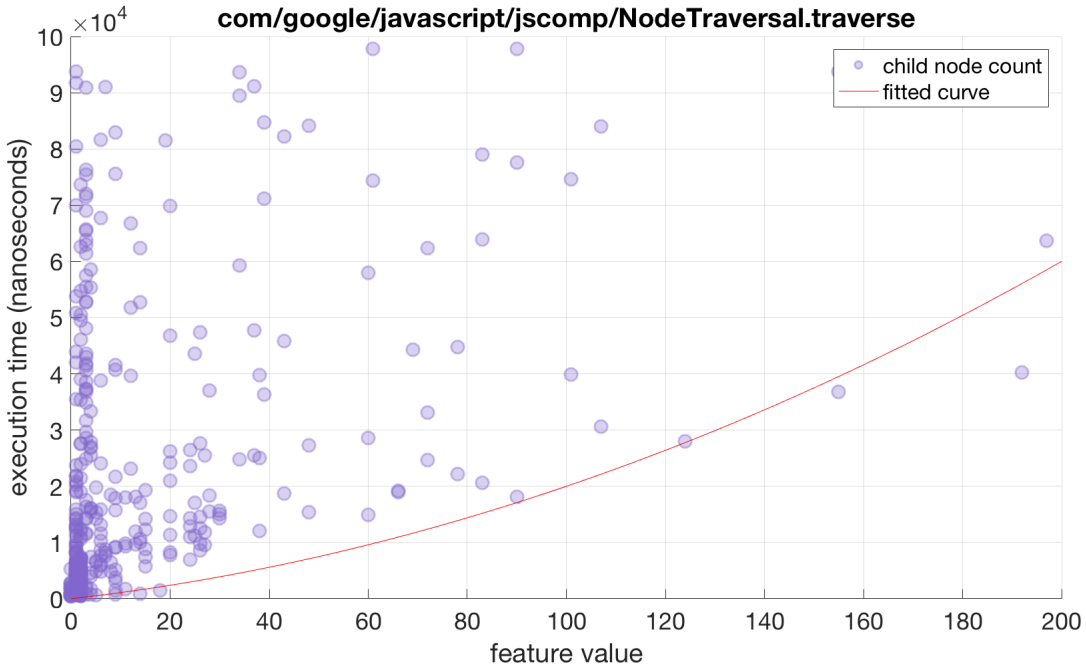

FIGURE 11

coefficient to see the relation between the feature and the performance metric (execution time in nanoseconds) we also look at the source code for some of the methods, to try and relate the feature-value correlation back to the method (that is, to understand the behaviour of the relation with regards to the structure of the method). The case studies presented in this report show one way of collecting meta-data for performance annotations in large software sustems and provides an analysis on the different types of feature-value correlations encountered.

5 Future Work

In future, we can expect similar case-studies with different systems which look into other features or other performance metrics. It would also be interesting to see other instrumentation processes than the one described in this report, which may be able to filter the more interesting scopes and methods of a system in a more efficient manner. In future, we also hope to see more in-depth analysis of the data collected using other statistical techniques and perhaps also machine learning. In particular, it would be interesting to see performance annotations being used in the development and maintenance of a large software system, with specific performance requirement. Another idea would be to create a tool (such as a Java plugin) where the performance of the system is monitored, recorded and analysed during development and then used to simulate and predict the performance in what-if analysis. The tool could also highlight areas of the source code which are being monitored and reflect on whether some recent change has a positive or negative effect on the performance of the piece of code. Using performance annotations is an innovative way to make better use of the results of instrumentation and to go further and use this information to give feedback on the performance-status of the system.

References