Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 6, 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...

[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 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());
}
```

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]

Then, we pick methods with promising pearson correlation coefficient 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 find better/ hidden 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 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 ?)

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

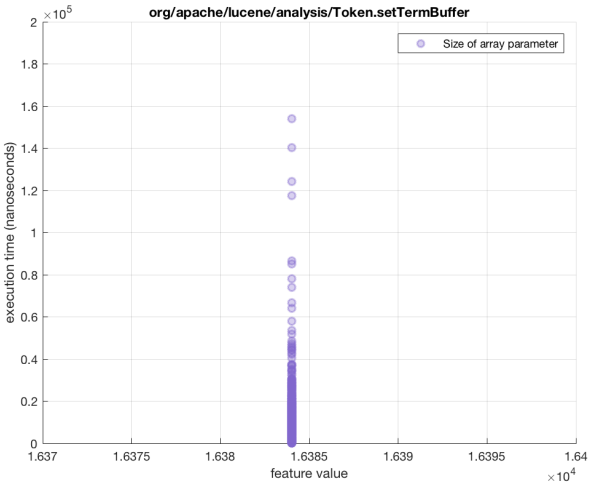
From the above table, we can see that “org/apache/lucene/analysis/Token” is quite a popular scope. Therefore we look into this scope further to calculate the PCC value for methods called from this scope with the features we explore.

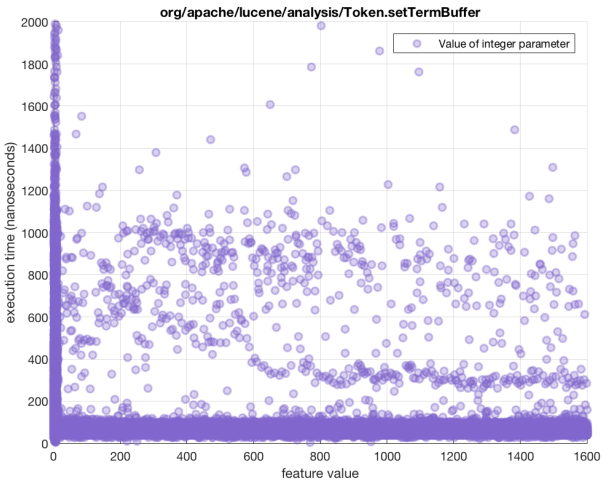
Table 2 shows methods in the scope “org/apache/lucene/analysis/Token” that have an absolute PCC value greter than 0.6.

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

Unfortunately, the pcc values tell us that there are no features to be explored in this scope, even though many of the popular methods are in this scope. However, the function setTermBuffer still stands out, because it definitely has some feature 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 integer parameters. We then collect data and plot the graphs for these two features.

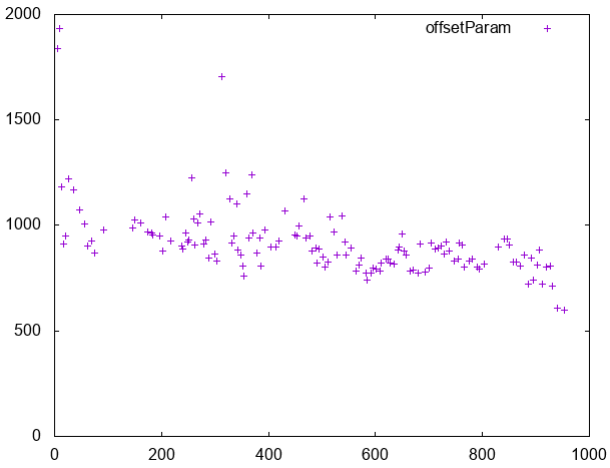


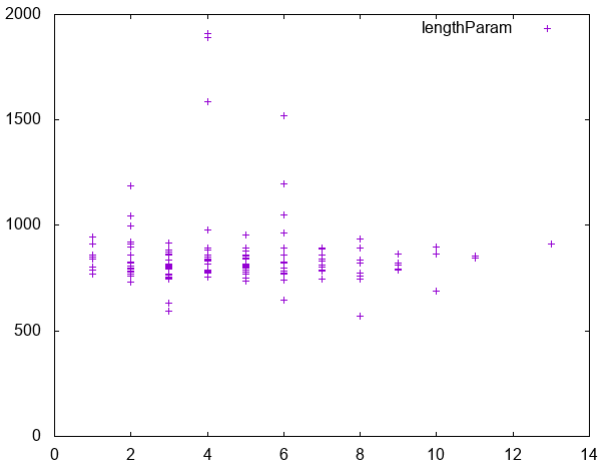


Both graphs confirm the initial results we achieved with the pcc value. Clearly the array graph is not very interesting because the feature does not tell us anything about the execution time. Likewise, the graph with the integer parameters as the feature does not show any reasonable pattern. However, 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 in our initial search for features we take both the offset and the length as two feature values with the same execution time. Perhaps plotting the two parameters seperately as individual features would result in graphs with better feature value correlations.

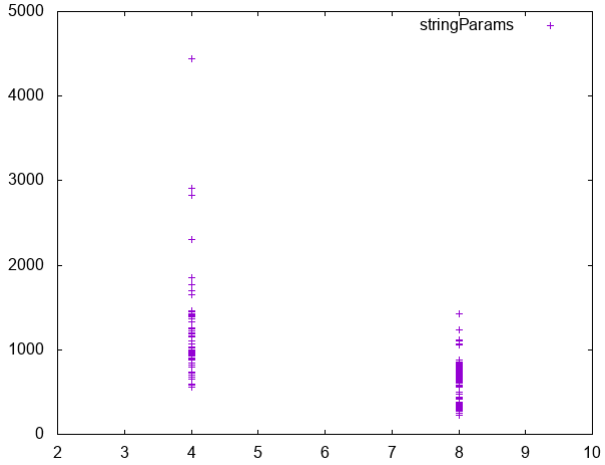




The graphs above clearly show a much better correlation between the feature and the execution time. In fact we calculated the pcc value for the two features seperately, and the [...].

3.1.4 Lusearch

We look at the constructor of a class called FieldInfo in lucene. The method has a pcc value of −0.6772449086839872 which shows a negative correlation. The feature of this method is the length of a string parameter.



This graph above tells us that although there are enough data points, there is limited variety with the independent variable, beause the feature is only ever between two values. Therefore, the pearson correlation coeficient might tell us that this graph shows a negative corellation, however, this information is not very useful for us, because the conclusion we reach is too broad. Let us take a look at this 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;
8           this.storeOffsetWithTermVector = storeOffsetWithTermVector;
9           this.storePositionWithTermVector = storePositionWithTermVector;
10          this.omitNorms = omitNorms;
11          this.storePayloads = storePayloads;
12          this.omitTf = omitTf;
13      }
```

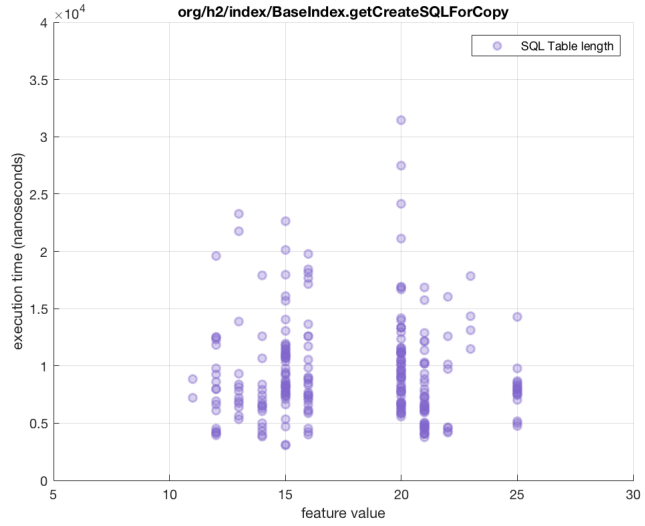
The source code of the method confirms that the constructor of this class is not a very interesting method to observe in terms of performance.

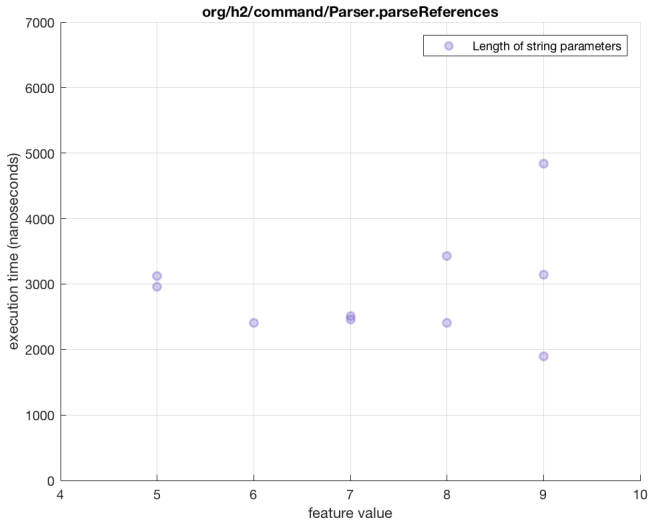
3.1.5 H2

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

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 I chose this system was because this system 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 usually tend to use a technique called 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, I decided to look further into the scope "org.sunflow.core.∗.∗"

Table 3 shows methods in this scope with a good correlation between its features and the performance metric (execution time in nanoseconds).

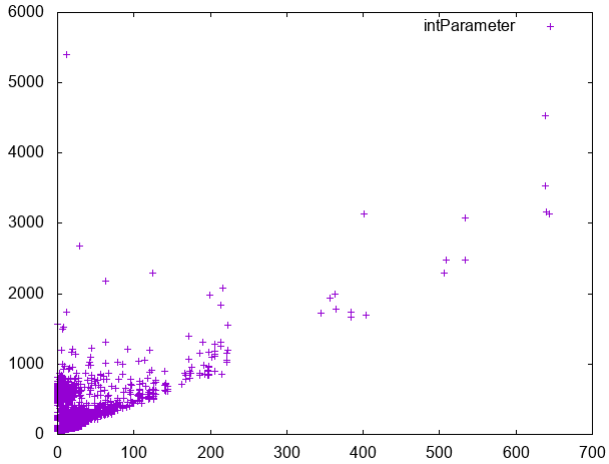
Let us take a further look into the constructor of the class BuildTask, as this was one of the methods I recognised from the search for interesting scopes. By looking at the source code, I realised that the constructor was overloaded in this class, and that the signature for both methods had an integer parameter called $numObjects$. Therefore, I

Method name	Scope	PCC value
addVectors	org/sunflow/core/ParameterList.addVectors	-1.0
access	org/sunflow/core/light/TriangleMeshLight.access$400	-1.0
getString	org/sunflow/core/ParameterList.getString	-0.9010669568495452
addFloat	org/sunflow/core/ParameterList.addFloat	0.9195098289521854
constructor	org/sunflow/core/InstanceList.<init>	-1.0
radix12	org/sunflow/core/accel/KDTree.radix12	1.0
constructor	org/sunflow/core/ParameterList$InterpolationType.<init>	-0.7859884871860157
aaDepthToString	org/sunflow/core/renderer/BucketRenderer.aaDepthToString	-1.0
getPoint	org/sunflow/core/ParameterList.getPoint	-1.0
getFloat	org/sunflow/core/ParameterList.getFloat	-0.6100019097800228
addPoints	org/sunflow/core/ParameterList.addPoints	-0.9904143581193274
createNode	org/sunflow/core/accel/BoundingIntervalHierarchy.createNode	-0.880663340905176
constructor	org/sunflow/core/light/TriangleMeshLight$TriangleLight.<init>	-1.0
getNumPrimitives	org/sunflow/core/InstanceList.getNumPrimitives	-0.6144959888457919
constructor	org/sunflow/core/accel/KDTree$BuildTask.<init>	0.8764541684463605
getBoolean	org/sunflow/core/ParameterList.getBoolean	0.7637467566304426
subdivide	org/sunflow/core/accel/BoundingIntervalHierarchy.subdivide	-0.9007371578525583

TABLE 3. methods with a good pcc value in the scope "org.sunflow.core.∗.∗"

decided to try and see if this integer parameter could be a possible feature for this method.

org/sunflow/core/accel/KDTree$BuildTask.<init>



From the above graph, we can clearly see that our intuition 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.

3.1.7 Issues

DaCapo is old:
One of the major issues I faced when 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, and for some of them I wasn't able to run the benchmarks using the current version of the jre, let alone instrument them. 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.

Lack of Data Points:
At this point, I would like to note that DaCapo was 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 was because in many cases, it was difficult to graph methods with enough data points such that the result was meaningful, which also showed a good correlation between the feature values and the performance metric. Dacapo benchmarks provides its own input for the benchmarked systems and thus, I decided to revisit the closure compiler system where I could provide my own input to ensure that graphs wouldnt be considered meaningless due to a lack of data points.

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.
Table 4 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
⋮	⋮	⋮

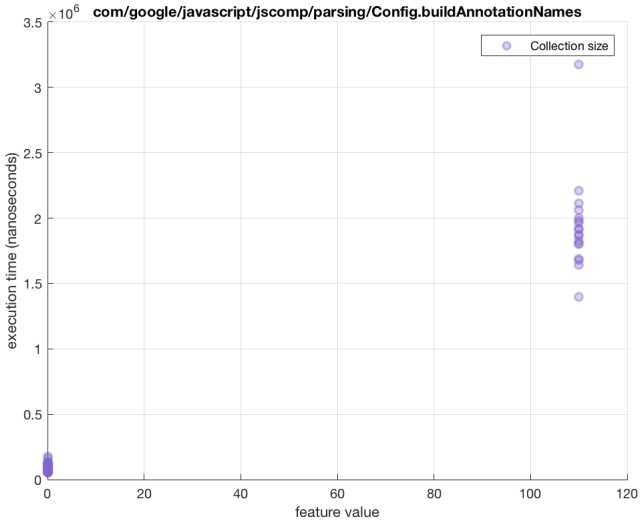
TABLE 4. Caption of the table

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.

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 5. Caption of the table

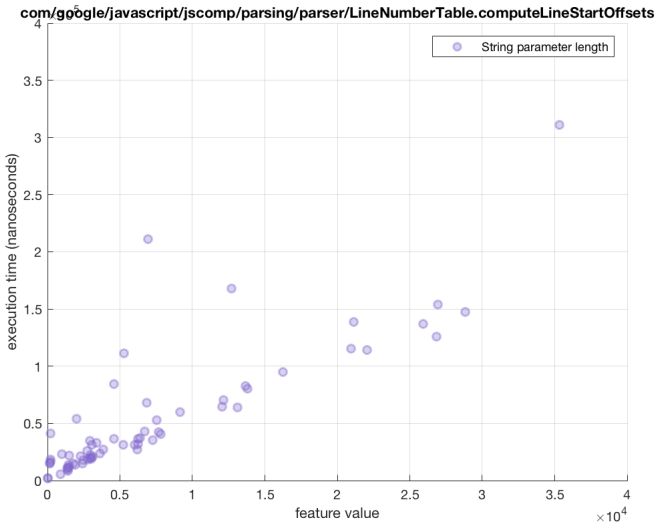
The buildAnnotationNames function looks promising with a pcc value close to 1.



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 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 6** 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.

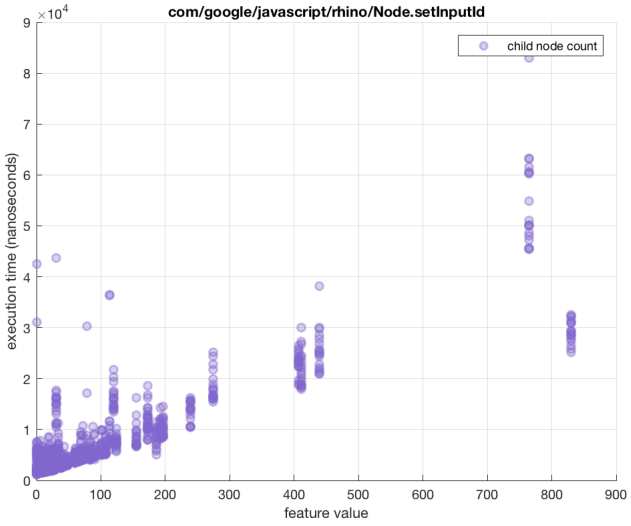
```
1       Object rec = apc.getReceiver(ArgumentProcessorMode.METHOD_ARGS);
2           if (rec instanceof Node) {
3               Node n = (Node)rec;
4               Measurement m = new Measurement();
5               m.arg_idx = arguments.length;
6               m.ft = Measurement.FeatureType.FT_RECEIVER;
7               m.fv = n.getChildCount();
8               m.value = duration;
9               ProfileExecutionTime.addValue(msc.thisMethodFullName(), m);
10          }
```

The code snippet above, shows how we looked at the Node object, and specifially at the child count of a node as a possible feature. **Table 6** 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 6. 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.

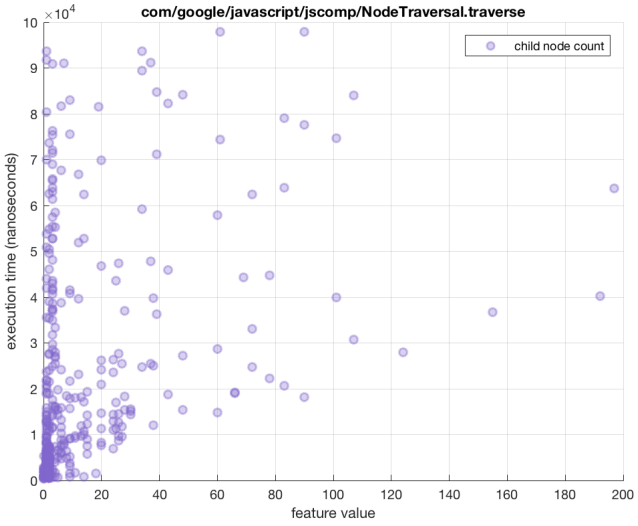
```
1    public void setInputId(InputId inputId) {
2      this.putProp(INPUT_ID, inputId);
3    }
```

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;
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 Results

5 Conclusion

References