VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Identifying Redundant Test Cases

Marc Shaw : 300252702

Supervisors: David J Pearce and A/Prof. Lindsay Groves

Submitted in partial fulfilment of the requirements for Bachelor of Engineering with Honours.

Abstract

Contents

Figures

List of Tables

Chapter 1

Introduction

Test suites are an important part in every major ~~project. They ensure a software program meets some~~ software project [**?**] . They check a specified set of behaviours ~~. To achieve sufficient confidence , these set of behaviours have to be met and~~ are met by a software program. Meeting the behaviours increases confidence in the program, but as a consequence~~a large portion of the code paths are executed. A large~~ , a large number of tests ~~are need to meet these behaviours and in turn take~~ need to be executed and may take up to several hours to run. ~~Throughout a project, test cases are constantly being added such as~~

Test cases are added to the test suite throughout the project, often when a piece of code is altered, new code is developed or a bug gets fixed [**?**, **?**]. ~~it be ensured~~ With tests being added throughout, we need to ensure that the thousandth test case is not replicating the behaviour of any of the previous~~?~~ . Even with careful planning, it is ~~near impossible~~ difficult to have no redundant test cases.

~~A trail of data is retrievable during the~~ A redundant test case is one which is replicating the behaviour of another. The goal of the report is to create a tool to identify these test cases within a test suite.

The execution of a test case leaves a trail of data. This trail contains information ranging from low level to high level run time data. A low level example would be machine code while a high level could be the method execution data. A variety of previous research [**?**, **?**, **?**, **?**, **?**, **?**, **?**] discuss using some of this information to identify redundant test cases. The ~~methods explored throughout this paper are interested in further investigating the potential to use such information to identify redundant cases.~~ papers identify redundant tests using statement coverage while the majority examine benchmarks with under 1,000 test cases. For benchmarks with a large number of test cases storing every statement execution could be costly. Therefore our report investigates using method execution data to identify redundant test cases. One of the issues that previous studies reported were high level of false positives. Method execution data gives a few different ways to explore this issue which are explored further on.

The tool developed can analyse method execution details, known as ~~a test's spectra~~*test spectra*, to determine the level of redundancy between two tests. Using this analysed information, potentially redundant test cases can be displayed to the developers. A basic visualisation of the idea is shown in Figure **??**~~. It shows a figurative spectra for~~ , the spectra is three different tests where colours represent method executions. Intuitively, it is clear that Test 1 is different from Test 2 and 3 as the ~~number of~~ method executions is ~~largely~~ different between them. However, there are similarities between Test 2 and 3 which ~~would be identified as a redundant test case and require a developer to examine the two~~could identify some level of redundancy between the test cases.

~~It~~ Once the tests are identified as being redundant, they may be removed, however it is

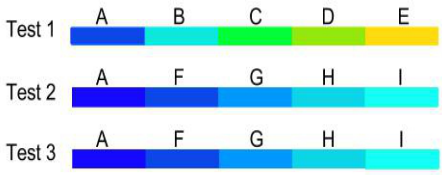

Figure 1.1: A figurative spectra where each colour represents different method execution details. We see similarities between Test 2 and 3. In contrast, Test 1 is different from 2 and 3.

important to understand the dangers of removing test cases. Unless two test cases are exactly the same, it is difficult to guarantee that they are redundant, even if one subsumes another. ~~Therefore the aim~~ This creates the need to be able to use different techniques, dependent on the project. Therefore to expand on the goal of the project~~is to create a framework that gives~~ , the developed tool should give developers different approaches ~~in~~ for identifying redundant test cases. The ~~framework~~ tool should allow a developer to configure different analysis metrics and view the results ~~through an output file. This means that the framework~~ in a file. Overall, the tool is useful for gaining an overview and understanding of the condition of the ~~current~~ test suite, allowing for manual inspection to determine if ~~potentially redundant tests are actually redundant~~the identified redundant tests should be removed.

Another potential use case of the ~~framework~~ tool is to redistribute the test cases. This can be achieved ~~through the splitting of any highly~~ by splitting the non redundant tests into ~~separate test suites and running the suites at different times,~~ another test suite and running this suite in place of the original. The original test suite may be run over night when no development is occurring. This separation of tests based on redundancy allows for a testing process, for example regression testing, to occur in a timely fashion while ensuring the original bug finding ability is retained. ~~For example, one test suite can be run during continuous integration, and the other over night. David Pearce~~This is applicable to David Pearce, he is currently writing a language called Whiley~~, which~~ . The language contains an extended static checking ~~framework~~ tool in order to eliminate run time exceptions through formal verification techniques. In the main compiler module alone, there are roughly ~~1500~~ 20,000 tests. Relocating a number of these tests into another suite would result in allowing ~~David Pearce~~ him to increase development speed due to a reduction in the time taken to run a large test suite. ~~Frequent execution of tests~~ Increasing the frequency of test suite execution also allows for bugs to be traced back to code changes ~~, using a smaller suite for every few changes means it will take less time to execute. The bug finding ability is not affected due to being able to run the full suitewhen development is not being done.~~

~~Other papers examining this research area identify redundant tests using statement information, our research explored throughout the report will be using method coverage. The majority of the previous studies examined bench marks with under 1~~easier and reduces the time spent debugging. Therefore, by redistributing the test suite, ~~000 test cases. For bench marks with over 1, 000 cases storing every statement execution could be costly. Therefore it would be intriguing to see to what extent can method execution data identify redundant test cases. One of the issues that previous studies reported were high level of false positive. Method execution data gives a few different ways to explore this issue which are explored in Section ??. The aim of the paper~~ not only does the time taken to run a test suite decrease, there is incentive for developers to execute the suite more often and in turn helping them reduce time taken to debug.

The contributions of the report is split into two sections:

- Create a ~~framework to identify~~ tool for identifying redundant test cases
- Analyse different strategies ~~to identify~~ for identifying redundant test cases through experimenting on realistic benchmarks

Chapter 2

~~Background~~

1.1 Outline

1.2 ~~Tracing~~

~~There were two considered tracing frameworks to use. Implementation details of these frameworks are further discussed in Section ??.~~

~~The first framework considered was AspectJ. It uses byte code weaving to trace method executions; this is when a piece of code is added to existing code without modifying the code itself and can be achieved through several different ways.~~

~~Compile time: The classes are compiled with the aspect weaved into them. When the jar is executed, the methods have the byte code from the aspect weaved into it already. This requires the source to be compiled with AspectJ's compiler. Load time: This involves binary weaving deferred until the point in which a class loader will attempt to load in a class file. This is achieved by using a command line argument notifying java to use the AspectJ class loader.~~

~~To distinguish what code to weave, and where in the existing code to weave it, a point cut can describe these in AspectJ [?]. The other framework is Java Debugging Interface (JDI). JDI is similar to using an observer pattern. The list of classes to observe are selected, when a method is called the listening class will be notified if it has registered to be observed and will be given the information of the method call.~~

1.2 ~~Performance Evaluation~~

~~The purpose for evaluating a performance of a given application is to create an understanding of the typical performance. For this typical performance to be valid, it has to be rigorous in~~ The report is structured as follows. Chapter 2 discusses background information that explores concepts needed to understand the following chapters as well as previous research in this area of interest. The design and implementation ~~. This involves taking into consider a variety of factors that will be examined in reference to research papers that explore the issues.~~

~~Andy Georges et al. [?] and Steve Blackburn et al. [?] present issues and alternatives to the current java performance methodologies used in research papers. They note that in premier conferences, 16 of the 50 papers examined did not examine the methodology they used. This limits the validity of the results and creates a situation where the experiment can not be reproduced. One of the main issues identified is specifying singular numbers, without expressing what the number is referring to (average, median, best, worst), leading~~

~~to potential misleading representation of the results. A set of parameters that are of interest are explored in the paper, these include, heap size, start up performance, number of virtual machine (VM) invocations and handling of garbage collection. An experiment should also take into consideration the environment that the analysis is executed on. Heap Size – A change in heap size can impact the garbage collection process. The smaller the heap size,~~ of the tool is then examined in Chapter 3. After the tool is discussed, Chapter 4 investigates the different techniques explored and conducts and analyses experiments. Finally, Chapter 5 concludes the report and identifies future work.

Chapter 2

Background

This chapter firstly explores the current research conducted in this area, it then covers the key ideas that are needed to understand the remainder of the ~~more often the garbage collector is run. Start Up Costs~~ report. These ideas are – ~~The costs associated with starting the application up. This will always involve class loading and just in time (JIT) re(compilation), it can also involve reading from a database and setting the application up. Number of VM invocations – The number of application runs that a single VM instance will execute. Garbage Collection – The process in identifying and removing objects that are no longer referenced. Environment – The hardware that the application is being run on.~~

2.1 ~~Grid Computing~~

~~A grid computing system is a distributed system that has no interactive work loads between machines. No interactions results in task independence being a necessity. The particular grid system used for this paper is located around Victoria University of Wellington. The machines in the grid are idle machines around the School of Engineering and Computer Science. There are a limited number of machines, therefore a total of 150 jobs can be run at a given time. Since the grid is located around an active community, if a user logs on while a process is being conducted, the application is paused until the machine returns to an idle state~~approaches to storing method coverage, edit distance metrics, evaluating performance of Java programs and the use of grid computing.

2.1 Related Work

2.1.1 Identifying Redudant Tests

Testing is a critical part to any software engineering process, not only to stop incidents stemming from the the product, but also partially related to the increase in popularity of agile methodologies [**?**]. Many of these methodologies employ test driven development and continuous integration resulting in testing becoming more important throughout the development process. For these reasons there have been research papers that examine the different approaches that can be taken to identify redundant test cases and reduce the size of test suites [**?**, **?**, **?**, **?**, **?**, **?**, **?**].

~~It is unclear whether~~ Whether programmatically reducing a test suite's size is worth the trade off in the ability to locate bugs is unclear. Wong et al. [**?**, **?**] explored the impact that reducing the size of the test suite based off of code coverage had on the fault detection capability. By introducing faults into several programs and comparing the performance

between the original and reduced test suites, they found that test suite reduction does not severely impact fault detection capability. In contrast to this finding, Rothermel et al. [**?**, **?**] further expanded on Wong's work by using different benchmark's and performance metrics. They found that test-suite reduction can severely impact the fault detection capability. The uncertainty created by these conflicting studies motivates our research to decouple the removal of tests from the framework and move the responsibility onto the developers.

A popular technique used in detecting redundancy involves analysing the statement executions, known as coverage information. Maurer, Garousi and Koochakzadeh [**?**] attempt to answer the question, is coverage information enough to determine redundant test cases? They state a redundant test case as being one that does not improve a specific criteria. For example, in Figure **??** we see that according to the statement coverage data of test cases , Figure **??** represents the statement criteria data from test cases T1 ... to T5, . The figure shows that T4 and T5 are fully redundant as T3 covers the statements that are executed by those tests. They the tests. The authors looked at two other criteria, branch coverage and granularities. Granularity criteria involved splitting the tests into setup, exercise (execution), verify (assert) and lastly teardown then performing analysis over each section. They implemented two different metrics in which both used the criteria described above. The first metric examined each individual test with respect to every other test. It was calculated by measuring the percentage of a test case that is also a subset of the other test case. The second metric examined each test case with respect to every test suite as a whole. It was calculated by measuring the percentage of a test case that is covered by the test suite without the test in it. Comparing By comparing with manual inspection, they were able to determine the level of false positive and actual redundant tests. Of the redundant tests manually identified, the algorithm matched 95% of those. However, of the tests that were manually identified as being non-redundant, 52% of these were identified as being redundant by the algorithm. They concluded that coverage-based information is vulnerable in giving false-positives when identifying redundant test cases, suggesting common code paths as being a root cause. Statement coverage criteria gave a high rate of detection, but the implementation allowed for false positives to impact the results.

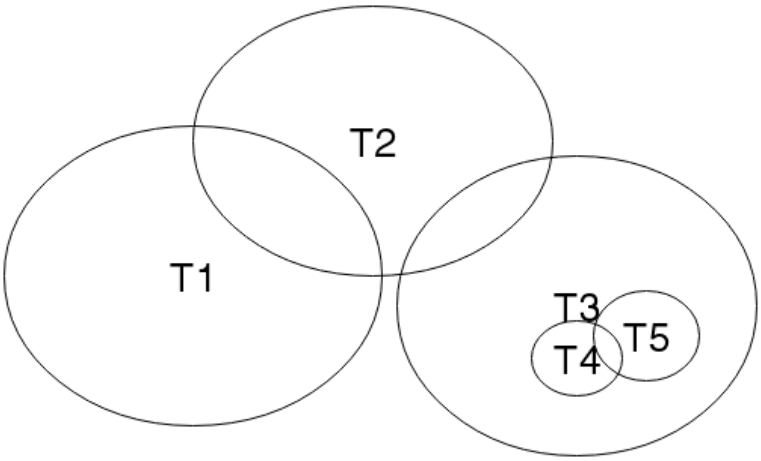

Figure 2.1: The coverage of each test is shown by a circle. It shows that T4 and T5 are redundant as T3 already covers those statements.

Zhang, Marinov, Zhang and Khurshid [**?**] examined the use of a greedy technique in comparison to heuristics. This required a criteria to be set by the tester, for example, statement coverage. The greedy technique would greedily select a test case that satisfies the

maximum number of unsatisfied test requirements and would continue until all the test requirements had been satisfied. So the new test suite will contain exactly the same coverage as the old test suite while removing redundant test cases. This means that if a test subsumed another, then it would always be removed. The heuristic implementation was first conceived by Harrold, Gupta and Soffa [**?**] where essential test cases are selected as early as possible. Essential being that only one test case satisfies a test requirement exclusively. The heuristic approach resulted in the most cost-effective reduction, this being the amount of reduction and the fault-detection capability of the reduced test suites, showing that although greedy approach worked, there were better techniques available.

In situations where it is not possible to generate a spectrum to analyse, static analysis can be used to determine the level of redundancy. Robinson, Li and Francis [**?**] examine this. The tests for the benchmark they use are written in a high level automation framework and consist of a list of commands. The commands perform actions such as file copying and loading configurations. To identify redundant test cases they examine the test ~~cases~~ case commands as well as the instructions within the procedures that the test case loads. To calculate the similarities between two test cases, they consider three different metrics, Manhattan distance, unigram cosine similarity and bigram cosine similarity. They each were measuring how closely related two tests were based on the sequence of commands and procedures loaded. Their findings were similar to Maurer et al. [**?**] in that there are a large number of false positives.

Static checking has several limitations in comparison to dynamic. ~~Firstly, during~~ The main disadvantage is the availability of data. During run-time is when a large ~~part~~ portion of the data trail is ~~accessible. Before then the only information available is the method calls. Static checking therefore~~ available and is important when needing to know the exact method calls and parameters passed to these methods. Static checking would provide useful information in a framework where dynamic data can not be collected~~and allows for the test cases to be~~. The framework would also need to be applicable to being examined in a static fashion such as the one used by Robinson, Li and Francis [**?**]. ~~Dynamic allows for more data to be collected such as the parameters passed to a method as well as the ability to be executed on more suites. The papers examined looked at the~~

Taking into the related work discussed, each paper used the coverage of a test suite at several different criterion levels but none looked at the ~~spectra~~ method execution explicitly. This leaves a potentially useful approach to the problem that may help determine the level of redundancy within a test suite.

2.1.2 Calling Context

When methods execute there is a trail of data that is left behind. Piecing together this data can show the relationships between methods, this relationship is known as a ~~call graph~~*call graph*. The call graph can be retrieved statically or dynamically [**?**]. A static graph represents every possible run of the program. A dynamic graph represents one particular run. ~~In contrast~~Comparing them, a static graph requires more information to be held and in the particular case of profiling test cases, dynamic would be more suitable due to the nature of a test case being the same for every run. Another advantage of a dynamic graph is the ability to collect functional parameters. Xiaotong Zhuang and et al. [**?**] describe a call tree as the overall tree of every method execution. To store the full call tree would involve an excess of memory, therefore they discuss the use of a calling context tree. This tree represents the same method executions however is compressed where the edge between the nodes contains a weighting, the weighting being the number of occurrences of that method execution (calling frequency). Examining Figure **??** shows the different variations that can be observed

with the same data. The top left hand picture shows the method execution trail, where A calls B, then B calls D twice and A calls C, then C calls D once. The call graph image (top right) condenses the information into one node per method call. This method is the most condensed variation, but gives the least amount of information. Directly below this, ~~(~~the calling context tree ~~) condense~~ condenses it slightly less. It separates each path into its own branch when it is unique to the tree, the connection between nodes contains a weighting which represent the frequency. Finally, the bottom left (call tree) creates a new path for every new method execution regardless of the uniqueness, giving us the most information about the context if the order is retained.

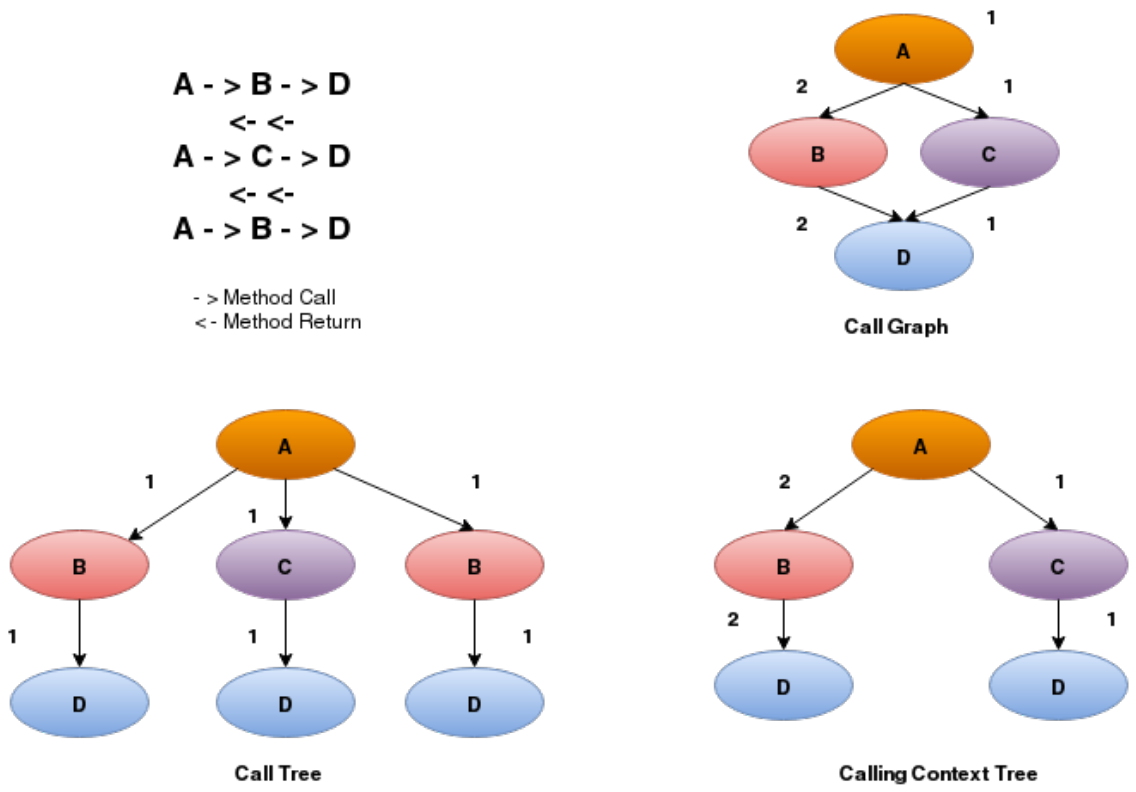

Figure 2.2: A diagram showing the three different variations of Call Graph information. Where each have a varying level of information that is stored.

2.1.3 Edit Distance Metrics

Edit distance algorithms play an important part in many domains, ranging from signal processing to mutations in genome sequences [?]. It calculates the number of edit operations needed to go from one string to another, in our research they are used to calculate the difference between test case spectras. There are a variety of different implementations of edit distance metrics. Cohen, Ravikumar and Fienberg [?] explore different edit distance metrics for name matching. They concluded that Monge and Elkan [?] performed the best for string edit-distance metrics. The metric works by splitting the two strings into tokens, ~~these~~ the best matching tokens are then calculated to find the similarity using other common metrics such as Levenshtein distance. One of the other techniques they explored was the Levenshtein distance [?] metric by itself. This metric is the minimal number of operations that can be done to make one ~~tests~~ test's spectrum equal to another. These operations

are inserting, deleting or substituting and a cost is associated with completing an operation. The maximum difference is the size of the larger ~~of the two spectra's~~spectra. The amount of redundancy is calculated by dividing the cost of operations with the max difference in order to normalize the value. This value ~~will be~~ is the percentage of ~~tests that are~~ the test that is not redundant, ~~we minus this value by~~ the value 1 is then subtracted by the value to give the redundant percentage.

An example of Levenshtein is shown in Table **??**, where 'kitchen' is being changed into 'kitten'. The example shows the number of operations needed is 2, and the max potential needed if the two strings were completely different is 7, as kitchen contains 7 characters. The redundancy is calculated by subtracting 1 from the cost over the max potential cost $(1 - (2/7))$. The outcome being that the words contain 71% redundant information.

Previous State	**Current State**	**Operation**
-	kitten	-
kitten	kitcen	Substitution of 't' with 'c'
kitcen	kitchen	Insertion of 'h' between 'c' and 'e'

Table 2.1: Using Levenshtein edit distance metric to transform kitten to kitchen.

An example of Monge & Elkan is shown in Table **??**. The comparison of the strings 'paul johnson' and 'johson paule' are being compared. Each of the strings get broken down into two tokens and the best matches are identified and used for the final score. The table shows that the final score of the Monge & Elkan is taking into account 'paul' & 'paule' and 'johnson' & 'johson'. Using these two matches, the final result is 1/2 * (.85 + .80) = 0.825.

Input string 1:	paul johnson
Input string 2:	johson paule
Levenshtein("paul","johson")	0.00
Levenshtein("paul","paule")	0.80
Levenshtein("johnson","paule")	0.00
Levenshtein("johnson","johson")	0.85

Table 2.2: The Monge & Elkan edit distance metric to calculate the similarity between "paul johnson" and "johson paule"

2.1.4 Wilcoxon Signed Rank Test

A Wilcoxon Signed Rank test is used to infer whether there are any significant differences between the techniques implemented in this ~~paper~~report. The Wilcoxon Signed Rank test was first proposed by Frank Wilcoxon in 1945 [**?**]. It is a nonparametric test for comparing two pair groups. A nonparametric test is a collective term that ~~are~~ is given to inferences that are valid under less confining assumptions than classical statistical inferences [**?**]. The test does not assume that the data follows a normal distribution which is ideal for the data presented in our research and is used when two nominal variables and one measure variable is being measured [**?**].

Chapter 3

~~Design and Implementation~~

2.1 ~~Overview~~Performance Evaluation

2.2 ~~Tool~~

~~An aim of the project as mentioned in Section ?? is to create a tool to allow developers to~~ Throughout the report different techniques are designed and implemented to identify redundant test cases. ~~The tool consists of a variation of settings that give developers a range of options when deciding the different confidence levels they are wanting. The tool allows for~~ To evaluate the techniques, experiments are conducted and must cohere to a standard procedure. The purpose for evaluating the performance of a given technique is to create an understanding of the ~~developers to firstly trace the test suite of an application. This data is then analysed to produce a list of redundant test cases. To get the tool into a usable state, several impediments had to be overcome and are discussed in the following section.~~ typical performance. For this typical performance to be valid, it has to be rigorous in design and implementation. This involves taking into consideration a variety of factors that will be examined in reference to research papers that explore the issues.

2.1.1 ~~Pipeline~~

~~The first issue encountered is the time taken to analyse the data. To conduct the analysis, every test is compared to every other test. The issue arises when the spectrum's of the test cases contain tens of thousands of method calls. This results in the analysis taking up to several days to complete. The approach identified to reduce the time is a pipeline combined with time reduction strategies.~~

~~The pipeline approach is shown in Figure ??. The idea is to use analysis stages to reduce the number of comparisons each additional stage has to compute. Each analysis stage can be set by the developer within a properties file, these settings are~~ Andy Georges et al. [?] and Steve Blackburn et al. [?] present issues and alternatives to the current Java performance methodologies used in research papers. They note that in premier conferences, 16 of the 50 papers examined did not discuss the methodology they used. This limits the validity of the results and creates a situation where the experiment cannot be reproduced. One of the main issues identified is specifying singular numbers, without expressing what the number is referring to (average, median, best, worst). This leads to a misleading representation of the results. A set of parameters that are of interest are explored in the paper, these include, heap size, start up performance, number of virtual machine (VM) invocations and handling of garbage collection. An experiment should also take into consideration the environment that the analysis is executed on:

- **Heap Size** – ~~the spectra type, analysis metric to use and level of redundancy for each. There are two approaches that can be used to reduce the time taken to analyse the data~~ A change in heap size can impact the garbage collection process. The smaller the heap size, the more often the garbage collector is run.
- **Start Up Costs** – ~~use of a heuristic and concurrent execution.~~ The costs associated with starting the application up. This will always involve class loading and just in time (JIT) re(compilation), it can also involve reading from a database and setting the application up.
- **Number of VM invocations** – The number of application runs that a single VM instance will execute.
- **Garbage Collection** – The process in identifying and removing objects that are no longer referenced.
- **Environment** – The hardware that the application is being run on.

~~Trace information goes in at the start of the pipeline. After each stage there will be a reduction of comparisons that the next stage has to complete. The last stages should be the most computationally heavy.~~

2.1.1 ~~Time reduction~~

~~**Heuristic**~~

~~The heuristic examined a subset of~~

2.2 Grid Computing

The time intensive nature of conducting the experiments resulted in a single computer not being adequate. A grid computing system was used instead, it is a distributed system of multiple machines that have no interactive tasks between each other. No interactions results in task independence being a necessity. The particular grid system used for this report is located at Victoria University of Wellington. The machines in the ~~available data. A subset of data created a trade off between confidence and speed. Increasing the confidence allows us to increase the speed. The following illustrates an example of this .~~

~~$K, I, T, C, H, E, N \neq K, I, T, E, N, Z$~~

~~In this case, the two test cases would not be redundant due to them having a large difference in method calls.~~

~~$K, I, T, C, H, E, N \approx K, I, T, C, H, N$~~

~~There is a chance that these two may be redundant so it implies it should be get through the analysing stage onto the next. Using a benchmark example of Whiley on the wyc package, a heuristic approach enables us to decrease the number of comparisons from 90, 902 to 38 for the most computationally heavy stage.~~ grid are idle machines around the School of Engineering and Computer Science. There are a limited number of machines, therefore a total of 150 jobs can be run at a given time. Since the grid is located around an active

community, if a user logs on while a process is being conducted, the application is paused until the machine returns to an idle state.

Chapter 3

Design and Implementation

~~Concurrent~~

3.1 Overview

~~Concurrent execution was implemented by splitting the test cases up into 8 different parts, each part knew the test cases that it had to compare. A new thread executed each split, making the implementation relatively easy. The concurrent execution lead to a decrease in roughly 2 times the time taken to analyse but lead to a trade off in increased memory usage. This meant concurrent execution was only usable for the smaller bench marks.~~ The goal of the project as mentioned in Section **??** is to create a tool to allow developers to identify redundant test cases. The tool can be split into several different sections and each section will be discussed in this chapter. The first objective of the tool was to trace the test data. This involved exploring two different frameworks and comparing the usability of each. After the tool was able to trace the data, we had to determine what spectras we were interested in. The three different spectra types are then discussed. The data that was traced from each test case was then compared to each other to determine the level of redundancy between each test using edit distance metrics. Throughout the testing of the tool, one of the main impediments was the time taken to analysis the large amount of data. We examine how pipelining the output of a heuristic analysis into more computationally heavy analysis decreases the time taken. The final two sections discuss two more advanced areas of the tool. Firstly, retrieving the parameters of the method calls and discussing the trade off parameters create between time taken and more information to analyse. Secondly, giving a weighting to each method call was implemented with the purpose of reducing false positives.

3.1.1 ~~Saving to the Network~~

~~Saving the data to network solved two issues. Firstly, it allowed for the datato be reanalysed without having to execute the test suite again. Secondly, the test data had to be accessible to the grid machines. Saving the data on my user profile allowed for the data to be retrieved, regardless of the machine an analysing job was being run on.~~

3.2 Tracing

~~During the test suite execution, each test case creates a data trail that is partially recorded. Accessing different sections of the data trail gives different information .~~

A fundamental requirement for this project was to trace which methods were executed by a test. As David Pearce's language Whiley is written in Java~~. Taking this into account~~ ,

it was decided to use Java to trace a ~~tests spectrum. There are two viable options, the Java~~ test. There were two considered tracing frameworks. The first framework considered was AspectJ. The technique that AspectJ uses is called Aspect Oriented Programming (AOP). AOP can be used to add code to an existing program without modifying the source code, this code added is known as an aspect. This process be achieved through the following methods:

- **Compile time** – The classes are compiled with the aspect woven into them. When the program is executed, the methods have the code from the aspect woven into it already. This requires the source to be compiled with AspectJ's compiler.
- **Load time** – The weaving process is deferred until the point at which a class loader attempts to load in a class file. Load time modification is achieved by using a command line argument notifying Java to use the AspectJ class loader.

To distinguish what code to weave, and where in the existing code to weave it, a point cut can describe these in AspectJ [**?**] . The other framework we considered was the Java Debugging Interface (JDI)~~or AspectJ, as discussed in Section ??~~. JDI is similar to using an observer pattern. The list of classes to observe are selected, when a method is called the listening class will be notified if the method has registered to be observed and will be given the information about the method call.

It was decided to use AspectJ over JDI. AspectJ ~~is~~ was easier to choose which methods to record and returns the actual object when retrieving the parameters of the method call~~allowing for reflection to be used to retrieve the information for every object.~~ . This detail becomes important when we explore tracing parameter values in Section **??**. In comparison, JDI was faster to execute, however there was a limited amount of documentation available and the parameters returned were not the actual objects~~, meaning that standard reflection could not be used to retrieve the values~~. The decision to use AspectJ was based off this trade off between information and performance. ~~The analysis tool was able to be altered to increase the performance of it, so having~~ Decoupling the data gathering and data analyse in the tool removed the emphasis on retrieval performance and resulted in the extra information ~~that AspectJ gave was more importantthan an taking less time to execute~~being more important.

To get AspectJ to trace the method ~~execution~~details, a point cut was made to record every execution. A simplified version of the aspect is shown in Figure **??**. There are two point cuts within the aspect, the first ~~pointcut~~ point cut is going to be called for every method which has a ~~junit~~ Junit Test annotation attached to it. This will then let a static service know that a new test has been started. The second ~~pointcut~~ point cut is used to trace everything ~~but methods attached to~~ apart from methods with a Test annotation ~~.~~ attached. The next stage was to weave the ~~point cuts~~ aspect into the benchmark. Using compile time weaving would have meant that each benchmark needed to be recompiled using the aspect compiler. In contrast~~to load time , which~~ , load time only requires the AspectJ class loader to be passed through a command line argument. ~~The load~~ Load time was chosen for it's ease of use when working with external benchmarks as it only required AspectJ's class loader~~to be use~~.

3.2.1 ~~Parameter Values~~

~~The related work in Section ?? explored the statement coverage while ignoring the parameter values that each method was passed. It could be argued that due to knowing the statement path, parameters are irrelevant as you know the path the method will take, and parameters~~

will not add any more information. When using method execution details alone, these become crucial when determining the level of redundancy due to the limited amount of information that the method details alone gives us.

AspectJ gives direct access to the object's that are contained in the parameter values of the method. This allows for reflection to be used to retrieve the objects in a representable state. By using reflection, the fields of the objects can be retrieved and returned in a string. This string is then examined to remove any java object reference location and stored. Initially it was decided to examine primitive types only, this resulted in a very limited number of parameters and did not have the desired effect of decreasing the level of false positives while increasing the time and memory. Changing to reflection allowed for more detailed parameters to be collected. This increases the certainty about the whether two tests are redundant, decreasing the false positives but increasing the time and memory even more than the use of primitives only.

The most common use case of the tool would involve the use of parameters therefore it was decided to optimize this through storing the data with parameters. The optimization involved saving the parameters with the trace information. If parameters value is set to false, the parameters have to be split off rather than added on. This means that setting the parameters to false would increase the set up time.

```
pointcut testMethod(): execution(@org.junit.Test * *(..));
before(): testMethod(){
    StaticService.addMethodCall("New Test:" + thisJoinPointStaticPart
            .getSignature().getDeclaringTypeName());
}

pointcut traceMethods() : ( !within(@org.junit.Test *) && execution(* *(..))&&);
before(): traceMethods(){
    StaticService.addMethodCall(New Test:" + thisJoinPointStaticPart
            .getSignature().getDeclaringTypeName();
}
```

Figure 3.1: A simplified point cut within AspectJ. The first ~~pointcut~~ point cut is for when a new test is executed and the second when a new method is executed.

3.3 Test Spectra

The idea of a spectrum was previously identified in Chapter **??**. Reexamining the idea, a spectrum is some abstraction of the method execution information that is retrievable during the execution of a test. ~~Our tool is particularly interested in utilising the method execution details. This allows there to be several~~ The data retrieved can be split into three different types of spectra~~. The main three spectra examined are~~ , these are ~~*set*~~*unique method calls*, ~~*list*~~*every method call* and *calling context*. To explore the different spectra's, an example trace of methods, 'kitten' will be used. Each letter represents a method execution and each letter is executed ~~from the~~ by the same method. The different spectra's take up different levels of resources. The resources explored are – time taken ~~, memory and complexity~~and memory.

3.3.1 ~~Set~~Unique Method Calls

~~A set of method executions~~ The unique method call spectra is where every method execution is only taken into account once. Examining the example trace, the output would be 'kiten',

with the difference being the removal of the repeated 't' method. The implication of limiting the method calls to one per method call on the number of comparisons will be dependent the benchmark. Using Whiley Compiler benchmark as an example, there are 494 unique method executions for a single test and 80,000 method calls to these method executions. Utilising the ~~set method~~ unique method call method, the tool will only ~~examine the~~ compute the redundancy of 494 method executions, rather than the 80,000, ~~a decrease of~~ $^{6}/_{1000}$ ~~. This leads to a substantial reduction in~~ of the original amount. The motivation of the spectra is to substantially reduce the time taken and memory used. However, by decreasing the amount of data means that there is a decrease in confidence that the tests produced are redundant test cases. ~~The motivation behind the spectra to use it as a heuristic pipeline, as~~ How this fits into the tool is explored in Section ~~??.~~ ??. Unsure if this last sentance is good or just to remove

3.3.2 ~~List~~Every Method Call

~~A~~ The every method call takes into every method execution, analogous to a list of method ~~executions is where every~~ calls. This spectra is the same as using a calling context of one, where only the highest method execution is ~~taken into account~~examined. Examining the example trace, the output would be 'kitten', with there being no difference between the two. Using the Whiley Compiler benchmark as an example, a list spectra would ~~examine~~ compute the redundancy of all 80,000 method calls~~to calculate the level of redundancy. A list spectra is the same as using a calling context of one, where only the highest method execution is examined.~~ . Comparing this to ~~a set~~ the unique method call spectra, this is expected to lead to an increase in the time taken but ~~increase the confidence of the redundancy. The motivation behind the approach is to use during the second stage in the pipeline to decrease the amount of comparisons that the final pipeline should do. This approach would not be used in the last pipeline as there is limited sense to use a subset of information retrieved from the test cases.~~ also increase the credibility of the results.

3.3.3 Calling Context

A calling context ~~of method executions is where each method call, the trace data~~ contains a separate ~~for each call stack~~ node for each parent method that the method was called ~~with [?] . The depth~~ from [?] . The number of call retained is referred to as ~~K.~~ the K depth. The motivation behind the approach is to increase the credibility of the results by taking into account for more data. The consequence of increasing the amount of data is an increase in the time taken and memory used to analyse. Examining the example trace, the output would be 'Parent $\rightarrow$ k, Parent $\rightarrow$ i, Parent $\rightarrow$ t ...). Using the Whiley Compiler benchmark as an example, a calling context spectra would examine all 80,000 method calls, with each containing K depth of method calls. ~~The motivation behind the approach is to be the final stage in the pipeline. Having this approach as the final pipeline would give the highest confidence that the tests were redundant, as it has access to the most information.~~

To retrieve the calling context in Java, the current stack trace of a method call is examined and then parsed to retrieve the relevant information. This parsing involves removing the method calls that are used to retrieve the stack trace and any memory location details. Should i remove this or introduce every section?In the next section, two different analysis metrics are introduced and examined.

3.4 Analysis Metrics

There were a variety of different edit distance metrics to consider, the two of particular interest were Monge Elkan [?] and Levenshtein [?] . These were explored in Chapter ??, it is discussed how The data retrieved from the tracing process needs to be analysed to produce a quantitative value. The value produced will be used to determine the level of similarity between two test cases. Edit distance metrics were discussed in Section **??**, with two different edit distance metrics considered. They were Monge & Elkan metric [?] which splits the strings into tokens sections and compares the tokens while Levenshtein looks over the whole stringsections and Levenshtein [?] which compares tokens, generally tokens are single characters.

For both of the algorithms, there were publicly available frameworks that implemented them. Monge Elkan distance metric split a method call into sections and ran Levenshtein over them. In contrast, the separate Levenshtein implementation Both implementations allowed for the method calls to be split into tokens, these tokens represented a whole method call rather than the method call being split up as in represented with a token. Each test case contained a list of the tokens to represent the trace information of it. The issue with Monge & Elkan . Visualizing the idea was it attempted to find the best match. Attempting to find the best match would increase the time taken, perform unnecessary computation for our requirements and increase the false positive rate. Using an example of trace information in Figure **??**. The top figure shows Monge and Elkan would split the method calls, comparing each token to produce the number of operations at a character level. The implementation of Levenshtein allows for whole method calls as shown to be compared to each other, for instance, if one method callis different then the number of operations is one. Comparison of method calls increases the comparison speed and decreased the time takenimplementation of Monge & Elkan would find the best match, in turn matching 'Method Call' in test case 1 to the 'Method Call' in test case 2. Levenshtein did not search for a match, and instead only looked at the opposing method call. This one to one matching was preferred as we were interested in the order of the method calls as well as what methods were called. Overall, Levenshtein integrated better into our toolthe tool's ability to identify redundant test cases.

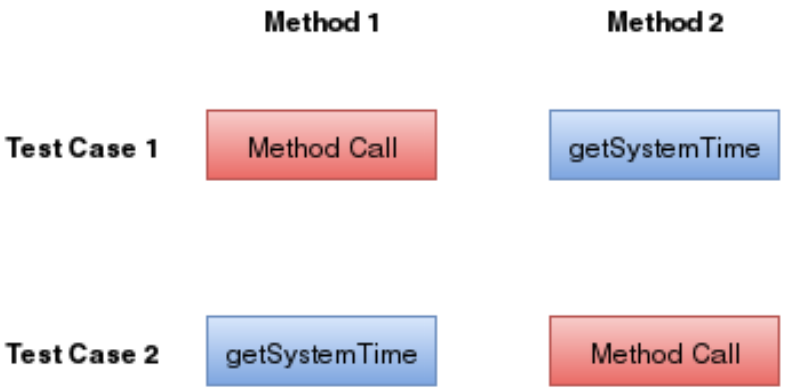

Figure 3.2: An example showing how Monge & Elkan split the method calls into token, where Levenshtein represents the method calls as a whole.

An example is shown in Table **??**, where 'kitchen' is being changed into 'kitten'. The example shows the number of operations needed is 2, and the max potential needed if the two strings were completely differentis 7 as kitchen contains 7 characters. The redundancy is calculated by subtracting

3.5 Pipeline

For the tool to help developers reduce the number of redundant tests, it has to analysis the information within an acceptable time frame. By comparing every test with every other while using all of the available information (calling context), this resulted in the analysis taking several days to complete. This is not considered an acceptable time frame. This issue arises when the spectrum's of the test cases contain tens of thousands of method calls. A pipeline approach was implemented which utilised the unique method call spectra as a heuristic. The goal of the pipeline was that each stage should soundly eliminate pairs from consideration that the following stages would have removed. The pipeline approach is shown in Figure **??**. Each analysis stage can be set by the developer within a properties file, these settings are – the spectra type, analysis metric to use and level of redundancy for each.

To further expand on the heuristic idea. Unique method calls spectra was used as it examined a subset of the available data. Inspecting the subset of data increased the speed of comparison but as a consequence decreased the credibility of our results. The following illustrates an example of this where each character represents a method call.

Comparison 1 K, I, T, C, H, E, N & K, I, B, E, N, Z

Comparison 2 K, I, T, C, H, E, N & K, I, T, C, H, N

Inspecting comparison 1 from the cost over the max potential cost $(1 - (2/7))$. The outcome being that the words contain 71% redundant information , it is clear that the two sets of trace information are different. Examining comparison 2, there is a chance that the tests may be redundant so the comparison should be get through the analysing stage and onto the next. Using a benchmark example of Whiley on the wyc package, a heuristic approach enables us to decrease the number of test case comparisons from 90,902 to 100 for the most computationally heavy stage.

The list spectra may also be used as another type of heuristic. The motivation behind the spectra is to use during the second stage in the pipeline to decrease the amount of comparisons that the final pipeline should do. This approach would not be used in the last pipeline as there is limited sense to use a subset of information retrieved from the test cases.

Figure 3.3: Trace information goes in at the start of the pipeline. After each stage there should be a reduction of comparisons that the next stage has to complete. The last stage should be the most computationally heavy.

~~Previous StateCurrent StateOperation- kitten - kitten kitcen Substitution of 't'with 'c' kitcen kitchen Insertion of 'h' between 'c' and 'e'Using levenshtein edit distance algorithm to transform kitten to kitchen.~~

3.5.1 Saving to the Network

To execute the analysis on a grid computing system, the test data had to accessible on the School of Engineering's local network. To achieve this, the data retrieved by the tool was saved to the local network. Not only did it allow for the analysis to be executed on the grid system, it also it allowed for the data to be reanalysed without having to re execute the test suite.

3.6 Tracing Parameter Values

The related work in the research area has explored using statement coverage while ignoring the parameter values of each method. It could be argued that due to knowing the statement path, parameters are irrelevant as you know the path the method will take, and parameters do not add any more information. When tracing at method level rather than at the statement, these become crucial to determine the degree of redundancy between test cases. Parameters give insight into the execution paths that will be taken and act as a proxy to the statement information without storing the same amount of the data.

There were two variations of parameters that were considered to trace. Firstly, primitive types only. By running test experiments on the benchmarks, the use of primitives showed that only a limited number of parameters were collected. The parameters collected had a limited effect on the number of false positives identified and time taken. The second approach is to use reflection. Reflection is the ability to examine and modify objects at run time [**?**] . Since AspectJ gives direct access to the object's in the parameters, reflection can be used. By using reflection, the fields of the objects are retrieved and returned in a string to represent the state of the object. This string is then examined to remove any Java object reference location and stored.

The most common use case of the tool is expected to use parameters therefore it was decided to optimize this through storing the data with parameters. The optimization involved saving the parameters with the trace information. If parameters value is set to false, the parameters have to be split off rather than added on. This means that setting the parameters to false would increase the set up time.

3.7 Weighting

Maurer et al. [**?**] and Robinson et al. [**?**] found that test ~~cases~~ suites often had a set of methods that were in every test, such as setup and tear down. These common methods could create false positives. To understand why, a redundant test is one where it is nearly or exactly a replication of another test. Since each method call within a spectra has the same weighting, the more setup and teardown calls made means that the execution stage has decreased weighting overall. We can see an example of this in Figure **??**. The figure shows test cases with different proportions of setup and tear down in relation in the execution stage. The figure also shows how the size of the test case may effect the proportion.

Two different variations of weighting were considered. The first variation involved ~~having~~ giving each method execution ~~be given~~ a weighting based on ~~it's~~ its call frequency, the higher the frequency, the lower the weighting. This would cause the more common

methods to have less impact on the final result, but not be removed completely. The other variation that was considered, and used, was completely removing the most used method calls for each test case. This involved removing every method execution that was more than 80 percent of the most frequent method call. ~~The~~ During initial experiments, the first variation was found to have less impact~~, with some of the method calls having a substantially larger number of method calls compared to the less frequent, it was difficult to find a weighting system that worked well for every test case, often the more~~ . The reason was that the frequent method calls were still having a large impact, even with a lower weighting. This meant that each benchmark needed different weightings therefore it was difficult to find a solution that could be applied on every benchmark.

Another decision was the scope of the weighting, either it was calculated per test case or ~~globally~~test suite. The issue with ~~using global~~ calculating per suite was that if the benchmark contained a mixture of large and small tests, ~~then the use of global~~ this could cause the smaller benchmarks to be reduced to a minimal number of method calls. The minimal number of method calls was often not enough information to present the test in a representable state. A per test case weighting was desired to reduce the issue of test cases being reduced to a minimal size.

Figure 3.4: A diagram showing how the different size of the test case can be affected by setup and teardown methods.

Chapter 4

Results and Discussion

The following section reports on the outcome of several experiments that explore the use of our framework ~~through~~ on a realistic benchmark suite. The key factors of interest are – the time taken, number of comparisons, the types of redundant tests identified and the overall cost of performance (time) vs precision (tests identified & types identified). Link back to req?

There are two points that the reader needs to be aware of. Firstly, in the significant tables below, a '+' represents that significant increase, '-' significant decrease and '=' represents no significant difference. Secondly, the graphs are displayed using a logarithm scale.

4.1 Method

The experiment method is key to producing believable and reproducible results. Before the experiment could be conducted, the data for each benchmark had to be retrieved. This involved setting the benchmark up locally and then executing the tests with the AspectJ class loader. The data was then run on a grid computing system. After the results had been returned, a Wilcoxon signed rank test [**?**] was used to determine whether the results were significantly different or not. The significant level used was 95% with a null hypothesis of the two samples median being equal. Overall, ~~a total of seven different property~~ seven different settings were run per benchmark with each ~~property~~ setting being run 30 times.

The goals of this paper as mentioned in Section **??** are the creation of a framework to identify redundant test cases and experimenting ~~the~~ with different techniques on realistic benchmarks. The rest of this ~~section~~ chapter explores the experiments conducted to give guidance to the users of the ~~framework~~tool.

4.2 Environmental Methodologies

As previously discussed in Section **??** there are a variety of challenges that present themselves when using Java to evaluate performance. These challenges are discussed throughout the following section.

4.2.1 Grid Computing

A grid computing system was used to execute the data analysis, with a total of 150 jobs queued on the grid at a single time. When using a grid system it is important to ensure that all the machines used are the same for every experiment. The grid allowed for particular

types of machines to be specified – 8gb ~~total~~ ddr3 ram, Linux 4.0.5 64 bit system and an ~~i5~~Intel Core i7-3770 CPU running @ 3.40GHz.

4.2.2 Measuring Time

The time taken is an important variable in evaluating the performance of a tool. In Java, this can be achieved by accessing a system method that returns the total time in milliseconds from 1970 00:00:00 UTC to the current time. Taking this time before the analysis, and subtracting from the time after the analysis gives the total time taken. This notation of time is defined as wall clock time. Utilising a grid system presents difficulties with using the wall clock time to measure the time taken.

As mentioned in Section **??**, when a job is run on the grid system, there is potential for the machine to be paused. If wall time were to be used, the time that the process is paused would also be included. To resolve this issue, the notation of CPU time was used. The CPU time is the measure of time in nanoseconds that the computer program spent on the CPU. Another ~~issue in regard to time taken~~ concern is if a user logs in during analysis, ~~is the programs~~ the tool's RAM could potentially be moved into virtual memory. This is dependent on the amount of RAM that the user needs. When the user logs out, the tool will restart and any information lost will need to be recalculated by the tool. Both these issues were limited by running the tests overnight when users were unlikely to log in.

When measuring the time taken to analyse, the start up cost was not taken into account. Running up to 150 jobs meant that a large number of applications would be attempting to access a single data file at a single time. This stress on the system introduces a large amount of non-deterministic behaviour. Removing the start up cost avoided this issue.

Heap Size

The heap is the location that the JVM uses to store objects that are produced during the execution of an application. The amount of heap that was allocated to the JVM was 6gb ~~for every benchmark over~~ on every benchmark for every run.

4.2.3 Software Environment

A concurrent-mark-sweep garbage collection strategy (default) was used. This approach uses multiple threads to scan the heap, mark unused objects and recycle them [**?**]. It allows for a high throughput however, tends to use more CPU time than other strategies. This was deemed worth the trade off as memory was the bottleneck rather than ~~CPU and increasing the throughput was more important~~the CPU.

4.3 Benchmarks

The benchmarks had to be realistic real world projects. They were located by looking at popular Java frameworks, Github repositories and David Pearce's personal projects. For a benchmark to be considered, it had to be Java based, have a reasonable number of tests (40+) and ~~was open source~~open sourced, while Ant and Gradle built projects were favoured due to their easier build process.

A variety of test types and sizes of benchmarks were examined. A range of sizes were used in order to fully evaluate the framework. The two ranges of benchmark sizes can be found in Table **??** and Table **??**, large and small respectively. Although ~~method execution details approach~~ the type of data retrieved is aimed toward larger test suites, it would be

Benchmark	Number of Tests	Type of tests	Authors
Whiley - Wyc		End to End	David J Pearce
Spring - Core		Unit Tests	Community
Metric-x - Core		Unit Tests	Community
Jasm		End to End	David J Pearce

Table 4.1: Large Test Suites

Benchmark	Number of Tests	Type of tests	Authors
Ant		Unit Tests	Community
Imcache		End to End	Community

Table 4.2: Small Test Suites

expected that the size will have no ~~correlation to the metrics examined~~effect on the outcome of the experiments. A range of test types were also chosen, David J Pearce's projects contain end to end tests which run through a whole module at once, while the rest attempt to test units of code at a time. Having a range of test types gives a wider evaluation view and gives insight into potential situations where different settings may not be helpful in achieving the aim.

The larger benchmarks that produced up to 100,000 method calls per test required a large amount of memory to store the details during data retrieval. Without the use of a database, this limited the number of test cases that could be collected. Whiley and Ant were the benchmarks that did not have all there tests executed.

4.4 Experiment I - Pipeline Length Comparison

4.4.1 Motivation

The pipeline size has impact on each factor of interest. Selecting a pipeline too small or too large will have varying negative effects on these. This experiment explores this trade off and the number of pipelines that should be used with the benchmarks.

Hypothesis 1 *Increasing the number of pipeline's improves the factors of interest*

Hypothesis 2 *Increasing the number of pipeline's leads to no difference between the output redundant test cases*

4.4.2 Settings

There were two different pipeline size's tested, two and three respectively. For both size's, the final pipeline stage was the same. It is noted that a single pipeline comparison was left out due to the amount of memory and time taken to finish analysing.

- Difference between runs: Pipeline size

4.4.3 Results

The significant table for comparing the use of pipeline of size two ~~vs~~ and size three is shown in Table **??**. It shows that there was a mixture of results for the total time taken to analyse the data. Metrics-x, Ant, Spring and Jasm performed significantly better with a pipeline of

size three in comparison to size two. Imcache had no significant difference and Whiley performed significantly worse with a pipeline of size three in comparison to size two. Every benchmark ~~has~~ had no significant difference between the number of redundant tests identified, every benchmark produced the exact same number of redundant tests in both pipeline size of two and three.

A chart showing how each benchmark reacted to the change in the pipeline size is shown in Figure **??**. It shows the number of test case comparisons that the final stage of the pipeline ~~has~~ had to conduct.

	Total Time	Redundant Tests Identified
Whiley	+	=
Jasm	-	=
Ant	-	=
Spring	-	=
Imcache	=	=
Metrics-x	-	=

Table 4.3: A table showing the significant relationship between the use of pipeline of size two with pipeline of size three for each benchmark

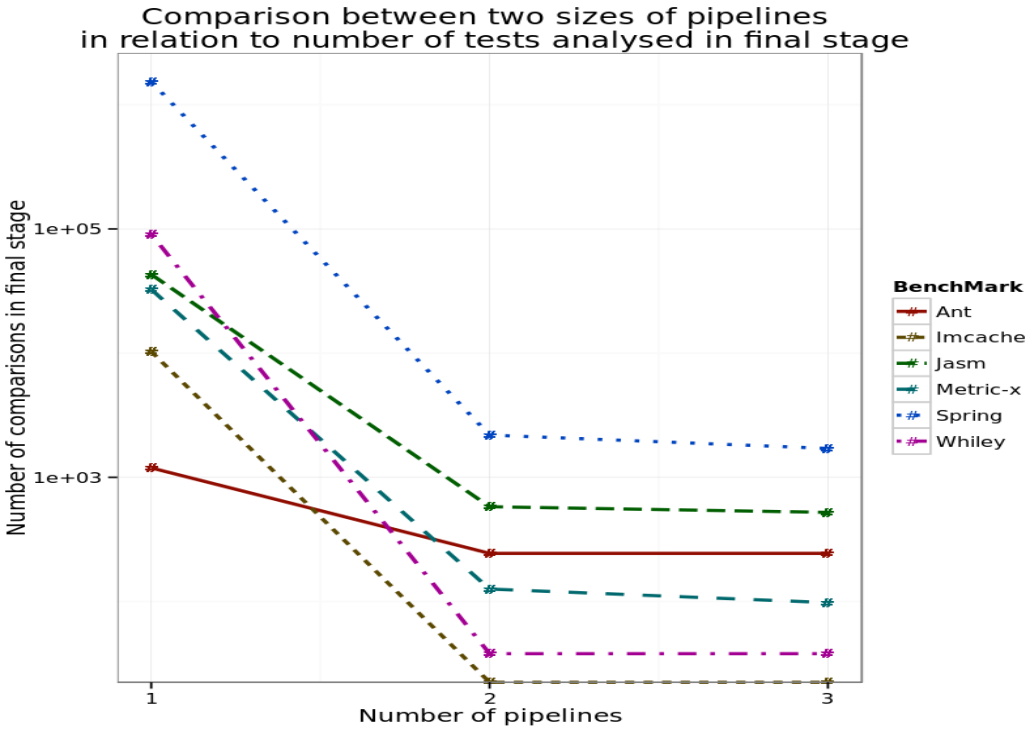

Figure 4.1: A figure showing the effect that using pipelines has on the number of comparisons that the final stage (Most computationally heavy) has to do.

4.4.4 Discussion

The first hypothesis states an increase in the pipeline size will improve the factors of interest. ~~An~~ The purpose of an increase in the pipeline size ~~aimed~~ was to reduce the number of

comparisons that the next stage had to perform. This ~~in turn decreased~~ reduction decreased the memory and time taken. ~~Immediately~~ Inspecting Figure **??**, it becomes clear that that pipeline size does improve the factors, but ~~doesn't~~ does not scale in every benchmark. ~~This becomes visible when examining Figure **??**.~~ Every two or three stage pipeline is a significant improvement on a single stage, however the majority of the benchmark's perform better ~~for~~ with a two stage over a three. These results imply that the benchmarks that took less time using a two stage, were spending more time on the second stage than they were saving from the reduced number of comparisons in the third stage.

Two situations which would cause more time to be spent on the second stage than was saved in the third stage are proposed. Firstly, each benchmark had it's own level of redundancy that was being looked for, this was dependent on how each benchmark reacted to different levels. Benchmarks with unit tests needed to have a higher level in comparison to benchmarks with end to end. Since each ~~benchmark~~ had their own percentage, this implies ~~that each benchmark also has its~~ they also have their own optimal settings for the second pipeline. The settings were chosen by experimenting with different options, therefore a inefficient second pipeline would have contributed to ~~it~~the result.

The second reason is that there may be little difference between the ~~second pipeline and third pipeline~~outcome from the second and third pipelines. An example is shown in ~~Appendix~~ Figure **??**. Examining the three stage pipeline, it shows the first pipeline causes a large reduction in the number of comparisons needed implying that the tests are highly separable. Inserting the second stage, a limited amount of reduction occurs while being computationally heavy and passes the majority of test cases into the third and most computationally heavy stage. Talk about cost here. This ~~leads to~~ creates a situation where the number of identified redundant test cases has little change from stage two to three. This leads me to accept the first hypothesis, that increasing the pipeline size improves the factors of interest, however the optimal length is different for each benchmark.

The second hypothesis is that there is no difference between the number of output test cases. As expected, each benchmark had no significant change. If there were a change between the two pipelines with the same final stage settings could imply two things. Firstly, there is a bug in the code. Secondly, the second pipeline stage is more specific than the ~~last~~third. Since both ~~the second stage and third stage~~ pipelines of size two and three share the same settings in final stage, ~~this would cause the last stage to be redundant and the redundant tests different to that of the two stage pipeline~~for the pipeline of size three, the second stage is a higher percentage than the final, this would cause a different output between the two sizes.

4.5 Experiment II - K Depth Comparison

4.5.1 Motivation

One of the abilities of the framework is to trace the K depth specified. This experiment is to determine the impact that altering the depth of K has on the overall cost.

Hypothesis 3 *~~Increasing the K Depth improves the precision is worth the trade off in~~ The precision improvement from increasing K Depth outweighs the performance decrease.*

4.5.2 Settings

There are three K depth's explored, one, two and three respectively.

- Difference between runs: K Depth

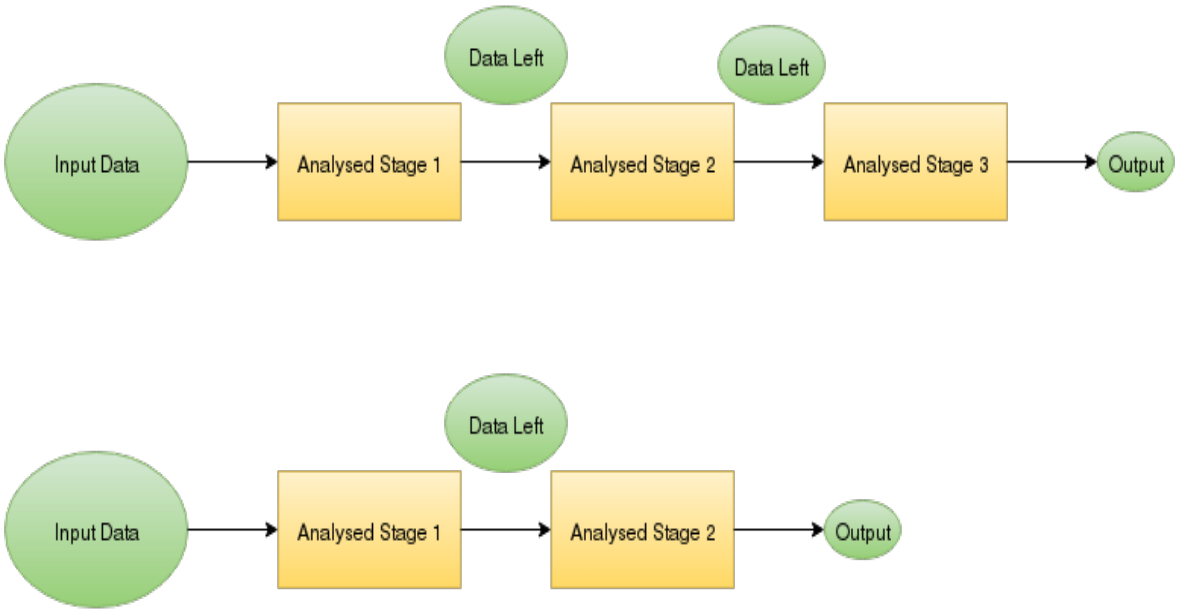

Figure 4.2: The circle size represents the data that is left in the pipeline. The figure shows that the difference that is between a two stage pipeline and a three stage only has limited room for a reduction in test cases identified. This shows that the time taken to run the extra pipeline costs more than it saves.

4.5.3 Results

Ant, Whiley and Spring appear to be the most responsive to the depth of K as shown in Figure **??** while the rest have limited change. Appendix **??** shows the differences in time taken. Overall, for every benchmark there wasn't a large amount of change between them however it shows that as the K depth increases, the time taken increases.

4.5.4 Discussion

Increasing the depth of the calling context increases the amount of data that is analysed, intuitively making it more difficult for test cases to be the same. This should be reflected through a reduction in the overall tests identified. The extra data used is going cause extra analysis, this implies an increase in the time taken as calling context depth increases. It is expected that the precision increase is worth the performance decrease.

Examining Figure **??**, it shows that the number of redundant tests is only slightly decreased as the K depth increases. This ~~may show~~ shows several things. Firstly, it ~~may imply~~ implies that when no other settings such as weighting or parameters are used, then the depth has limited effect on the redundant test cases identified. Secondly, the number of comparisons that were output from the first pipeline stage may cause there to be a limited differentiation. When the first pipeline stage outputs a low number of comparisons, then the final number of redundant tests identified would be similar regardless of the K depth specified, similar to the situation in Section **??**. This occurs in Whiley~~and~~, Metric-x, Imcache and Jasm.

Increasing the depth of K has varying effect's on the time taken to analyse a benchmark. The differences are shown in Appendix **??**. It shows that Ant, Imcache and Whiley have larger differences than the rest, however in the perspective of time these equate to around half a minute difference. Although increasing the depth has an smaller effect than expected

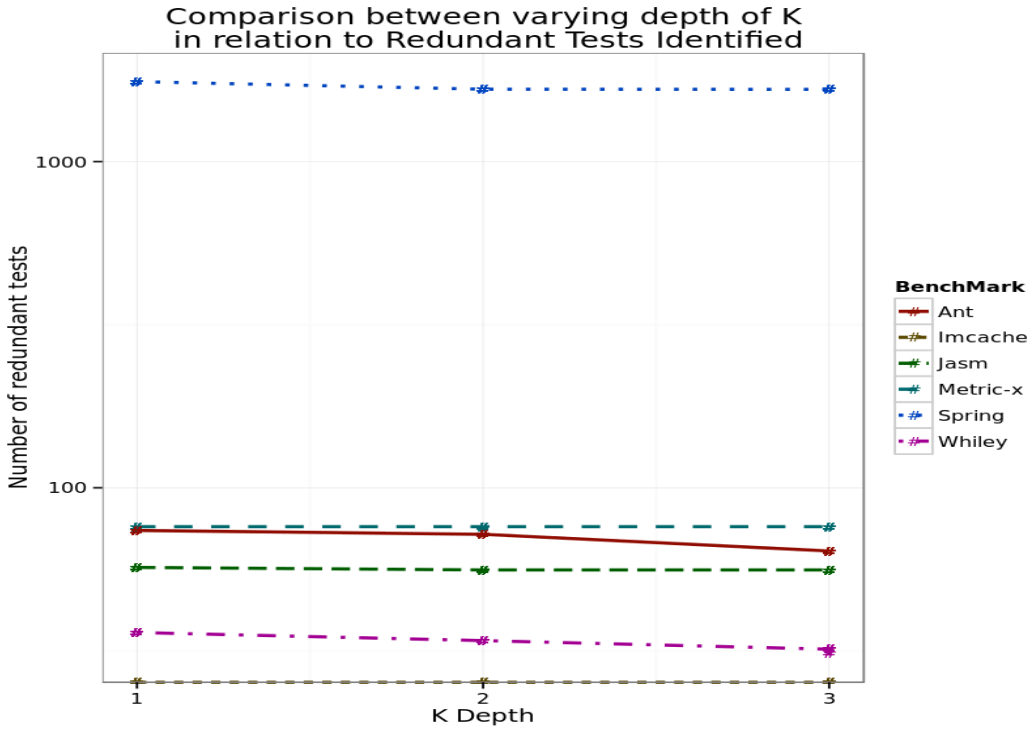

Figure 4.3: A figure showing the effect that a change in the depth of the calling context has on the number of redundant tests are identified.

on precision, the impact on time taken isn't substantial. Taking this into account, I can accept the hypothesis with the results implying that for some projects the calling context does not have to be particularly deep.

4.6 Experiment III - Parameter Comparison

4.6.1 Motivation

Parameters show the different contexts that a method was being called in. Retrieving this information gives us more confidence in determining whether two tests are redundant. ~~This shows the importance of parameters and the~~ The experiment explores the ~~increase in~~ impact on performance and precision.

Hypothesis 4 *Utilising parameters ~~increase~~ increases the time taken ~~of the framework~~ in comparison to a ~~standard~~ three stage pipeline.*

Hypothesis 5 *Utilising parameters ~~increase the precision of the framework~~ increases the precision in comparison to a ~~standard~~ three stage pipeline.*

4.6.2 Settings

The use of parameters is compared directly to the pipeline size three settings.

- Difference between runs: Parameters used during analysis

4.6.3 Results

The significant table for comparing parameters and no parameters is shown in Table **??**. It shows that there was a negative relation (increase) for every benchmark in regard to time taken. Every benchmark had a significantly positive effect (decrease) on the number of redundant test cases identified. This is not the case in Imcache due to it having zero redundant test cases identified in both, therefore no difference between the two.

	Total Time	**Redundant Tests Identified**
Whiley	+	-
Jasm	+	-
Ant	+	-
Spring	+	-
Imcache	+	=
Metrics-x	+	-

Table 4.4: A table showing the significant relationship between the use of parameters and no parameters for each benchmark

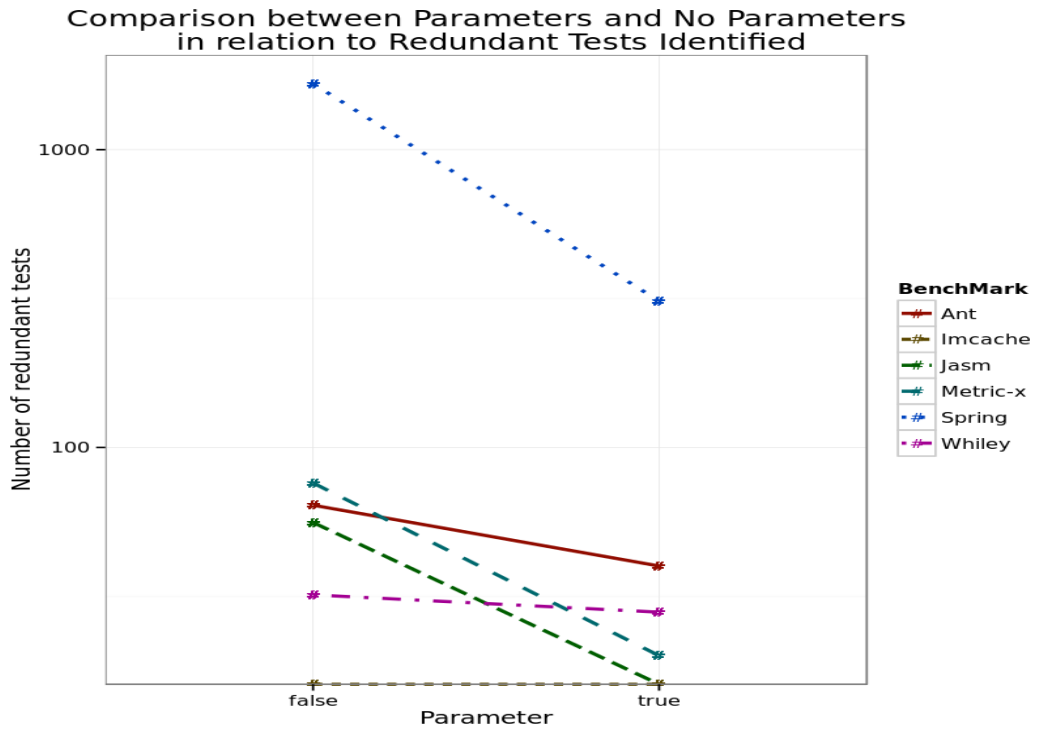

Figure 4.4: A figure showing the effect that using parameters has on the number of redundant tests are identified.

4.6.4 Discussion

Two factors have to be taken into account when discussing the impact of parameters on the time taken. Firstly, it is intuitive that analysing the extra data generated increases the time. The amount of increase is interesting, parameters only add extra computation when the method execution of the given K is exactly the same. For example, if test execution is

$A- > B- > C$ is compared to $A- > B- > F$. Then the parameter won't be taken into account and therefore have limited effect on the time taken.

The amount of time that the VM spends garbage collecting would also increase the time taken, due to increased amount of information in memory, the garbage collection process will have to execute more frequently for the analysis to continue to run. This would cause non-deterministic attributes to be increased. Examining Appendix **??** – Ant, Imcache, Jasm and Spring appear to affected by this due to the increased variance. This creates the hypothesises that parameters ~~decrease~~ increase the precision and ~~increase~~ the total time taken.

The significant Table **??** matches what is expected. For every benchmark, using parameters significantly increases the time taken. Examining Appendix **??** shows how the benchmarks react with more detail. The most interesting would be ~~JASM~~Jasm. Without parameters, it analyses the data in less than one minute, with parameters, it takes just under five minutes. This may be indicative of the results discussed in Section **??**. With K depth having little effect on a reduction in comparisons, the number of tests that match the K depth of three is higher than expected. This effect led to parameter information being examined more often, sequentially causing an increase in time taken. A factor that may have an accumulate effect is related to the setup methods. If the set up methods are a large majority of the method calls, and match each other for the K depth, this would further increase the time taken. This allows us to accept hypothesis 4.

Every benchmark caused a significant decrease in the number of redundant test cases. Looking at Figure **??**~~shows~~ , it confirms that every benchmark reacted with a substantial decrease in redundant tests identified, with Whiley being the least affected. This reiterates the discussion in Section **??** that K depth is not enough of a factor to identify redundant test cases. It is expected that ~~identified tests decreases~~the number of false positives decrease, inspecting Table **??** and **??** gives insight into the types of tests identified. The Whiley benchmark coding shows that parameters remove the limited redundant tests as well as different array values. The Metric-x coding ~~shows the influence that parameters has. Parameters reduce the~~ demonstrate a reduction in the number of the different parameter value tests identified substantially, from 52 to 8. ~~These coding~~ Taking these coding tables into account, allow us to accept hypothesis 3.

4.7 Experiment IV - Weighting Comparison

4.7.1 Motivation

Utilising weighting is an attempt to remove any false positive tests identified. The experiment attempts to identify if weighting has any potential to remove these false positives, while exploring the impact on performance.

Hypothesis 6 *Weighting decreases the number of false positives compared to a standard three stage pipeline*

Hypothesis 7 *Weighting decreases the time taken compared to a standard three stage pipeline*

4.7.2 Settings

The use of weighting is compared directly to the pipeline size three settings.

- Difference between runs: Weighting Used

4.7.3 Results

The significant table for comparing the use of weighting is shown in Table **??**. There are a mixture of results in regard to the total time taken – Whiley, Ant and Imcache had a significantly negative relation and weighting increased the time taken to analyse. Jasm, Spring and Metric-x had a significantly positive relation and weighting decreased the time taken to analyse. The majority – Whiley, Ant, Jasm and Metrics-x had a significantly positive relation in regard to the number of redundant tests identified, showing a decrease in the number identified when weighting was applied. Spring was the only benchmark where weighting had a significantly negative impact.

	Total Time	**Redundant Tests Identified**
Whiley	+	-
Jasm	-	-
Ant	+	-
Spring	-	+
Imcache	+	=
Metrics-x	-	-

Table 4.5: A table showing the significant relationship between the use of weighting and no weighting for each benchmark

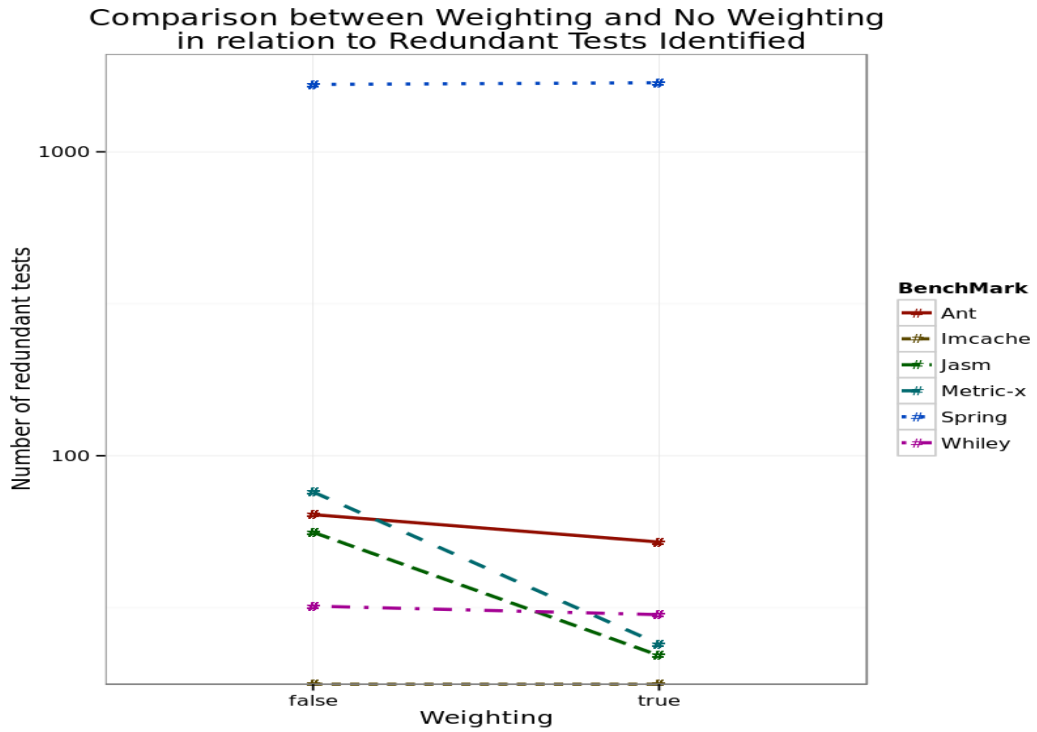

Figure 4.5: A figure showing the effect that using weighting has on the number of redundant tests are identified.

4.7.4 Discussion

As previously discussed in Section **??**, much of the related work encountered difficulties with test cases sharing setup and teardown method calls, resulting in a high false positive rate. Intuitively this makes sense when the test cases are small in comparison to the setup and teardown methods where the setup and teardown methods ~~are a larger~~ represent a large majority of the ~~tests~~ test data. The approach of removing a portion of the most executed method calls meant that the amount of data that would be compared decreases. This should reflect onto the results by showing a decrease in the time taken, as per hypothesis 7. The effect that it has on the number of redundant test cases should be dependent on the benchmark when weighting is used. This is because removing the most common tests should imply an decreased false-positive rate, but also may increase the number of actual redundant tests picked up. ~~To examine~~ The precision is of interest when examining the cause effect of weighting, ~~the precision to validate or invalidate~~ this is reflected by hypothesis 6.

Table **??** shows a mixture of results in regard to the total time taken. Two out of the three benchmarks that had a significant increase in the total time were small benchmarks. This may imply that the size of the benchmark has some relation to the effect of weighting on the time taken. One reason ~~may be~~ is that weighting is calculated once per test case per analysis stage. The weighting calculation has a linear relation with the number of test cases. In comparison, every test case is compared to every other, therefore a factorial relation with the number of test cases. Since the factorial relation is closer to linear, the smaller the number of redundant test cases. Then the smaller the number, the more impact the linear weight calculation has on the overall time taken. This may explain a relation between size and time taken. Examining the overall impact from weighting on the time taken in Appendix **??** allows us to invalidate hypothesis 7 as there is no general consensus between benchmarks.

A reduction in the false positive rate was weightings primary goal. Spring was the only benchmark that significantly increased the number of redundant test cases identified. The other benchmarks had a significant decrease. At first, this makes it appear that by using weighting it may solve some of the issues that were identified in [**?**] [**?**]. To confirm this, examining Table **??** and **??** will give insight into the effect of weighting. Comparing the two columns for the Whiley coding, Pipeline 2 and Weighting, the only difference is the removal of the two limited redundancy test cases that were picked up by Pipeline 2. The weighting is able to remove the test cases that have similar set up and tear down methods, but fails to reduce the number of other redundancies. By removing the method executions that were common throughout the benchmarks, this ~~lead~~ led to a decrease in ~~false-positive~~ false positive tests identified. The Metric-x coding paints a similar story. Weighting reduces the number of parameter value redundancies identified as well as the limited redundancies, however does not completely remove them. This is interesting and implies that the number of setup and tear down methods that remained in the data analysed was still a large portion of the data. We are able to accept hypothesis 6, although work remains to improve the weighting method to remove redundant tests completely.

4.8 Experiment V - Weighting and Parameter Comparison

4.8.1 Motivation

Combining weighting and parameters is an attempt to reduce the number of false positives, and at the same time ~~increasing the confidence of~~ increase the confidence that the redundant tests ~~being~~ are truly redundant.

Hypothesis 8 *A combination of weighting and parameters increases the precision in comparison to weighting and parameters separately*

4.8.2 Settings

There are several comparisons that occur in this experiment. The use of weighting with parameters is compared directly to both, the pipeline size ~~three settings~~ two settings as well as weighting and parameters separately.

- Difference between runs: Weighting and Parameters combined

4.8.3 Results

The relevant significance table is shown in Table **??**~~showing~~ , it shows a comparison between the combination of techniques and pipeline of size two. It shows that for every Benchmark apart from Imcache, there was a negative relation for the time taken, ~~showing~~ indicating that the time increased when using weighting and parameters for the majority of benchmarks. In contrast to this, every benchmark had a positive relation to the number of redundant tests identified. Figure **??** indicates the number of tests identified for each benchmark in comparison to weighting and parameters separately. Each benchmark reacted differently to the use of weighting and parameters combined – Spring, Jasm and Metric-x had large decreases compared to the weighting with the rest being similar between the three types.

	Total Time	**Redundant Tests Identified**
Whiley	+	-
Jasm	+	-
Ant	+	-
Spring	+	-
Imcache	-	=
Metrics-x	+	-

Table 4.6: A table showing the significant relationship between the use of weighting with parameters and pipeline size two for each benchmark

4.8.4 Discussion

Using parameters to increase the confidence, and weighting to remove common method executions should increase the precision of the framework. To appraise the combination, Figure **??** displays the comparison between each of the three ~~types~~techniques. Weighting identified more redundant tests in every benchmark ~~but~~ except Ant. Parameters produce similar results to the combination, apart from in Jasm and Metric-x. Whiley coding in Table **??** show no difference between parameters and the combination. Metric-x's coding Table **??** hints that the combination may have some appeal by removing all eight limited redundancy tests. At the same time, the similar tests are also removed which is a negative outcome. This suggests that the combination improves the precision for some benchmarks, but not for others. This conclusion means hypothesis 8 can not be accepted.

Combining weighting and parameters presents interesting time taken results. Table **??** shows that it is nearly identical to the parameters significance table with the only exception being that the Imcache benchmark changed from significant increase in time to a significant decrease in time. This indicates that parameters have a stronger effect than weighting on the

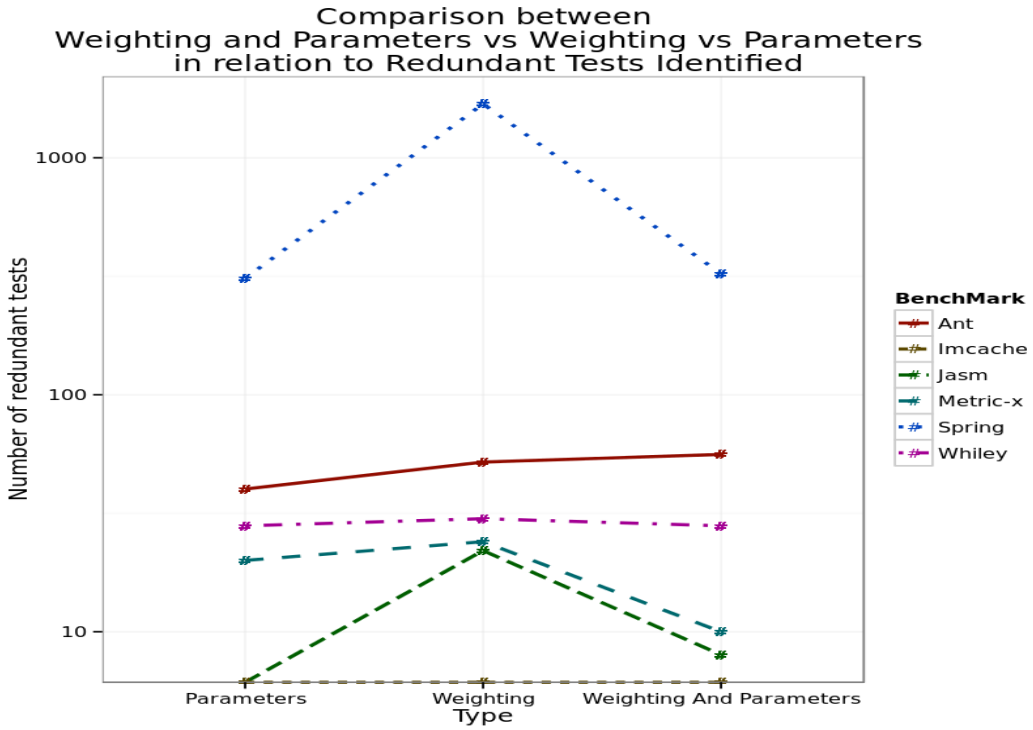

Figure 4.6: A figure showing the effect that using weighting and parameters has on the number of redundant tests are identified.

final outcome. This is an unexpected result. As previously discussed in Section **??**, one of the reasons that parameters increase the time taken is due to the similar setup and tear down method calls. With weighting removing a large portion of the common method executions it would be expected to cause a decrease in the number of comparisons using parameters. Looking at Appendix **??**, this appears to be true for Jasm, Metric-x, Spring and Imcache. However, for Whiley and Ant benchmarks the time to calculate the weighting information is taking longer than the comparisons saved.

4.9 Redundant Test Case Coding

	Whiley			
Types of redundancy	**Pipeline 2**	**Weighting**	**Parameters**	**Parameters and Weighting**
Different Equation Value	6	6	6	6
Different Equation Sign	8	8	8	8
Different Array Values	2	2	0	0
Same	10	10	10	10
Limited Redundancy	2	0	0	0
Rearranged Equation	2	2	2	2
Extra if statement	2	2	2	2
Total	32	30	28	28

Table 4.7: A table displaying a list of coding's for the Whiley Benchmark for four of the different techniques used.

	Metric-X			
Types of redundancy	**Pipeline 2**	**Weighting**	**Parameters**	**Parameters And Weighting**
Different Parameter Value	52	10	8	4
Different Object Type	6	2	2	0
Different Array Values	8	6	4	6
Similar	2	2	0	0
Limited Redundancy	8	4	6	0
Total	76	24	20	10

Table 4.8: A table displaying a list of coding's for the Whiley Benchmark for four of the different techniques used.

Chapter 5

Conclusions and Future Work

5.1 Future Work

It may be that a lot of tests subsume another rather than are redundant. The project has explored the use of higher level information than previously explored for identifying redundant test cases while examining how pre processing and changing the depth of information retrieved can impact the results. Although the output of the framework is adequate in the sense that there is a low number of tests identified, where the developers can easily look over and remove them as they see fit. As discussed in Section **??** one of the limitations is identifying when one test subsumes another. Using a statement coverage information with weighting would provide an interesting set of results. It would allow for several questions to be answered in regard to the performance of method execution vs statement information. The results would be expected to build off those achieved in Maurer et al. [**?**] and Robinson et al. [**?**].

One of the issues identified in Section **??**, there was difficult identifying the best pipeline variables to use. In Section TODO there are guidelines that are produced which were created based on the benchmarks size and types, although this would highly likely to not be universal. Finish if I produce guidelines.

Memory consumption hampered the number of test's that could be run for some of the benchmarks. To overcome this the use of a database could be employed. This would increase the time taken as it would be retrieving the data from non-volatile memory store, but would allow for an increased set of data to be examined, potentially using statement information and method execution data where the method execution data is used during the n-1 pipeline stages and the statement information is explored once the number of potential redundant tests is reduced. I feel this should be switched around. Combining the statement and method execution should start then why and how.
Database usage

5.2 Conclusions

The use of method execution data seemed like a good candidate to identify matching tests
Weighting and parameters show potential in helping to limit the number of redundant test cases
Calling tree has limited effect beyond depth of 1 when by itself Pipelining saved substantial time

Talk about framework Talk about strategies

- Explore the use of method execution details
- The impact of weighting on performance
- The impact of parameters on performance
- The impact of calling context depth on performance
- The impact of pipelining the data on performance

5.2.1 Can method execution details give a good overview of the level of test redundancy?

It is difficult to make a direct comparison between using statement information and higher level method execution details without implementing and comparing both on the same benchmarks. Therefore, the level of redundancy identified will be evaluated by the potential redundancy identified as well as the type of redundancies. It is important to take into consideration both. Knowing the number of tests allows us to see how method executions handle different types of tests, unit vs end to end respectively. The potential redundancy gives us insight into the types of redundant tests identified, giving information on which types of test redundancy is easier to pick up and which is harder when using method execution.

The use of method execution allows us to look at test cases that produce a larger amount of total information, in particular ones which are end to end tests (Whiley) which produce a large amount of data. It would be interesting to see the comparison between statement information and method execution details. It appears to be intuitive that statement coverage would give more accuracy in regard to determining whether two test cases are redundant but this would come at a cost of time and memory consumption. What is the optimal trade-off for this situation? Maybe talk about it here?

By using a higher abstraction of information, there appears to be no relation to the time taken and the number of redundant test cases with the two types of test cases, unit and end to end respectively.

Examining the Table **??** and **??**, allows for an comparison to identify the difference between end to end tests and unit tests while using method execution details. It appears that the types of the tests may react differently to the settings used. For the end to end test of Whiley, where the size of a single test case is larger than Metric-x, it appears to respond to the use of parameters and weighting in a positive manner. Where the tests that are picked up as limited redundancy by the Pipeline 2 are both removed when weighting and parameters are used. In comparison to the unit test case. The unit test appeared to not respond either to weighting or parameters by identifying tests that had limited redundancy. When combining both of these, it removed the tests that were identified, but had limited redundancy. Explain why it might be happening This may reflect on the amount of pre processing and information needed for the different types to get identified. This would mean that the larger bench marks can be analysed faster ?

5.2.2 Can we improve the overall performance with weighting?

More Weighting attempts to improve the accuracy of the redundancy by removing false-positives while improving the time taken to analyse.

Examine the coding tables, state that using weighting can help

5.2.3 Can we improve the performance using parameters?

The answer to the question depends on the type of parameters that are used. Initially the only parameters that were being retrieved were the primitive types. The reason was due to the easy nature of retrieving the values of them. This lead to a less than expected outcome as the majority of parameters involved object types which contained the important information that could be used to increase the separability of tests. By using reflection to retrieve the fields of these objects allowed more insight into the content that the parameter objects were holding and the nature of them, but it also meant more data had to be held and analysed resulting in an increased time taken. There exists this trade off between time taken and confidence of redundancy when using parameters. The reason for running a framework such as this is to reduce the number of redundant test cases, therefore the majority of situations will prefer a higher confidence of redundancy over time taken. The answer is then to use reflection to retrieve the parameters.

Examining the weighting and parameter time taken in Appendix **??**, it shows that when weighting is taken into account with parameters then for Jasm, the time taken goes from just under five minutes to just under one minute.

Examine the coding tables, state that using parameters improves performance

5.2.4 Can Calling Trees we used to better identify redundant tests?

The result was unexpected. It showed that as the depth of the calling context increased, there was limited level of reduction in the number of redundant tests identified. Although there was a change, it was lower that what was expected. A possible reason for this was after a certain number of calling context, the tests that were going to be different, were already different and increasing the size of the calling tree did not affect it. This may imply that using a calling context depth of two is enough to generate a list of redundant tests similar to a depth of two. Todo check with K Length Comparison - might be talking about same things

Using a calling tree allows for the analysing tool to better identify redundant tests as shown by the comparison above where the one, two and three calling sizes are compared.

5.2.5 How does pipelining impact the time taken?

Pipelining was one of the critical aspects of the project. It not only lead to a decrease in the time taken but also the memory consumed. Like calling trees, there is an optimal number of pipelines before they are causing a larger increase in time than they save. We can see in the results that each check results benchmark responded differently to the length of the pipeline. This may have been due to not finding the optimal pipeline variables for the second pipeline causing a large number of comparisons still being needed to be done by the last pipeline. Another reasons may have been due to the nature of the tests where they were difficult to separate until you used more data such as a list or calling context.

Maybe guidelines for the settings that each type of test should use?

5.3 Justify your approach

Unsure if I should put anything ehre

5.4 Critically evaluate your study

A major limitation to the study was volatile memory. For test suites such as Whiley, storing the parameter data and calling context endured a high level of volatile memory usage, limiting the amount of tests for Whiley that were able to be retrieved (Limited to valid tests). An approach that should have been considered with more thought was the use of a database. Although this would have introduced difficulty reducing the non-deterministic behaviour to a minimum. A major factor would be the difficulty around executing the analysis on a grid system and replicating the conditions each run.

Some of the benchmarks that were chosen contain multiple unit tests within one test case. The framework described throughout this paper would not be able to pick up when a test case subsumes another. The frame work only looks at the total method set, and for each of the singular unit tests, there would need to be a skewed proportion of the conjoined test case in regard to the total method executions. This means that unless one of the unit tests within the conjoined test case was arbitrary larger than the other, roughly 98:1 lines of code then it would not be picked up as being redundant, unless there was another conjoined test case similar. This is where the other approaches identified in Chapter **??** would be more effective. By looking at the statement information, it is easier to determine when a test is a subset of another and have more confidence on it being a redundant test case in comparison to looking at solely the method execution details. Need to explain this better.

As discussed in Section **??**, some benchmarks performed better with a two stage pipeline in comparison to a three stage, and vice versa. This may have been due to not being able to identify the optimal parameters for each benchmark, although there were multiple different parameters tested to explore each benchmark's optimal. Maybe explore how I decided the optimal ?

Appendices

Appendix A

The effect of pipeline size in regard to the time taken

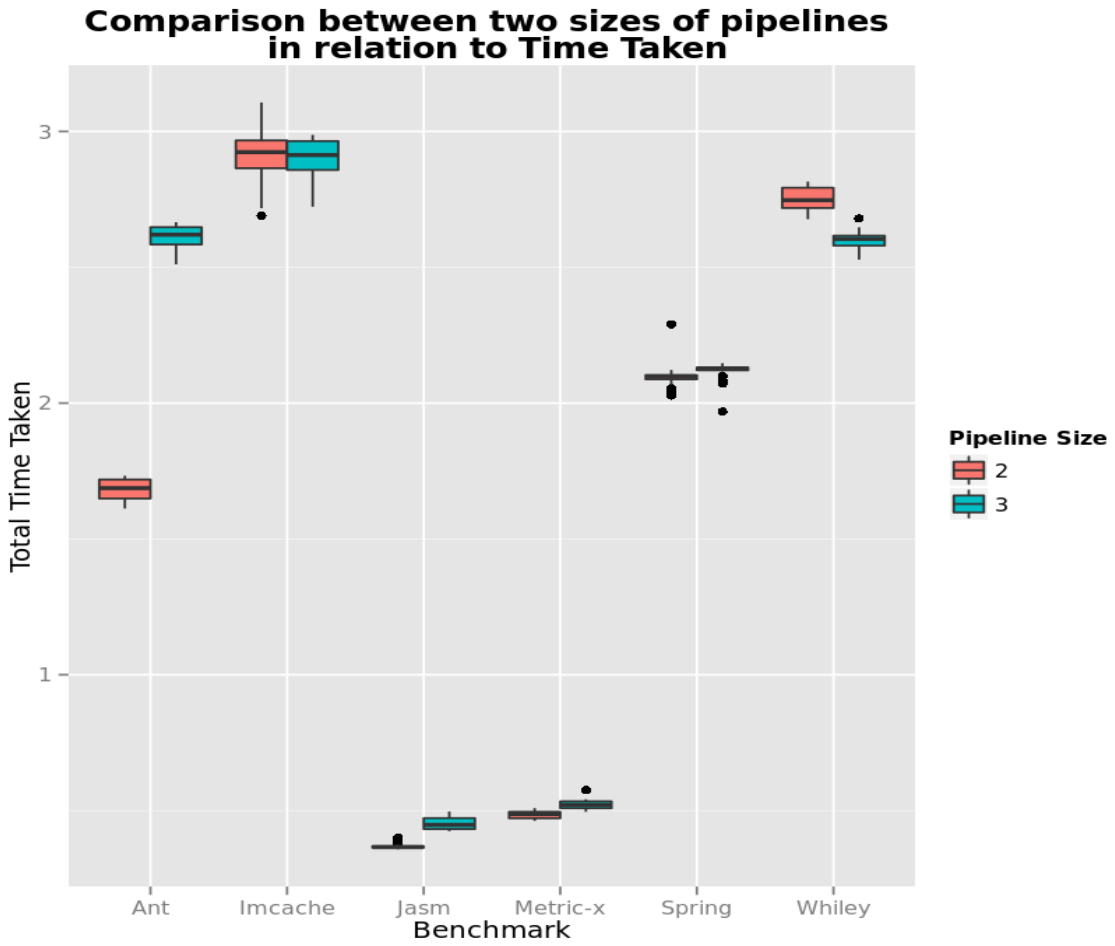

Figure A.1: A figure showing the relationship that using different pipeline sizes has on the total time taken to analyse the data.

Appendix B

~~Pipeline linear scaling~~

~~The circle size represents the data that is left in the pipeline. The figure shows that the difference that is between a two stage pipeline and a three stage only has limited room for a reduction in test cases identified. This shows that the time taken to run the extra pipeline costs more than it saves.~~

Appendix B

The effect of K Depth in regard to the time taken

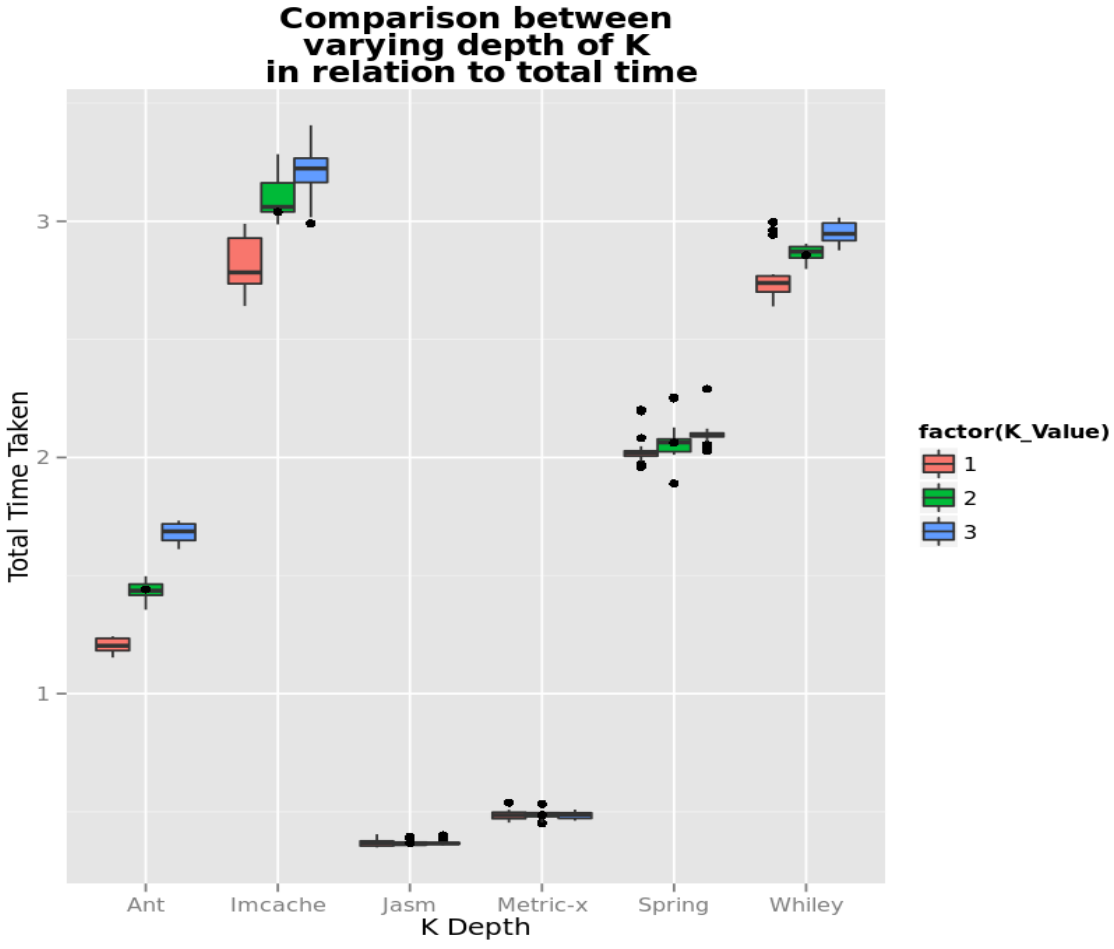

Figure B.1: A figure showing the relationship that using a different K Depth has on the total time taken to analyse the data.

Appendix C

The effect of parameters in regard to the time taken

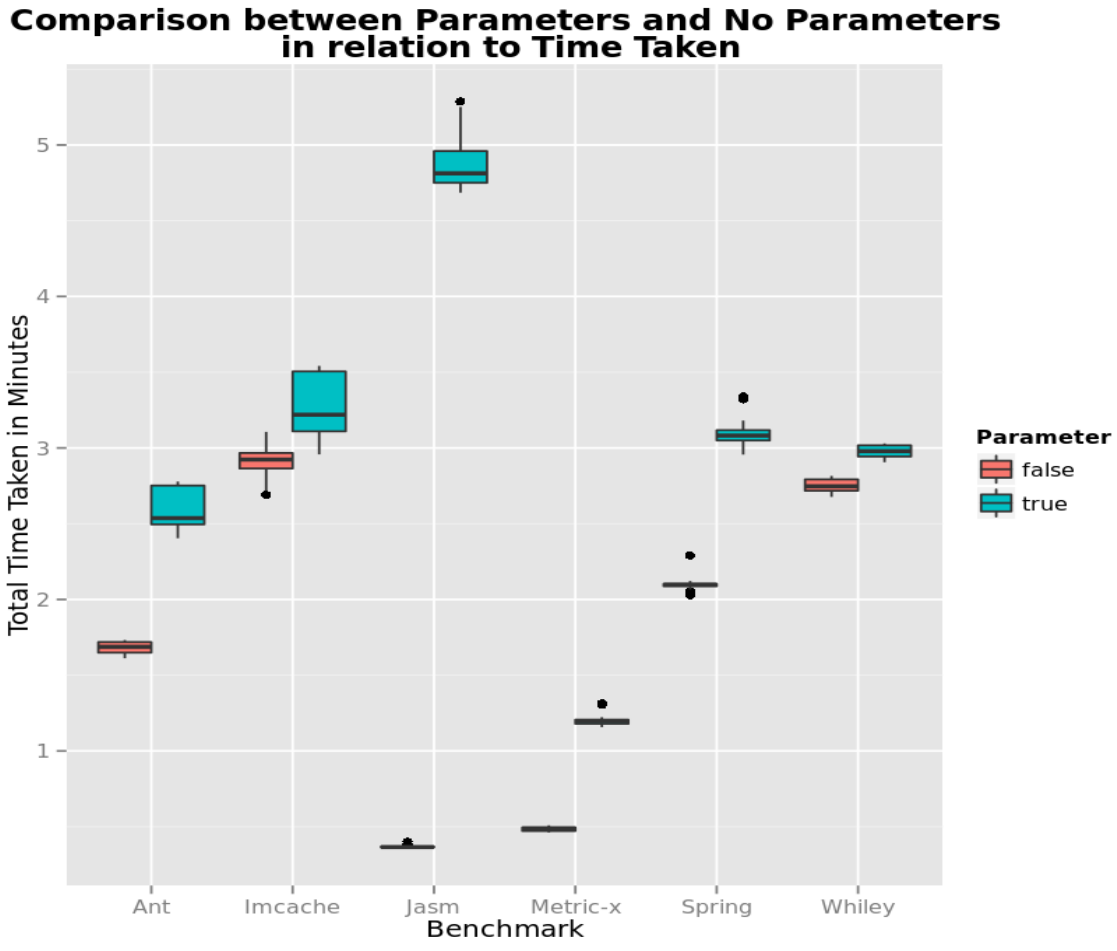

Figure C.1: A figure showing the relationship that using parameters has on the total time taken to analyse the data.

Appendix D

The effect of weighting in regard to the time taken

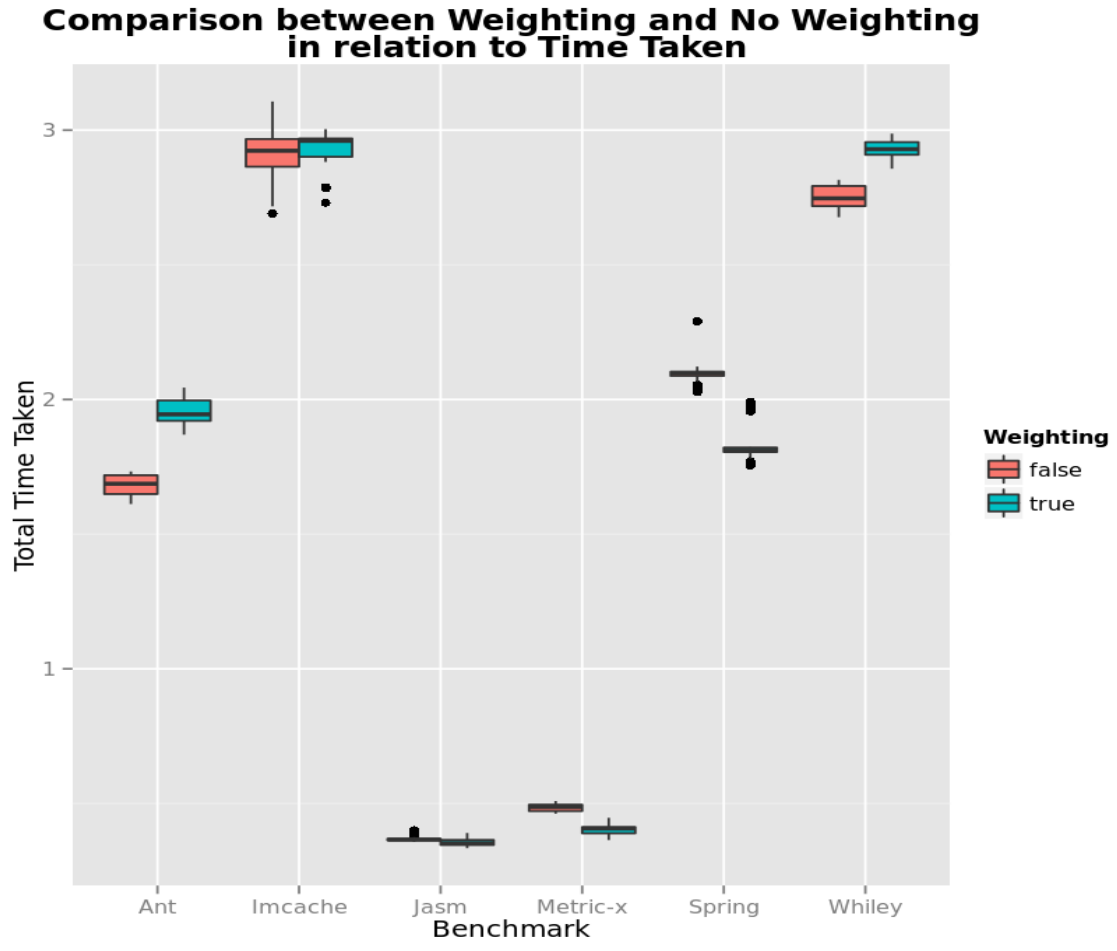

Figure D.1: A figure showing the relationship that using weighting has on the total time taken to analyse the data.

Appendix E

The effect of weighting and parameters combined in regard to the time taken

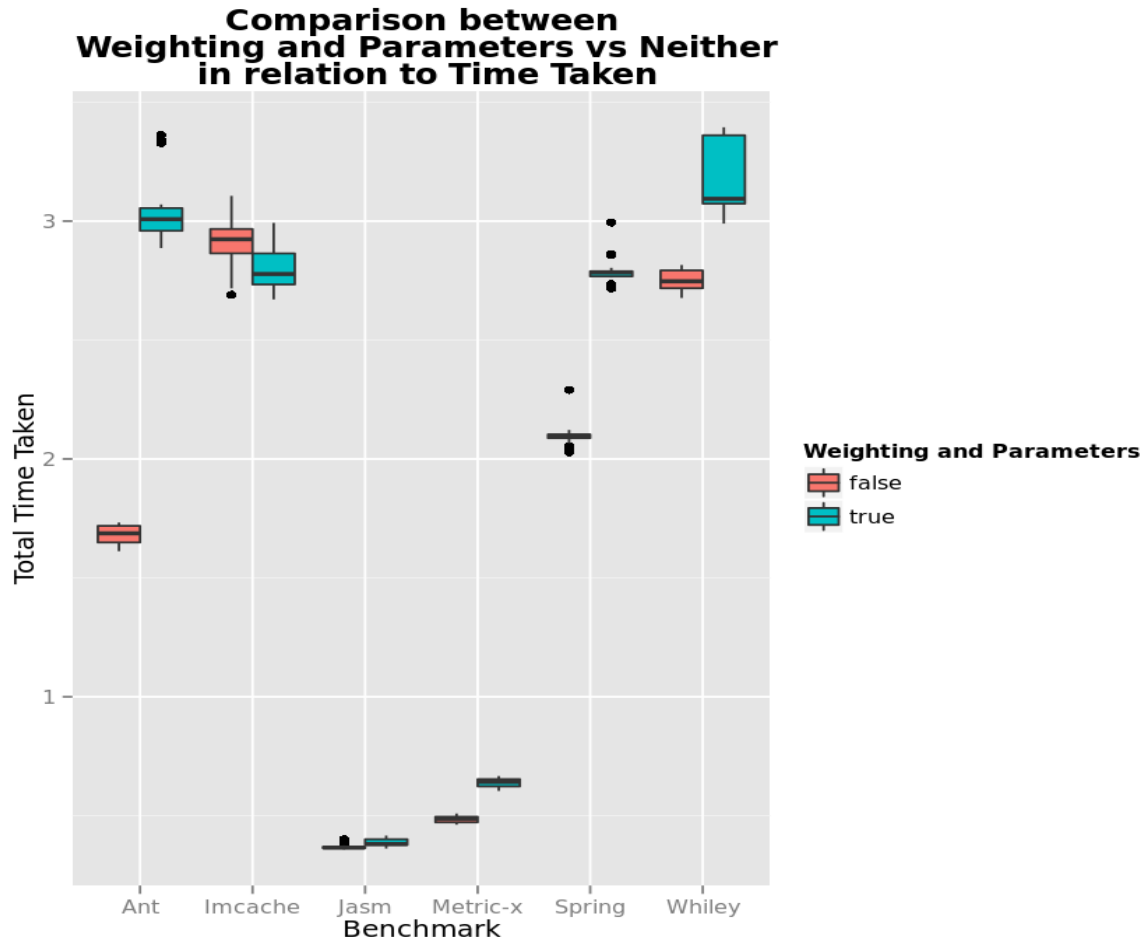

Figure E.1: A figure showing the relationship that using weighting and parameters combined has on the total time taken to analyse the data.

Appendix F

The effect of weighting and parameters vs parameters in regard to the time taken

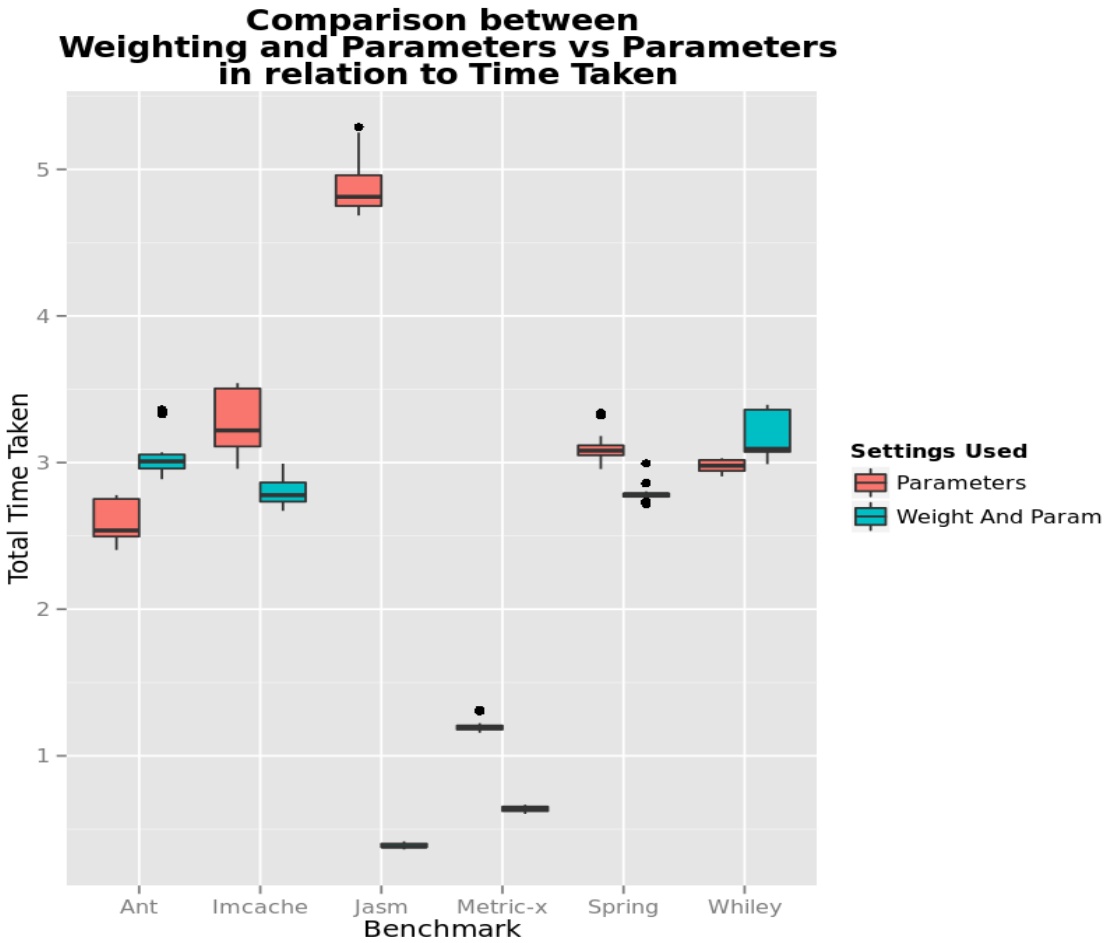

Figure F.1: A figure showing the relationship that using weighting and parameters combined vs parameters alone has on the total time taken to analyse the data.