VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematics, Statistics and Computer Science
Te Kura Tatau

PO Box 600
Wellington
New Zealand

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

The Java Collections Profiler

Simon John Hibbard

Supervisor: Dr David J. Pearce

Submitted in partial fulfilment of the requirements for
Bachelor of Information Technology.

Abstract

This document contains a full report on the Profiling Collections in Java project. The project focuses on the development of a tool that profiles the use of different implementations of the `Collection` interface in Java and allows us to analyse their use with regards to efficiency. It is our hypothesis that the efficiency, and therefore performance, of software may suffer as a result of poor decisions made by the developer when choosing which `Collection` implementations to use in their code. Our profiling tool will allow us to determine whether or not this is true.

Contents

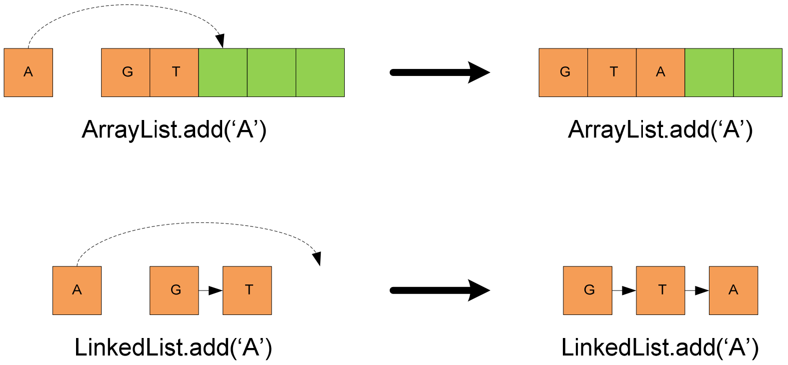
Chapter 1
Introduction

1.1 Motivation

Almost every piece of software that is written today incorporates the use of structures that represent collections of data. Sets, lists, queues, stacks and maps are all examples of such structures. In Java, all of these data structures fall under the common interface of `Collection`. There are a number of implementations of these data structures available to software developers in common libraries for most popular languages. The availability of these implementations saves the developer a lot of time but does not remove them from the process entirely. They must still choose which implementation to use and this choice can have severe effects on the performance of the software.

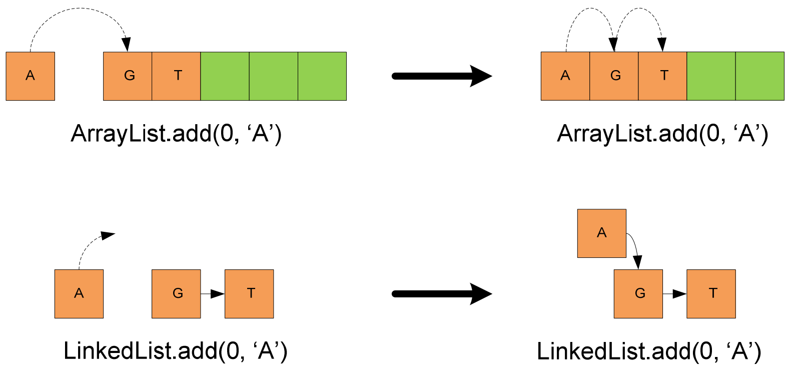
1.2 Example

Let us consider the two different `add` methods in the `List` interface in Java. The first method, `List.add(<E>)`, is inherited from the `Collection` interface. It takes a single argument – the item to add to the list – and inserts this item at the end of the `List`. The following diagram illustrates the process for the `ArrayList` and `LinkedList` implementations:



As the diagram shows, `List.add(‘A’)` has a complexity of O(1) for both `ArrayList` and `LinkedList`. Occasionally `ArrayList` will incur a larger cost for this operation. This is the case when the array fills up and it must copy its contents to a new, larger array. In the average case though, the complexity is O(1). We can see quite clearly that the performance characteristics for this operation will be about the same, whichever implementation is chosen.

The second `add` method is specific to the `List` interface and takes two arguments in the form of `List.add(x, <E>)`. The extra argument in this version specifies the point in the `List` at which the new item is to be inserted. Again, processes for each implementation are illustrated below:



When we consider the `List.add(x, 'A')` method as shown above, a gap appears in the performance of the different implementations. `LinkedList` retains its O(1) complexity while `ArrayList` degrades to O(n). The reason that `ArrayList` becomes so much more expensive is that it stores its data elements in a contiguous linear space. Every time it tries to add an element at a position that is not the end of the list, it must first shift every successive element one position to the right.

`LinkedList` does not suffer from the same problem as `ArrayList`. Rather than using a contiguous linear space to store data elements, it uses random free memory spaces and can therefore create new space without having to rearrange anything. To place the new element in the list, pointers from each of its new neighbours are simply redirected to point at the new element. This does, however, introduce a new cost – finding the neighbours. To find position `x` in a `LinkedList` you must first find position `x-1`. This means that finding the place at which you want to insert the element becomes O(n). While this can add to the cost in some situations, there are other situations where it is not a problem.

For example, consider the case of a bulk insertion of items at specific points in a `List`. Using an `ArrayList` implementation the complexity is O(kn). This is because for each of the *k* items being added, there is a cost of *n* involved in moving down the successive items to make room. The code for achieving this bulk insertion may take the form:

```
for(int i = 0; i < items.length; i++){
      list.add(positions[i], items[i]);
}
```

In the example above, the 'positions' array contains integers representing the positions in the `List` where we want to add the items. The 'items' array contains the items themselves. These two arrays are related in such a way the `positions[i]` is the position at which `items[i]` is to be added.

If a `LinkedList` implementation is used, the complexity falls to O(k+n). This is because `LinkedList` does not have to shift existing items every time it wants to add a new one. It can use the ListIterator class to its benefit so that it searches the list at most once to find insertion points. It can then use the ListIterator.add(<E>) method to insert the desired item at the current point. This code may take the form:

```
ListIterator<E> itr = list.listIterator();
for(int i = 0, j = 0; i < items.length; i++){
      while(j < positions[i]){
            itr.next();
            j++;
      }
      itr.add(items[i]);
}
```

There are some scenarios where finding a position in a `LinkedList` is not a complex operation. These include the case where the `List` is being added to at the very beginning, and the case where the `List` is being iterated over anyway. Neither have any added cost in finding the position at which to insert the new element. They may sound like isolated and unlikely scenarios, but there are several imaginable situations in which they might occur.

The point is that operations have differing costs associated with them which are defined by the specific implementation chosen by the developer. If we can identify operations that have implementation-dependent complexities and monitor their usage, we may be able to help developers to make the best decisions with their choice of implementation.

1.3 Hypothesis

It was our hypothesis that developers do not always make the best decisions with their choice of `Collection` implementation. This may be for a number of reasons:

- The developer uses one type of implementation all of the time, out of habit.
- Little or nothing is known about the performance characteristics of different `Collection` implementations by the developer.
- The developer is not sure exactly how the program is going to use the `Collection`. A common example of this is the development of libraries. If the developer is creating a library to be used by other developers, he or she cannot possibly predict exactly how that library will be used.

Through this project, we hoped to test whether this hypothesis is true or not.

1.4 The Java Collection Profiler

To test our hypothesis we decided to develop a tool for monitoring the usage of `Collection` objects within Java programs. This tool should be able to assess the usage of `Collection` objects within Java programs without the need to modify the source code of that program. It should be easy to use and provide an output that allows developers to easily see which points in their code have potential for performance gains through `Collection` implementation selection.

The tool that we developed is called the Java Collection Profiler. It is designed to be easy to modify to allow for developers to monitor exactly what they want in their programs. It is implemented in AspectJ and Java and has been tested to work perfectly with a number of Java programs. The results produced by the tool are informative and clear, and we have managed to achieve significant performance enhancements in some programs as a direct result of using the tool.

What follows is a description of the development of the tool; along with examples of the results it has produces and an analysis of its effectiveness.

Chapter 2
Implementation

2.1 Methods of Interest

Choosing which methods our profiler was to intercept required a degree of thought and consideration. An obvious approach would be to simply intercept all `Collection` interface method calls. This approach has a number of drawbacks:

- The more methods we intercept, the more overhead there is associated with our profiler.
- For every method we intercept, there is another piece of data that must be displayed in the output of the profiler. This can lead to clutter and make the results difficult to interpret and understand.
- Not all methods are relevant to our goal of performance improvements.

The first two points are relatively straight forward – it takes more effort to gather and understand more points of information. This holds true for a great number of scenarios, outside of profiling and outside of computer science.

The final point is quite specific to our task. While all methods incur a cost in performance, not all have costs that differ with implementation. We are interested in comparing the performance of different implementations so only methods that differ in cost with implementation are of interest to us.

To give an example of the distinction between 'interesting' methods and irrelevant ones, let us again consider the `List` interface in Java. There are three main implementations of the `List` interface provided in the common Java library – `ArrayList`, `LinkedList`, and `Vector`. `ArrayList` and `Vector` have very similar performance characteristics and so `Vector` is largely ignored throughout this project. `LinkedList`, however, uses a completely different underlying data structure. Because of this, there are a number of methods that differ in performance when compared to those in the `ArrayList` implementation.

We have already seen how a `LinkedList` implementation can perform better than an `ArrayList` in some situations. Conversely, the `List.get(x)` method is a much cheaper method when an `ArrayList` is used over a `LinkedList`. This method retrieves an item from a specific point in a List. The associated complexities are O(1) and O(n) respectively. Because both of these methods have performance differences that relate to the implementation chosen by the developer, both are monitored by our profiler.

Other `Collection` methods are not so interesting. For example, the `Collection.size()` method has a complexity of O(1) which is common across all implementations. Since profiling methods like this would offer no benefit to the developer in choosing the best implementation to use, they are not profiled by our tool.

The complete set of methods that we have identified as interesting, and included in our profiler for monitoring, is:

- `Collection.add(<E>)`
- `Collection.remove(<E>)`
- `List.add(x, <E>)`
- `List.get(x)`

- `List.remove(x)`
- `List.set(x, <E>)`
- `ListIterator.add(<E>)`
- `ListIterator.remove()`
- `Set.contains(Object)`

All of these methods have potentially different performance characteristics for each of the standard implementations provided by the Java framework. Furthermore, our profiler is designed in such a way that monitoring extra methods can be achieved with very little modification to the source code. This helps to cover the possibility that there exist other methods of interest not currently monitored by our profiler. It also caters for future developments to the `Collection` interface and other interfaces that inherit `Collection` which may produce more methods of interest.

2.2 Intercepting Method Calls

For our profiler to work it must be able to recognise points in the target program where there are calls to methods of interest. It must then be able to intercept the target program at these points and execute our own profiling code. Object-Oriented programming languages do not provide an easy way for us to achieve this so we must look elsewhere. The solution we chose was to make use of an Aspect-Oriented programming (AOP) language [9]. The specific AOP language we chose was AspectJ.

2.2.1 AspectJ

AspectJ is a popular AOP language designed for Java (see e.g. [7,8] for more on AspectJ). It was chosen very early on in the project as the programming language to implement our tool in. It effectively allows us to automatically place code of our own throughout any Java program, as long as we have its source code. To achieve this interleaving of code using simply Java itself would be an incredibly difficult and tedious task. For example, using AspectJ we are able to intercept a program every time any `Collection` method is called and print out a line of text to the system console. The code to achieve this could take the form:

```
before() : call(* Collection+.*){
      System.out.println(“Collection method called!”);
}
```

Every time a call is made to any method defined in the `Collection` interface, the message will be printed to the system console. In the definition, `call(* Collection+.*)` the ‘+’ indicates that all calls to methods defined in subtypes of the `Collection` interface will also be caught. For example, `ArrayList` is a subtype of `Collection`. Therefore, all calls to methods in the `ArrayList` class will be caught.

To achieve this goal without using an AOP language we would have to search by hand through the entire source code of the target program, inserting print commands at each point where a `Collection` method call is found. We would then have to recompile the program. Once finished with the profiling, we would have to reverse this in an equally tedious fashion.

The way to intercept Java code with AspectJ is to use *pointcuts*. *Pointcuts* effectively define places at which you wish to insert code in the target program. Since our tool is intended to monitor the use of `Collection` objects, we created *pointcuts* for a number of `Collection` methods. Examples of these *pointcuts* are:

```
pointcut colAdd() : call(* Collection+.add(..)) && !within(jcp.*);
pointcut colRemove() : call(* Collection+.remove(..)) && !within(jcp.*);
pointcut listGet() : call(* List+.get(..)) && !within(jcp.*);
pointcut listSet() : call(* List+.set(..)) && !within(jcp.*);
```

Again, we make use of the '+' character to make our *pointcuts* apply to all subtypes of the `Collection` (or `List`) interface. The '`(..)`' indicates that methods with any number and type of arguments will be intercepted. To avoid monitoring method calls from our own code, we used the '`!within(...)`' statement. This statement allows us to exclude our own package from being intercepted by the *pointcut*.

Once we had defined our *pointcuts* we were able to add *advice* to them. *Advice* is code that is run when the *pointcut* is triggered. Advice can include any AspectJ specific code, as well as standard Java code. In the following sections we discuss what the *advice* in our profiler actually does.

2.3 Identifying Objects

When a method call is identified and intercepted by our profiler, there is a great deal of information that is available to us courtesy of AspectJ. This information can provide us with details about the arguments being passed to the method, objects that are being returned by the method, and information about the point in the code at which the interception occurred.

Ideally we want to provide the developer with an analysis of their program which tells them "here is a point in your code that may be using a sub-optimal `Collection` implementation". This is not quite as straight forward a task as it sounds. Most of the interceptions that are done by our profiler are at points in the code where methods are called on `Collection` objects. An example of these interception points follows:

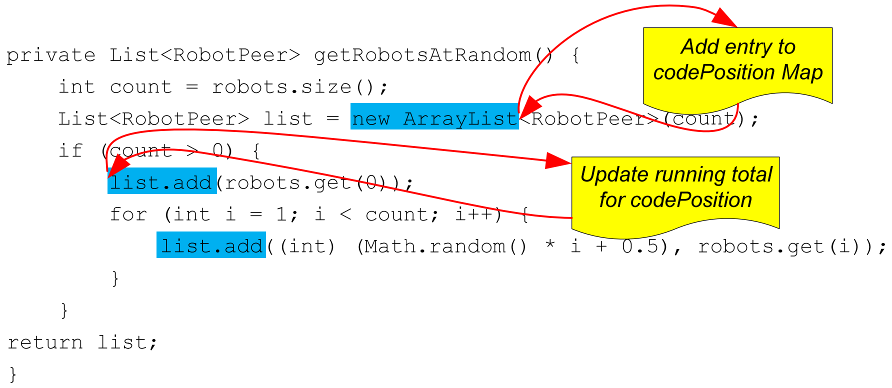
```
...
if (count > 0) {
    list.add(robots.get(0));
    for (int i = 1; i < count; i++) {
        list.add((int) (Math.random() * i + 0.5), robots.get(i));
    }
}
...
```

In the above example, suppose we intercepted the highlighted method calls using AspectJ. Then, we would know the position in the source file where the method call occurred. However, this does not tell us where in the program the object which is being called upon was originally created. The developer really needs to know the location in their code where the `Collection` object was created because this is the point at which the `Collection` implementation is chosen.

In order to provide the developer with this information, we need some way of identifying the creation of `Collection` objects and keeping track of their usage throughout the program execution. Our solution to this uses the following steps for each `Collection` object created in the developer's code:

1. Intercept the program when a `Collection` object is created.
2. Place an entry in a *codePosition* `Map` that ties the unique identifier of the newly created object (effectively its memory address) to the program point at which it was created.
3. Intercept method call of interest.
4. Match the unique identifier of the object to a code position in the *codePosition* `Map`.
5. Update a running total of usage statistics for that code position.
6. Repeat from step 3 until program terminates.

The following diagram illustrates this process in action:



```
private List<RobotPeer> getRobotsAtRandom() {
    int count = robots.size();
    List<RobotPeer> list = new ArrayList<RobotPeer>(count);
    if (count > 0) {
        list.add(robots.get(0));
        for (int i = 1; i < count; i++) {
            list.add((int) (Math.random() * i + 0.5), robots.get(i));
        }
    }
return list;
}
```

Add entry to codePosition Map
Update running total for codePosition

The creation of a `Collection` object adds a link from the object ID to the code position in the *codePosition* `Map`. Later on, when the object is used, the *codePosition* map is referenced so that the running total for the appropriate code position can be updated.

The ‘unique identifier’ referenced above is acquired via the `System.IdentityHashCode(Object)` method that is provided by Java. This method returns an integer that is unique to the object and remains so throughout the entire lifetime of that object. But what happens if an object’s lifetime ends and another object is created with, quite by chance, the same `IdentityHashCode`? Does this distort our statistics? As it turns out, no, it does not. When you consider the previously mentioned sequence of operations you notice that one of the first steps is to place an entry in a `Map` linking the object to the point at which it was created in the code. This means that if an object assumes the identity of an extinct object it must also update the `Map` to point to the new code position at which it was created. Since statistics are paired with code positions, and not object identifiers, there is no risk of contaminating the results with statistics from objects created at another code position.

To provide out profiler with the ability to identify objects and match them up to the position at which they were created in the code we have made use of a couple of `Maps`. Because `Map` implements the `Collection` interface, and our profiler intercepts method calls to objects implementing the `Collection` interface, we can run into problems with self-accounting and, possibly, infinite loops. To avoid this problem we make use of the AspectJ keyword `within`. While defining our *pointcuts* (see section 2.2.1) we insert this keyword along with a negation symbol and our package name to exclude code in our profiler from being intercepted. E.g. `!within(jcp.*)`

2.4 Sampling

Profiling method calls is not a free operation. Our tool incurs overhead in the system when it is run. Because our tool is only intended to be used during the development of software this should not be an issue. However, for our tool to be truly appreciated by developers it should introduce as little cost as possible in the software development process.

There may also be situations where extra overhead must be minimised in order for the program to run correctly. For example, if the program that is to be monitored is highly time dependent then introducing large amounts of extra delay with our profiler might cause the program to function incorrectly or fail completely. For our profiler to be as universal as possible we need to minimise the delay that it might introduce in programs.

To reduce the overhead introduced to the program by our profiler, we used a sampling approach. By sampling a small ratio of the method calls that the program makes, we are able to dramatically reduce the overhead incurred by our profiler. Sampling has been used in many statistical applications to give a relatively accurate view of the population. It is, however, simply a view. If care is not taken, there may be situations where results are not a fair representation of actuality because poor samples have been taken.

Before we developed a technique for sampling in our tool, we needed to decide exactly what we wanted to sample. There were a couple of obvious choices:

- *Object creation:* When `Collection` objects are created, we could sample which ones we took notice of. We would then follow the sampled objects through their entire lifetime and take notice of all calls to methods of interest on them.
- *Method calls:* Take a note of every `Collection` object that is created. Then sample the calls to methods of interest across all `Collection` objects.

Sampling ‘object creation’ would likely not return the same performance benefit as sampling ‘method calls’ in AspectJ. This is because every time a method call was picked up, it would have to have its object ID checked back against the *codePosition* map to determine if it should be sampled or not. We chose to sample ‘method calls’ instead, to gain the greatest benefit in performance.

The benefit of choosing the ‘method calls’ approach is that we are far more likely to gather information on all code positions in the program. If we sample ‘object creation’ then it is entirely possible that some code positions will slip through the profiler without being sampled at all. This is especially likely for positions in the code that only ever create an object once. Simply because they are only created once does not mean they are not heavily used in the program and therefore very important that we monitor.

For our sampling technique we considered a number of approaches:

- *Purely random sampling:* for each member of the population, generate a random number and if that number is over a given threshold include the member in the sample group.
- *Time based sampling:* use a timer that works off the system clock and samples every method call that directly follows the expiration of the timer.
- *Framed sampling:* take the first method call in every frame of *n* method calls as a sample.
- *Frame bounded random sampling:* for every given frame of *n* method calls, select one method call at random to sample.

The technique we ended up using was the final one – frame bounded random sampling. This technique has a number of benefits in our situation:

- *Samples are dispersed throughout the entire program.* With purely random sampling it is quite possible that large portions of the program will go unsampled while others may be oversampled. Framing our random selection helps to dramatically reduce this issue.
- *The sample rate is not defined by program performance.* Time based sampling lead to a different number of samples being taken on machines that differ in performance. The slower the machine, the more samples will be taken. Since more samples leads to poorer system performance, a time based sampling approach may well lead to a snowballing effect in this regard. Since we are basing our selection on the occurrence of method calls, and not an unrelated clock, we avoid this issue.
- *Cycles in the program are unlikely to skew results.* If a plain framed sampling approach is used we face the possibility of inaccurate results in the case of program loops that match our frame size. Assume we are sampling every fiftieth method call. If the program we are monitoring contains a main loop with fifty calls to methods of interest then, with simple framed sampling, we are going to be sampling the exact same method call every time around

the loop. This could lead to a highly distorted result set which would be of little use to developers. By randomising the one method that we select from every frame, the chance of this sort of distortion occurring is highly unlikely.

In Figure 1 we can see a visualisation of how our chosen technique – frame bounded random sampling – works. The program is effectively divided up into even frames of *n* calls to methods of interest. Within each of these frames, a random method call is chosen and sampled. We tested the accuracy of this sampling technique by running it over a number of benchmark programs and comparing the results produced to those of a non-sampled run. We found that the difference between the results of the two was minimal and did not change the implications of the results.

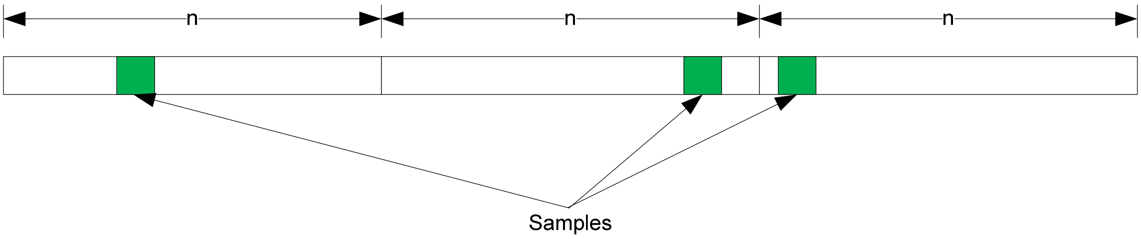


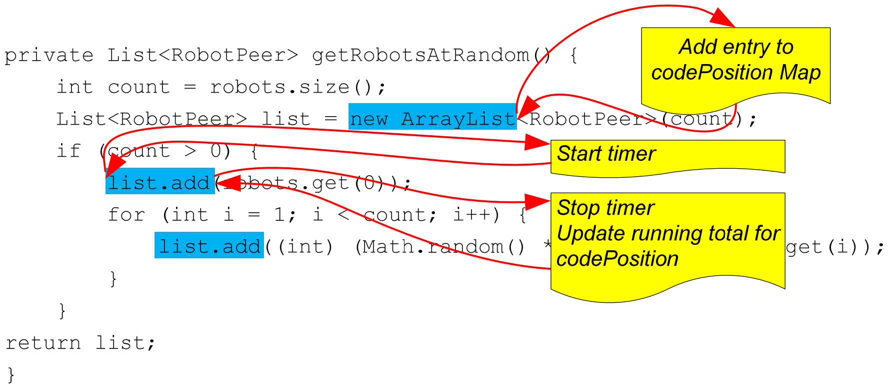
Figure 1 – Frame bounded random sampling

In order to cater for programs of all sizes, and target machines of varying levels of performance, the size of the frame can be easily changed to suit. It can also be reduced to '1' so that every method of interest is sampled.

2.5 Timing

For developers wishing to improve the performance of their programs, simple measures like counting method calls are not overly useful. It would be far more useful to be able to see which methods were consuming the most time for each `Collection` object. For this reason, we designed our profiler to base its statistics gathering on the time spent executing each method. To achieve this measure of timing, we must intercept the program before the method is called and start a timer. We must then leave the program to complete its method, and finally jump back in and stop our timer. Only once we have done all of this can we add the measured time to the appropriate total in our statistics.

If we consider again the example from above we can visualise how this measure of timing might be achieved:



```
private List<RobotPeer> getRobotsAtRandom() {
    int count = robots.size();
    List<RobotPeer> list = new ArrayList<RobotPeer>(count);
    if (count > 0) {
        list.add(robots.get(0));
        for (int i = 1; i < count; i++) {
            list.add((int) (Math.random() *                    get(i));
        }
    }
return list;
}
```

Add entry to codePosition Map
Start timer
Stop timer
Update running total for codePosition

Using the *around advice* in AspectJ, we are able to place our own code on either side of an intercepted method call. Before the method proceeds we take the current time from the `System.nanoTime()` method that is provided by Java and store it in a variable. This method provides us with the current system time at a high resolution. Since some methods execute very quickly, using a millisecond degree of accuracy would not be sufficient in all cases. We then allow the method to execute by using the `proceed()` call. Finally, once the method has completed, we can once again make use of `System.nanoTime()` to get the finish time of the method and calculate the time spent executing it. The measured time is then added to our statistics. All of this happens within the *around advice* in our profiler code.

One issue that must be considered when using this approach is the issue of multithreaded programs. A method call may be intercepted and then lose its time slice just after our timer starts. Since our timer relies on the system time for its measurement, every unit of time spent waiting for the thread to re-enter execution will be included in our measurement. This is, potentially, quite a large problem and may need to be addressed in future work. In our testing we included multithreaded software and did not observe any obvious inaccuracies, but acknowledge the fact that they may well exist.

Chapter 3
Experimental Results

After implementing our profiler it was necessary to gain a real world view of how well it might achieve its goal – to help developers make the best decisions with `Collection` implementations. The best way to do this was to find some real world programs and run our profiler over them. We did this, and then analysed the results to assess if `Collection` implementations had been chosen wisely by the developer(s). If we found decisions that our profiler suggested might be poor, we recompiled the program using the better implementation and measured any performance benefits.

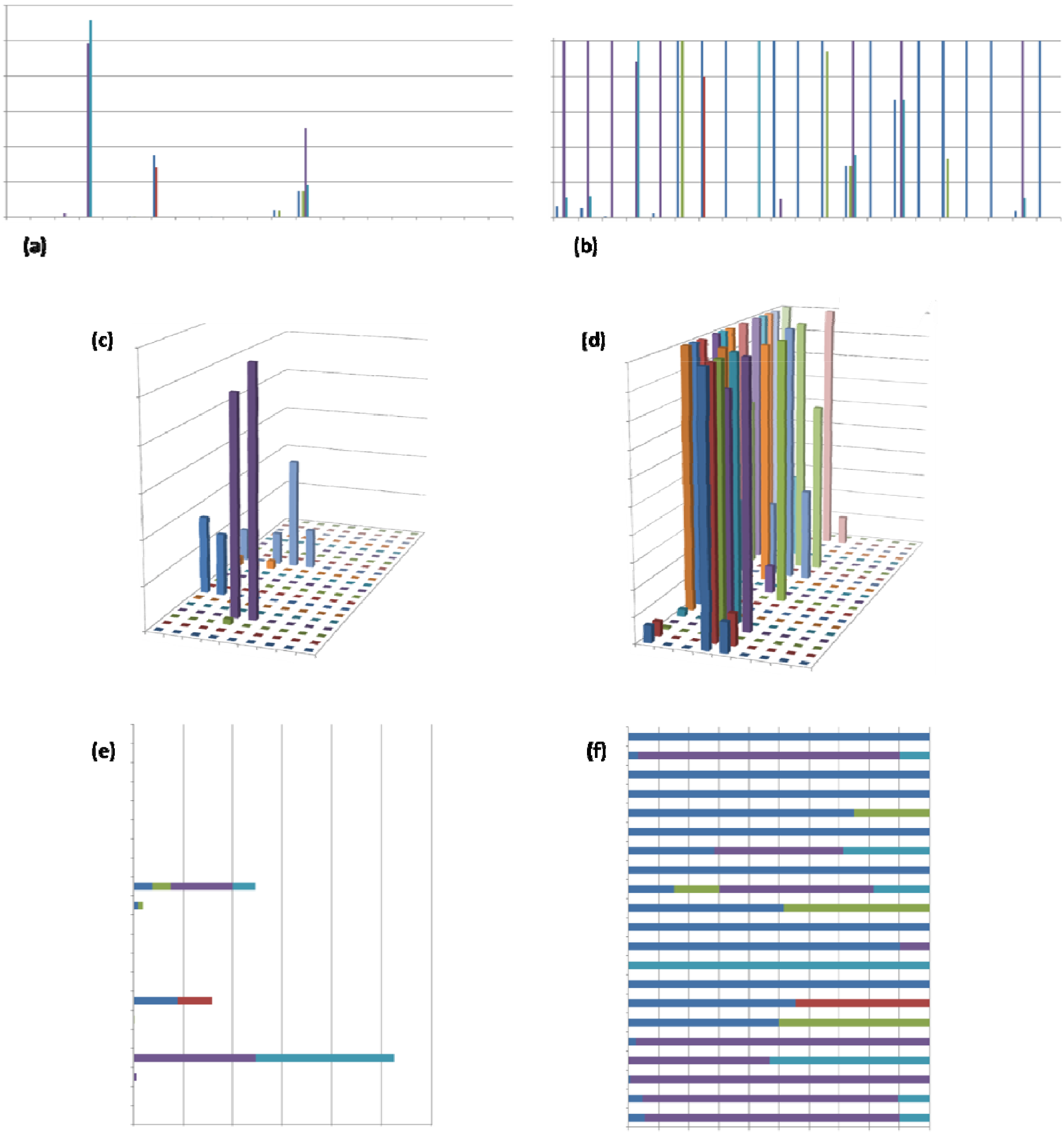
3.1 System

For the vast majority of testing and benchmarking we used just one system configuration. We ran tests of our tool on other system configurations to ensure portability but the results presented are all from a common system. That system is:

Make	Toshiba
Model	Portege R100 (PPR10A-04M8ZP)
CPU	Intel Pentium M (Banias), 1GHz, 400MHz FSB, 1MB L2 Cache
Chipset	Intel i855GM
Memory	1280MB, 266MHz
HDD	Toshiba MK6006GAH, 60GB, 4200RPM, 2MB Cache
OS	Microsoft Windows XP Professional, Service Pack 2
JRE	1.6.0_07-b06
AspectJ	1.5.1

3.2 Displaying Results

Having developed a tool that could gather usage statistics for the use of `Collection` objects we were faced with another challenge – how to present the results. Most of the benchmarks that we used to test our profiler produced amounts of data that were just not practical to try to analyse as raw numbers. For data to be useful it should be in a form that is easy to interpret and analyse, and in many cases the best way to achieve this is through visual representation. A chart of some description seemed like the best candidate for visual representation in our case. We considered a number of styles of chart, examples of which are presented below:



From the above diagram, the chart types are as follows:

a) Columns
b) Normalised Columns
c) 3D Columns
d) Normalised 3D Columns
e) Stacked Bars
f) Normalised Stacked Bars

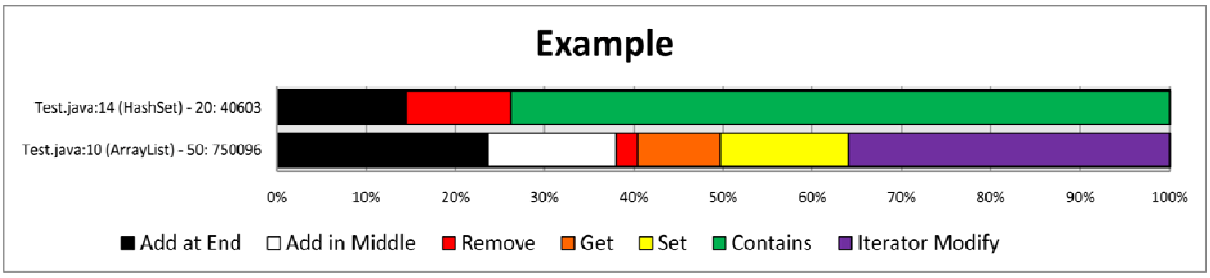
We identified problems with each of the above chart types. (a), (c), and (e) all dwarf lesser used objects into practical non-existence. It may be argued that the performance characteristics of lesser used objects are not very important because they have little effect on the program as a whole. This may be true, but we would like our tool to benefit all areas of developers' software, not just those that will have the greatest effect.

To ensure that all of the results are visible to the developer we investigated normalising the charts as shown in (b), (d), and (f). This meant that for each program point that produced data in our results we could see the data stretched along the full length of the chart. We found that the simplest of the normalised charts to interpret was (f), the Normalised Stacked Bars style. This style uses a standard width bar for each program point, and splits that bar up into method types by colour. This allows the developer to quickly and easily determine which methods each `Collection` object spends most of its time executing.

We would later add program point labels to the Y axis, and a legend to map colours to method descriptions. A further enhancement was to add the total time spent in all measured methods to the label and sort the results according to this value. This value aids the developer in estimating the expected improvements in performance associated with using the best `Collection` implementation at each code position. It effectively gives all of the fixed width bars a ‘size’ relative to each other.

3.3 Charts

The type of chart we use is a Normalised Stacked Bar Chart and is described briefly in Section 3.2. For an understanding of how we use this chart type, consider the following example:



Each bar is representative of the performance characteristics of `Collection` objects created at a specific position in the code. That code position is displayed at the beginning of the label on the Y axis. From the example: ‘`Test.java:14`’ identifies the code position in the file ‘Test.java’ at line number 14. Following this in the label is the type of object being created at that position in the code. Using the same example, ‘`(HashSet)`’ identifies that a new `HashSet` object is being created. The final part of the Y axis label related to our statistics. The first number (in our example ‘20’) is representative of the number of method calls that were sampled from objects created at the given code position. The last number (in our example ‘40603’) is the total time spent in the execution of these sampled methods, as measured by our profiler.

As mentioned earlier, providing information about the total time spent executing methods for each code position can be very helpful. It allows developers to estimate how much a better implementation selection might benefit the performance of their program. The count is also important as it gives a measure of how often methods from each code position are being used in the program.

Within each bar the colours represent the different methods that have been measured by our profiler. These bars have been ‘normalised’ to each span the entire width of the chart. Because of this, the size of each ‘chunk’ of each bar does not represent an absolute measurement. Instead it represents a relative measurement of time spent performing the associated method as compared to time spent executing other methods on objects created at the same code position. For example, if a bar is displayed with a half in one colour and a half in another, then it has spent an equal amount of time executing each associated method. Comparisons can only be made within the same bar because other bars may represent a different overall execution time.

Brief descriptions of the methods are given in the legend at the bottom of the chart. A more substantial description is given below:

Legend Label	**Included Methods**
Add at End	`Collection+.add(<E>)`
Add in Middle	`List+.add(x, <E>)`
Remove	`Collection+.remove(<E>)`, `List+.remove(x)`
Get	`List+.get(x)`
Set	`List+.set(x)`
Contains	`Set+.contains(Object)`
Iterator Modify	`ListIterator+.add(<E>)`, `ListIterator+.remove()`

3.4 RoboCode

RoboCode is an open source, educational Java game in which user-created simulated robotic tanks fight each other in a 2D space. The purpose of the game is to help teach people how to program in Java and develops ideas of Artificial Intelligence [1,2,3,4]. All of the tanks in the game are 'physically' identical but differ in the way developers program them. The tanks can shoot at each other, move around, scan for each other and bump into walls and other tanks. The program is designed in such a way that it is not difficult at all to program a robot but programming a really good one is quite a challenge. This provides a learning curve for new Java developers that allows their robots to get better as their programming knowledge and technique improves.

3.4.1 Procedure

The version of RoboCode used throughout the development of our profiler was 1.6.0 Beta 2. Although a beta release, we found it to perform perfectly with no stability issues at all.

We ran our profiler over the program a number of times, using different game parameters each time. The different combinations we used are described in the following table:

Parameter	Configuration 1	Configuration 2	Configuration 3	Configuration 4
Robots	5	10	5	5
Rounds	10	10	20	10
Playing Field	800x600	800x600	800x600	1600x1200

We used Configuration 1 as a 'base' configuration. For Configurations 2 – 4 we changed a single parameter from that of the base configuration. The reason for this was that we wanted to see how the use of `Collection` objects changed within the program when the input variables changed.

3.4.2 Data

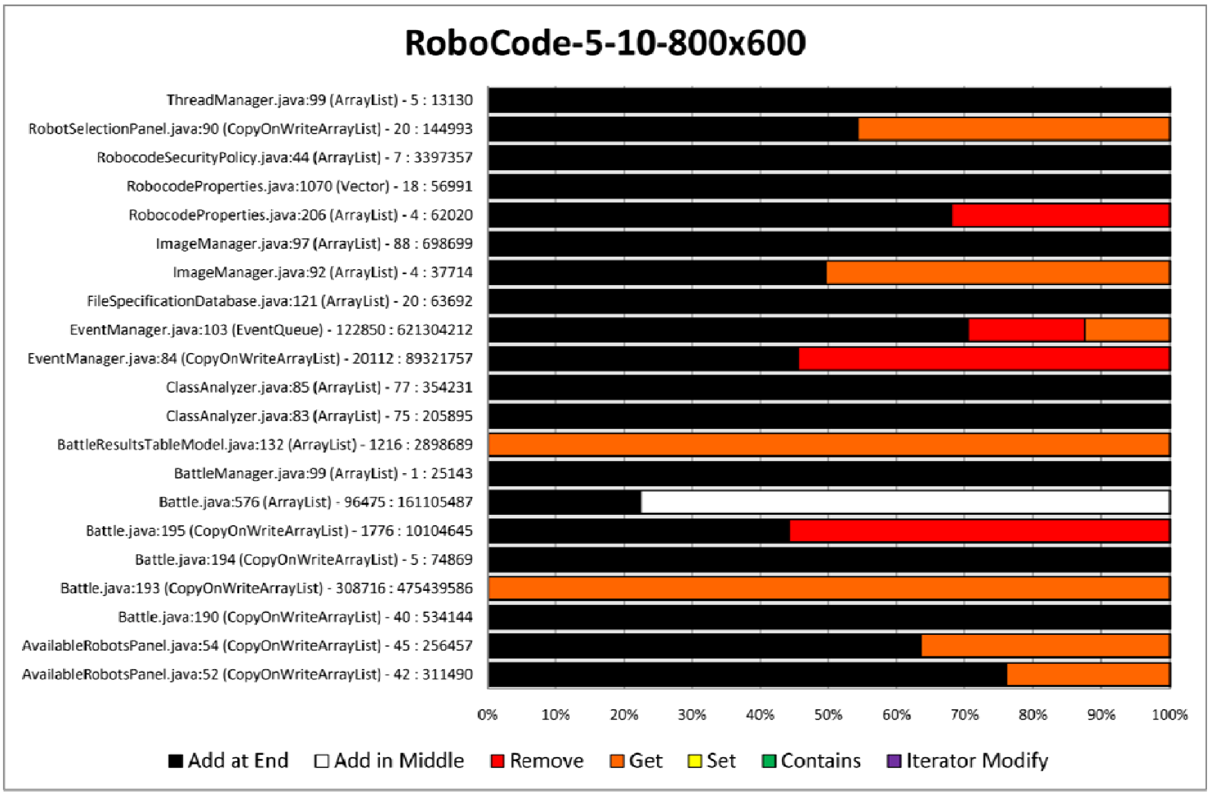


Figure 2 - RoboCode results - Configuration 1

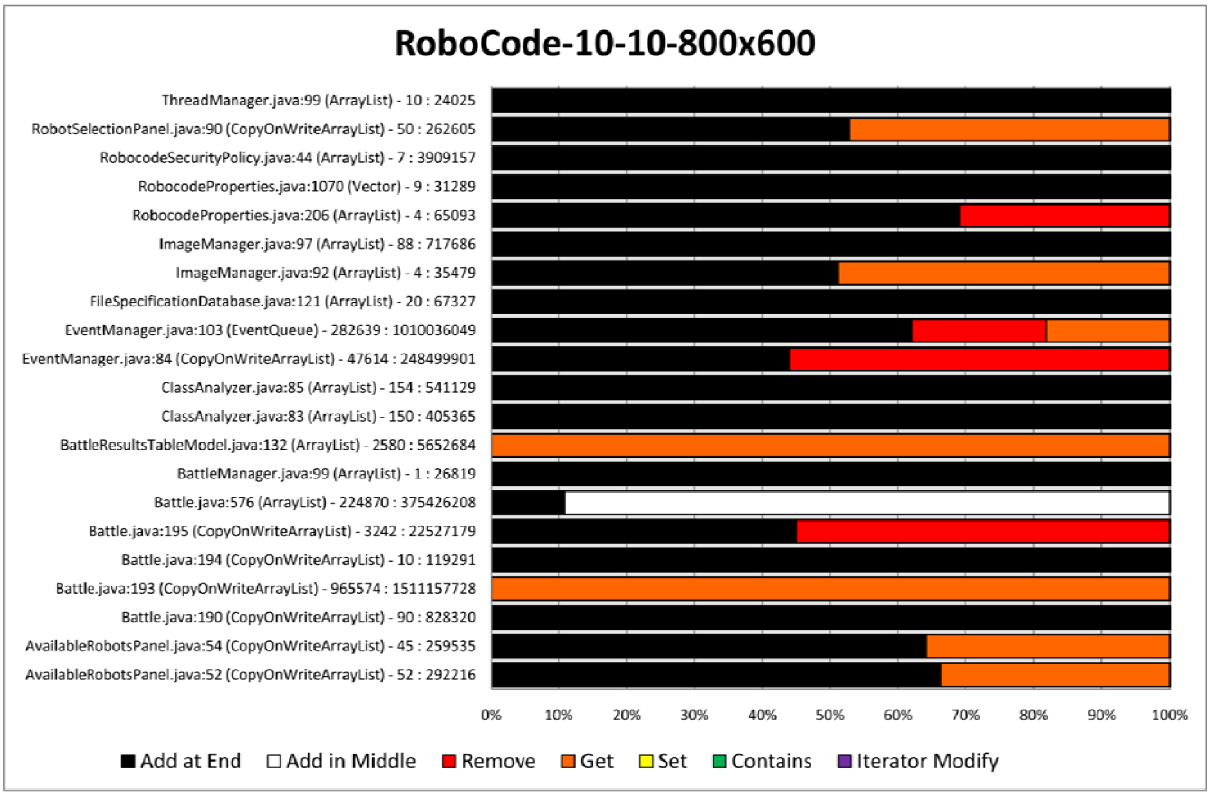


Figure 3 - RoboCode results - Configuration 2

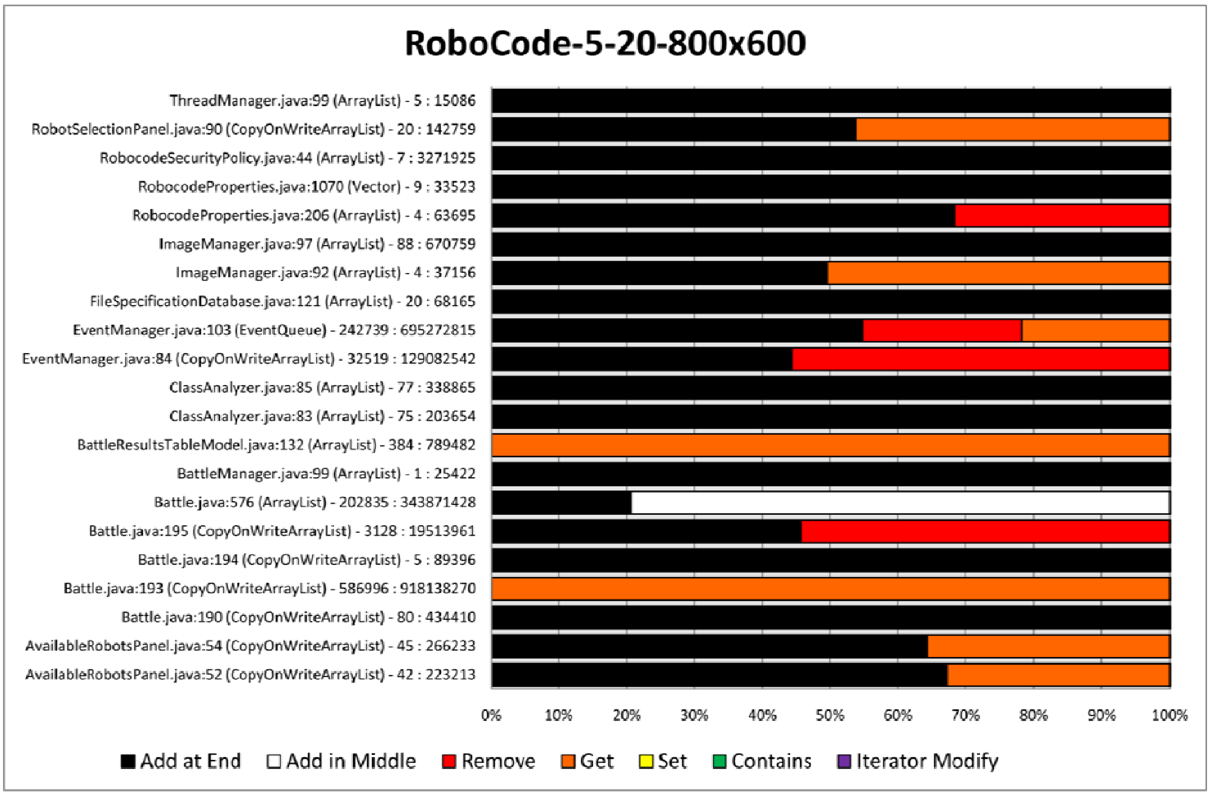


Figure 4 - RoboCode results - Configuration 3

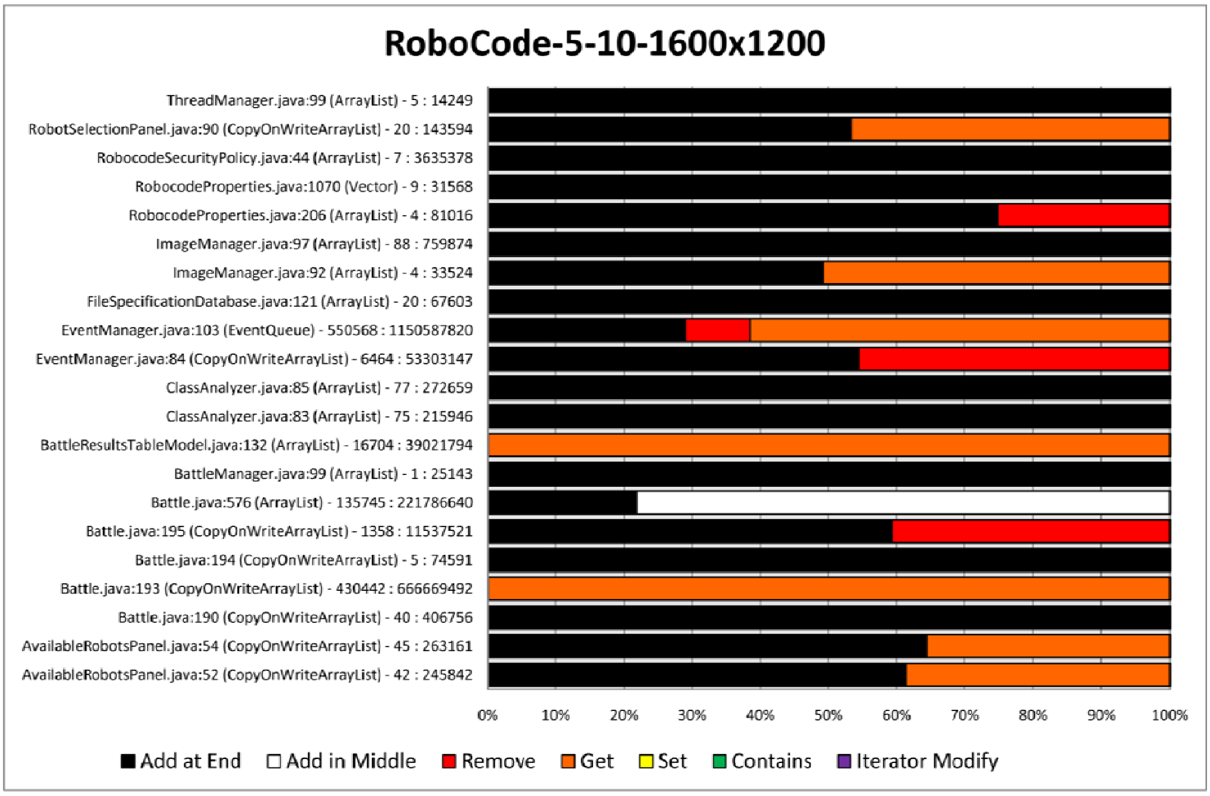


Figure 5 - RoboCode results - Configuration 4

3.4.3 Discussion

The results from RoboCode show heavy use of the `ArrayList` `List` implementation throughout the program (see Figure 2 through Figure 5). This implementation is favoured for operations like `List.get(x)` and `List.set(x,<E>)`. We can see from the results that all four configurations of RoboCode make use of the `List.get(x)` method quite extensively at a number of code points in the program. In these cases, where `List.get(x)` and sometimes `List.add(<E>)` are used, `ArrayList` is the best choice of `List` implementation. This is because it has complexity of O(1) for both of these operations.

If, however, we look at the code position ‘Battle.java:576’, we can see that the overwhelming majority of the execution time is spent in the `List.add(x,<E>)` method. As discussed earlier in this report, this method can yield performance benefits when implemented with a `LinkedList`. This assessment is not conclusive, as the performance characteristics depend on how the program is using the method. Only in certain situations will a `LinkedList` benefit program performance. Nevertheless, this is a point in the program that should be examined because it shows potential for improvement.

We examined the code in the ‘Battle.java’ file of RoboCode at and around line 576 and found the following:

```
private List<RobotPeer> getRobotsAtRandom() {
      int count = robots.size();
      List<RobotPeer> list = new ArrayList<RobotPeer>(count);
      if (count > 0) {
            list.add(robots.get(0));
            for (int i = 1; i < count; i++) {
                  list.add((int) (Math.random() * i + 0.5), robots.get(i));
            }
      }
      return list;
}
```

The `List.add(x,<E>)` method is being called inside a loop which explains its large presence in the results. When we look at where exactly the items are being added to the list (the ‘x’ value) we see that a random number method is being used to determine position. In order to gain performance benefits from a `LinkedList` we either need to have the items being added consistently at the beginning of the `List`, or be in a situation where the list is being iterated over anyway. In this case neither applies. Since the position at which the new item is being added is being determined at random, we cannot expect to gain any benefits in performance from a `LinkedList` implementation.

However, if we look beyond what the code *says* and assess what is actually *achieves* we can see room for improvement. The point of the code is to create a randomised copy of the ‘robots’ `List`. As the code stands, it is an expensive operation. For every item in the ‘robots’ `List`, a `List.add(x,<E>)` operation is being performed on the new `List`. This results in a complexity of O(n^2). Instead, we could start by performing an initial copy of the ‘robots’ `List` which has complexity O(n). We could then use the `Collections.shuffle()` method to randomise this new `List`. `Collections.shuffle()` is based on an algorithm developed by Donalth Knuth [10] and also has complexity of O(n). Using this approach means that we are able to reduce an O(n^2) complexity operation into a much more efficient O(n) complexity operation.

It is important to note that this performance improvement is not related to implementation choice but rather algorithm efficiency. However, it was because of our profiler that this code point came to our attention. This demonstrates the flexibility of our tool in being able to help developers to improve efficiency in their code.

Observant readers will note that the list that is created in the above example is returned at method completion. This means that there may be code elsewhere that performs `List.add(x,<E>)` operations on the object. Such code could potentially lend itself towards a `LinkedList` implementation. This is a valid argument, but upon searching for calls to the `getRobotsAtRandom()` method within the program we found no such code.

While this code position turned out to be using the most appropriate `List` implementation, it shows up another point of interest in our results. This is that the configuration of a program can indeed have an effect on how the program uses its `Collection` objects. If we consider the same code point across all four figures, we see that Figure 3 has a noticeable difference in its ratio of calls to `List.add(<E>)` and `List.add(x,<E>)`. While not being applicable in this case, it is important to note that a specific implementation may not be the best choice for all configurations of the program. This point will be revisited later in Section 5.1.

Another code position that illustrates the effect that program runtime configuration can have on its use of `Collection` objects is in the file 'EventManager.java' at line 103. Objects created at this code position exhibit similar patterns of use when run with Configurations 1 through 3 (see Figure 2 through Figure 4). That pattern is a split of approximately 70% (Add at End), 15% (Remove), 15% (Get). When we look at the results from Configuration 4 (Figure 5) this pattern shifts dramatically. In this configuration we see a split that is close to 30% (Add at End), 10% (Remove), 60% (Get).

It is clear from the results that the two code positions described above have patterns of use that vary with the runtime configuration. It is interesting to note that individual parts of the configuration can be linked to usage patterns at specific positions in the code. Consideration should therefore be taken to profile the program with as many different configurations as possible. This will help to achieve the most accurate view of the use of `Collection` objects in the program.

3.5 JGraph

JGraph is a freely available, open source graph component library for Java [11]. It integrates into the popular Swing framework [12] to provide developers with easy-to-use visual graphing components for their software.

3.5.1 Procedure

Since JGraph is a component library, not an executable program, it cannot be profiled. There are, however, a number of example programs that are packaged with the distribution we obtained (5.9.2.1) that integrate components from the library. One of the example programs is called FastGraph. This program has the ability to produce fully connected graphs of any size. For profiling, we used this program to generate and display a fully connected k64 graph.

3.5.2 Results

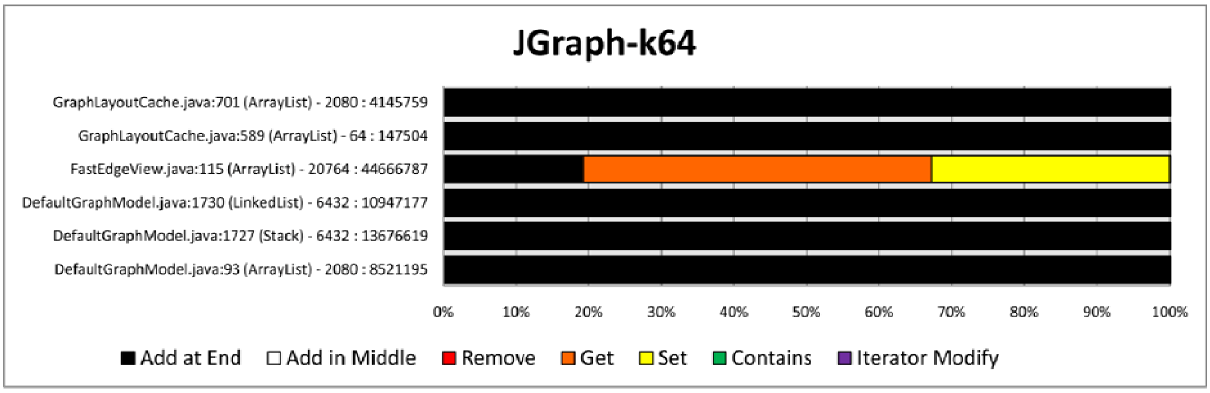


Figure 6 - JGraph (k64) results

3.5.3 Discussion

From Figure 6 we can see that `Collection` objects are not widely used throughout the FastGraph program. Just six creation code points are monitored and all but one of these are observed to only add items to the `Collection`. It would seem peculiar that the only method calls on a `Collection` object would be to add items. After all, if the program is not going to do anything with the items once they have been added, why bother adding them at all? In actuality, there are almost certainly other method calls being performed on the `Collection` objects being created at these code positions. The reason our profiler isn't picking them up is that they are likely methods that we have determined to be non-interesting (as mentioned in Section 2.1).

The one code position that does show some variation in its use is in the file 'FastEdgeView.java' at line 115. Objects created at this point are observed to spend a considerable amount of time performing `List.get(x)` and `List.set(x)` methods. These methods will tend to perform best under an `ArrayList` implementation. As we can see, the developer has used an `ArrayList`, so in this case the best decision has been made.

3.6 JavaC

JavaC is a popular Java compiler that is included in the Java Development Kit (JDK) from Sun Microsystems [13]. As an input it takes Java source code, and produces Java bytecode as an output. JavaC uses a number of `Collection` objects and has no user interaction throughout its execution which makes it an ideal benchmark for our profiler.

3.6.1 Procedure

The version of JavaC that we used was from version 1.6 of Sun Microsystems' JDK. As input for the program we used the source code of the program itself. We were therefore profiling JavaC compiling itself. This included a number of Java source code files, which we felt were sufficient to provide a broad execution path for the program to be profiled on.

3.6.2 Results

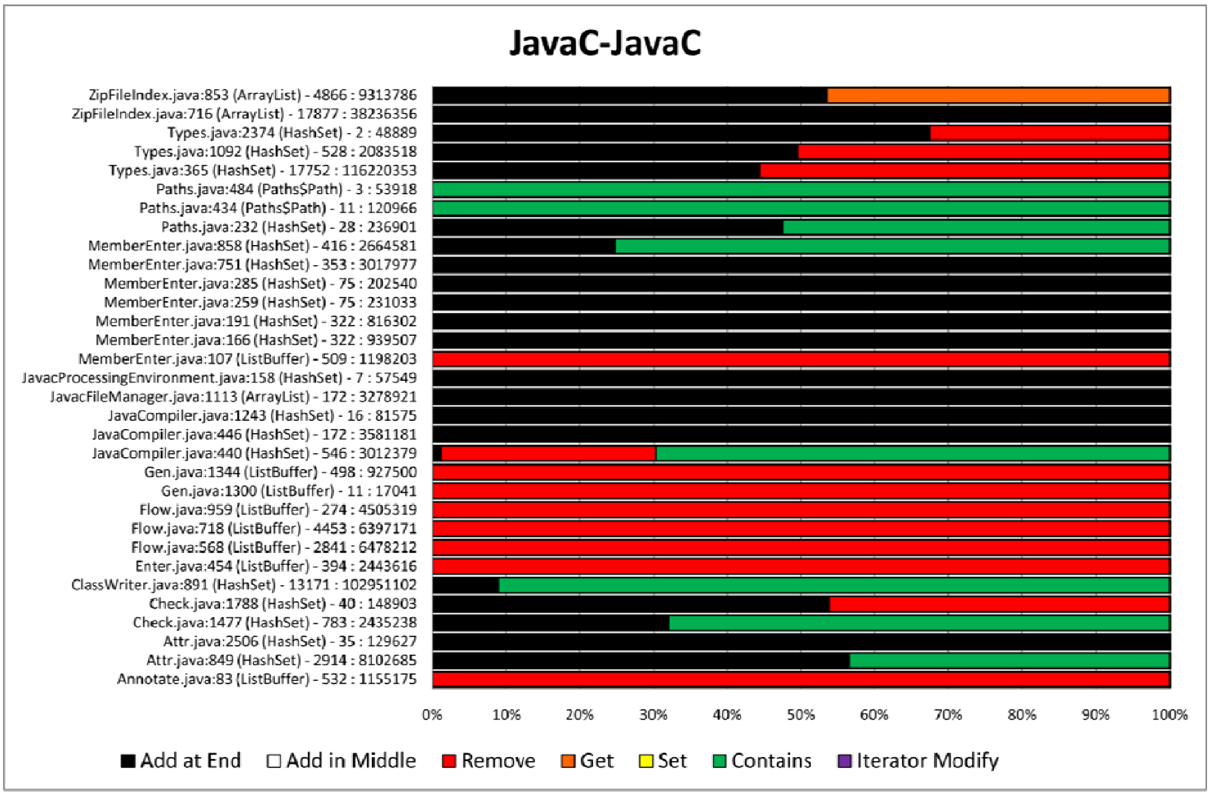


Figure 7 - JavaC self-compilation results

3.6.3 Discussion

The results shown in Figure 7 illustrate the extent to which `Collection` objects are used within JavaC. `HashSet` is the most common standard `Collection` implementation used, but there are also a few `ArrayList` implementations shown. The performance characteristics for these standard implementations seem to match their use well. `HashSet` has complexity of O(1) for its measured operations (`add`, `remove`, and `contains`), as does `ArrayList` for its measured operations (`add` and `get`).

Interestingly, another prevalent implementation in the results is one that is not found in the standard Java libraries. This custom `Collection` implementation (`ListBuffer`) seems to be used at a number of places in the program and spends a great deal of its time performing the `remove` method. Due to the naming of the class, we would expect it to be related in some way to the `List` interface. This may not be the case though, and to understand exactly how it is used we must investigate the class. The original developer would, presumably, already have this knowledge.

After investigating the code and comments contained in the `ListBuffer` class, it became apparent that the class is indeed closely related to the `List` interface. From the comments in the code, the class is described as “A class for constructing lists by appending elements”. Because of its `List`-like use, we should expect that there is potential for a `LinkedList` style of implementation to perform better than an ArrayList one in certain circumstances.

We investigated `remove` calls on `ListBuffer` objects throughout the program and found that this method was only ever called within the `ListBuffer` class itself. Furthermore, it is only ever called to remove the very first item in the underlying `List`. This is an ideal situation in which a `LinkedList` can perform well, as the complexity for the `remove` method is just O(1). As it happens, a `LinkedList` style underlying `List` implementation has indeed been used, so performance is optimal.

3.7 Hospital

Hospital is a simple program used in a first year programming paper at Victoria University of Wellington. It models a scenario of patients queuing to be admitted in the emergency ward of a hospital. It was used in the Computer Science paper COMP103 in the second trimester of 2005 at Victoria University of Wellington. There are a number of reasons properties of the program that made it attractive to us as a benchmark:

- It was not written to be an example for this project.
- It was not intentionally written to be inefficient.
- It has timing mechanisms built in which provide a means of measuring any performance gains (or losses) we might achieve by selecting different `Collection` implementations.

Hospital is not a very complex program, and does not make use of many Collection objects, but as we shall see it makes for some very interesting analysis.

3.7.1 Procedure

Hospital takes two input parameters that must be defined by the user at runtime. Those parameters are 'Number of Patients' and 'Number of Priorities'. The results we present for the profiling of the program are measured from runtime parameters of 10,000 and 50 respectively. We only present one chart in the results for Hospital because in this case, changing the parameters did not seem to have any effect on the ratios of time spent in each method.

Once we had completed profiling, we ran the program a number of times with a range of parameters. This was for the benefit of measuring performance improvements. We were able to extract the timing measurements from the inbuilt timing mechanism and use them for comparison between implementations.

3.7.2 Results

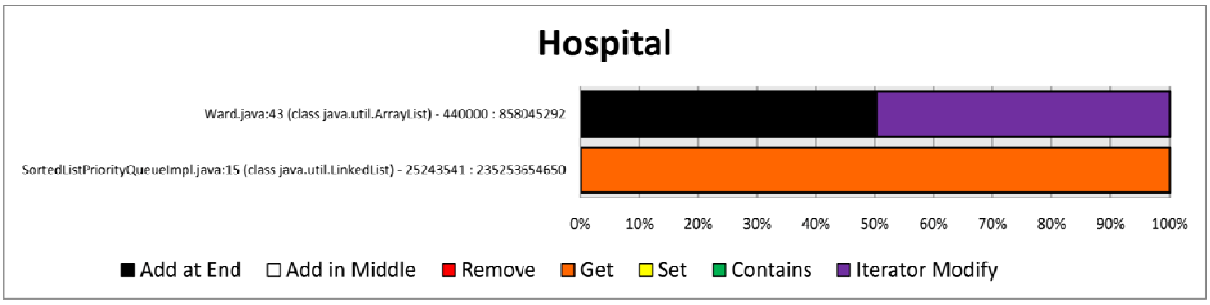


Figure 8 - Hospital results

3.7.3 Discussion

Although somewhat brief, the results that we gained from testing the Hospital program (see Figure 8) make for some interesting analysis. Both of the points in the program that created `Collection` objects that were later monitored seem to show usage characteristics that are not suited to their implementation.

The first code point shows a `LinkedList` implementation with characteristics that appear to better suit an `ArrayList` – the overwhelming majority of method calls are to `size()` and `get(x)`. The size() method has O(1) complexity for both LinkedList and ArrayList. The get(x) method, however, has O(n) for LinkedList and O(1) for ArrayList. We would expect to see a benefit from changing to an ArrayList implementation in this case. As it happens, the LinkedList at this point of Hospital is actually implementing the Queue interface, not the List interface. Since ArrayList does not implement the Queue interface, it cannot be used in place of a LinkedList.

The second code point in our results also shows an interesting mix of characteristics and implementation choice. An ArrayList had been chosen when the only point of significance is a large number of calls to the ListIterator methods we are monitoring. These methods lend themselves to the LinkedList implementation by turning operations like add and remove into O(1) complexity operations. They remain O(n) complexity in the case of an ArrayList implementation. It seems pretty clear cut in this case that a LinkedList should have been used instead of an ArrayList. To validate this claim, however, some real world testing is required.

To test the prediction that changing the ArrayList creation in Hospital to use a LinkedList implementation will benefit performance, we ran Hospital a number of times with a number of different configuration variables. We then ran the same simulations with the source code modified to use a LinkedList, and the program recompiled. As mentioned earlier, Hospital already includes mechanisms for timing executions of the simulation and these were used to produce a measure of performance that we could analyse.

Our results were conclusive. In all of the configurations that we tested, the build of Hospital that used a LinkedList performed at least as well as the original ArrayList build. Most cases showed a significant performance advantage to the LinkedList build. Some cases showed as much a 40% increase in performance with the LinkedList build. These results are very pleasing as they demonstrate the potential for our tool to be incredibly beneficial to the world of software development.

3.8 SimpleLISP

LISP is one of the oldest high-level programming languages in use today. SimpleLISP is an interpreter for this language that has been developed in Java by Dr. David J. Pearce. It provides an editor in which LISP programs can be created, edited, loaded and saved, as well as the interpreter which executes the code.

3.8.1 Procedure

The SimpleLISP distribution comes bundled with a few example LISP programs. We made use of the 'Fibonacci' example to run through the interpreter and profile the program with. 'Fibonacci' is a basic LISP program that takes one user defined integer input, and calculates a Fibonacci sequence with length equal to the user input. We kept the sequence length relatively low (< 40) in order to keep execution time reasonable, and implemented a simple timing mechanism for measuring performance using the System.currentTimeMillis() method.

3.8.2 Results

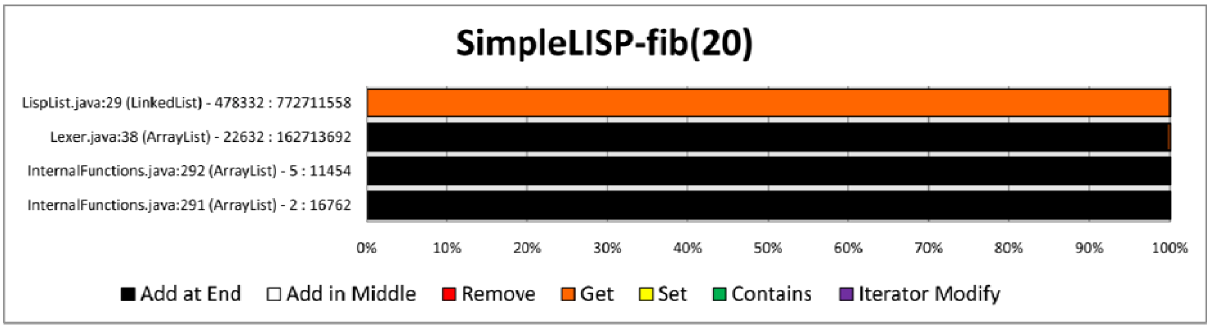


Figure 9 - SimpleLISP (Fibonacci sequence) results

3.8.3 Discussion

Our results for SimpleLISP, displayed in Figure 9, show only four positions in the code at which Collection objects that we monitor are created. These results indicate that there is only one position in the code at which we might be able to improve performance through implementation selection. This position is in the file 'LispList.java' at line 29. At this point in the code, a LinkedList is being used to implement the List interface, but the results indicate that an ArrayList might be better suited to the task. This is because the List has been measured to spend a great deal of its time in the List.get(x) method, which has complexity of O(1) for ArrayList and O(n) for LinkedList.

To test whether an ArrayList would improve performance in this case, we implemented a simple timer around the execution of the LISP program using the System.timeInMillis() method in Java. This method offers millisecond resolution, which was adequate for our purposes given that the shortest execution time was over 300ms. We then ran the 'Fibonacci' LISP program in the interpreter with a number of different input values and recorded the measured time spent in execution. We then recompiled SimpleLISP with the original LinkedList implementation and repeated the same executions of 'Fibonacci'.

What we found was a little surprising. While there was a measured performance improvement using the ArrayList implementation, this improvement was only about 2%. This percentage remained relatively constant across executions with all input values. There are a number of reasons why the measured performance improvement might have stayed quite low:

- Access to Collection objects at this point might account for only a very small amount of the execution of the entire program.
- The position in the List that is being accessed might consistently be at the beginning of the List.
- The List might only ever contain a very small number of items.

Nevertheless, a certain performance improvement was achieved.

Chapter 4
Related Work

Little work is known to us in the specific area of profiling Collection objects to benefit implementation selection. There has been, however, a reasonable amount of work done in the general area of profiling. Some of these works are described briefly in this chapter.

4.1 DJProf

DJProf is a tool that is designed to profile four elements of Java programs [5]. These elements are:

- Heap usage
- Object lifetime
- Wasted time
- Time-spent

At the time of its creation, there were a number of other tools that could profile these same elements in Java programs and were readily available. The reason that the tool was developed was not simply to add to the collection, but rather to explore whether or not it would be possible to efficiently achieve this task using an aspect oriented programming language (AspectJ).

The development and testing of DJProf sought to answer questions at two levels. The base question was whether or not AspectJ provided sufficient functionality to carry out profiling tasks for the Java program elements mentioned above. The second level of investigation was whether or not this could be achieved efficiently. As the paper shows, profiling the above program elements is indeed possible and in most cases reasonably efficient at doing so. It does, however, highlight some limitations with AspectJ as a language for profiling.

DJProf was very useful as a proof-of-concept for our project and provided a solid foundation to build upon.

4.2 Absolute Timing vs. Sampling Time Profiling

To determine the ratio of time that an object spends executing one method as compared to another, our tool uses an absolute timing approach. This sort of approach involves measuring the entire time from entering a method to method completion. Once all measurements have been made, the totals for each method are compared to determine the ratio between them.

An alternative approach to time profiling is known as time sampling. Rather than measuring the absolute time spent in each method, it periodically records the currently executing method. This builds up counts for each method type which are then compared to determine the ratio of time spent in each. There benefit of a time sampling approach is that it has a much lower overhead than absolute timing. The downside is that it is not nearly as accurate as absolute timing.

Further discussion of the two timing approaches can be found in [5] and [14].

4.3 Profiling Allocation Behaviour

Other profiling work has been done in the area of performance costs associated with the garbage collection (GC) of unreferenced objects in a running program [6]. Many popular programming languages do not support the notion of GC. C and C++ are examples of such languages. Java, however, does include automatic GC.

There have been arguments made for and against the use of automatic GCs in programming languages. The main benefit of GC is that is helps to reduce memory leaks in programs that can cause poor performance and sometimes complete failure of the program. Unfortunately, like any piece of code, automatic GC has a performance cost associated with it.

The work detailed in [6] focuses on tuning GC in Java to optimise its performance for a given Java program. In order to optimise GC for the program, the program itself must first be analysed to see how its memory is allocated and used. In this case, profiling is achieved by taking a trace from the program and running it through a simulation environment where measurements are made.

This sort of approach seems to work for its intended purpose which involves assessing memory use. It is not, however, a viable solution for our purpose as we are concerned with live timing information. It would seem optimistic at best to assume that any simulation would accurately reflect the timing characteristics of the live execution of the program.

Chapter 5
Conclusion

Every day, developers find themselves faced with decisions about which Collection implementations they should use to represent their data structures. These decisions have the potential to drastically alter the performance characteristics of the programs in which they are used. We hypothesised that developers do not always make the best decisions with Collection implementations and envisioned that a tool that could help make the best decisions would be very valuable. Having developed such a tool, we proceeded to test its usefulness. We found out tool to provide useful information for Collection implementation decision making and proved our hypothesis to be true by analysing a number of real world benchmarks.

5.1 Future Work

This project has illustrated the value that lies in making the best decisions when selecting Collection implementations in software development. There are a number of directions that work could continue in this area of research. Some examples of these are briefly discussed in the following sections.

5.1.1 Hybrid Implementations

Our Java Collection Profiler has shown to illustrate the ways in which Collection objects are used quite effectively. It has been shown to be very useful in making the best decisions when selecting specific Collection implementations. The value of the tool doesn’t stop there though. Because results are displayed in a form which shows all interesting aspects of the use of Collection objects, they are able to highlight situations where no available Collection implementation is perfect for the task at hand. It could prove to be useful in realising potential for completely new Collection implementations. Such implementations may be tailored to provide optimal performance for situations where the currently available implementations are less than ideal.

Another direction of work in this area would be to investigate the viability of self-managing Collection implementations. These could be designed in such a way that they are able to monitor their own use and adjust their underlying implementation to best suit their operating environment. This would likely be an interesting area of study as achieving such an implementation would remove the need for the developer to make any decision at all. Careful consideration would need to be given to the overhead associated with the self-management and whether or not this overhead negates the performance improvements.

5.1.2 Profiling Other Collection Types

Our main focus throughout the project has been in the List interface. This is just one of the subtypes of the Collection interface in the Java language. Other subtypes include:

- BlockingQueue
- Queue
- Set
- SortedSet

There are also interfaces such as Map in the Java language that do not implement the Collection interface but do represent data structures.

Each of these types has its own unique methods and implementations. If work is put into profiling these unique methods, and comparing the results with the performance characteristics of each implementation, further benefits to developers may be achievable.

5.1.3 Alternatives to Aspect Oriented Approach

Earlier in this paper we discussed the benefits in using an AOP language for profiling over modifying source code by hand. Using an AOP language is not the only way to avoid this tedious task though. Another way might be to implement profiling wrapper classes for the existing `Collection` implementations and then change the class loader to a customised version. This custom class loader could effectively replace all implementations of one sort with the profiling wrapper associated with it. No source code modification would be necessary in the target program and these wrapper classes would appear transparent it.

There are pros and cons to a solution like this. One of the benefits is that code position lookups and object identity methods would not be necessary as the wrapper class could hold local variables to store the timing information for method calls. Sampling could also be easily achieved by building it in to the class loader so that not every `Collection` object is replaced with one that is encased in the associated customised wrapper class.

Perhaps the biggest problem with taking an approach like this to profiling is that any program that relies on an object *being* an instance of a specific class, rather than simply *behaving* like one, will break. An example of when this might occur is in code that makes us of the `instanceof` statement. This boolean statement will only return true if the object that it is assessing is an instance of the exact class that is specified. Therefore any sort of class substitution, no matter how similar it is to the original, will cause the statement to return false every time.

Chapter 6
References

1. Ken Hartness. Robocode: using games to teach artificial intelligence. *J. Comput. Small Coll.*, 19(4):287–291, 2004.
2. Jin-Hyuk Hong and Sung-Bae Cho. Evolution of emergent behaviors for shooting game characters in robocode. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 634–638. IEEE Press, 2004.
3. Mathew Nelson. Robocode, http://robocode.sourceforge.net, 2008.
4. Jackie O'Kelly and J. Paul Gibson. Robocode& problem-based learning: a non-prescriptive approach to teaching programming. In *Proceedings of the SIGCSE conference on Innovation and technology in computer science education*, pages 217–221, New York, NY, USA, 2006. ACM.
5. David J. Pearce, Matthew Webster, Robert Berry and Paul H. J. Kelly. Profiling with AspectJ. In *Software – Practice and Experience*, pages 747–777. Wiley InterScience, 2006.
6. Sylvia Dieckmann and Urs Hölzle. *A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks*. University of California, CA: Department of Computer Science. Retrieved Oct 11, 2008, from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.9704
7. J. D. Gradecki and N. Lesiecki. *Mastering AspectJ : Aspect-Oriented Programming in Java*. Wiley, 2003.
8. R. Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.
9. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, 1997.
10. Donald E. Knuth. The Art of Computer Programming. Retrieved Oct 11, 2008, from: http://www-cs-faculty.stanford.edu/~uno/taocp.html
11. JGraph - The Java Open Source Graph Drawing Component. Retreived Oct 11, 2008, from: http://www.jgraph.com/jgraph.html
12. Marc Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole. *Java Swing*. O'Reilly, 2003.
13. javac - Java Programming Language Compiler. Retreived Oct 11, 2008, from: http://java.sun.com/javase/6/docs/technotes/tools/windows/javac.html
14. Sheng Liang and Deepa Viswanathan. *Comprehensive Profiling Support in the Java Virtual Machine*. Retrieved Oct 11, 2008, from: http://www.usenix.org/events/coots99/full_papers/liang/liang.pdf