Darwinian Data Structure Selection

Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, Donald Lawrence, and Earl Barr

Abstract—Data structure selection and tuning is laborious but can vastly improve application performance and memory footprint. We introduce ARTEMIS a multiobjective, cloud-based optimisation framework that automatically finds optimal, tuned data structures and rewrites applications to use them. ARTEMIS achieves substantial performance improvements for every project in a set of 29 Java programs uniformly sampled from GitHub. For execution time, CPU usage, and memory consumption, ARTEMIS finds at least one solution for each project that improves all measures. The median improvement across all these best solutions is 8.98% for execution time, 24.27% for memory consumption and 11.61% for CPU usage. In detail, ARTEMIS improved the memory consumption of JUnit4, a ubiquitous Java testing framework, by 45.42% memory, while also improving its execution time 2.29% at the cost a 1.25% increase in CPU usage. LinkedIn relies on the Cleo project as their autocompletion engine for search. ARTEMIS improves its execution time by 12.17%, its CPU usage by 4.32% and its memory consumption by 23.91%.

Index Terms—Search-based software engineering; Genetic improvement; Software analysis and optimisation; Multi-objective optimisation

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

—Donald E. Knuth [1]

1 INTRODUCTION

Under the immense time pressures of industrial software development, developers are heeding one part of Knuth’s advice: they are avoiding premature optimisation. Indeed, developers appear to be avoiding optimisation altogether and neglecting the “critical 3%”. When selecting data structures from libraries, in particular, they tend to rely on defaults and neglect potential optimisations that alternative implementations or tuning parameters can offer. This, despite the impact that data structure selection and tuning can have on application performance and defects. For performance, examples include the selection of an implementation that created unnecessary temporary objects for the program’s workload [2] or selecting a combination of Scala data structures that scaled better, reducing execution time from 45 to 1.5 minutes [3]; memory leak bugs exemplify data structure triggered defects, such as those in the Oracle Java bug database caused by poor implementations that retained references to unused data entries [4].

Optimisation is time-consuming, especially on large code bases. It is also brittle. An optimisation for one version of a program can break or become a de-optimisation in the next release. Another reason developers may avoid optimisation are development fads that focus on fast solutions, like

“Premature Optimisation is the horror of all Evil” and “Hack until it works” [5]. In short, optimisations are expensive and their benefits unclear for many projects. Developers need automated help.

Data structures are a particularly attractive optimisation target because they have a well-defined interface, many are tunable, and different implementations of a data structure usually represent a particular trade-off between time and storage, making some operations faster but more space-consuming or slower but more space-efficient. For instance, an ordered list makes retrieving the entire dataset in sorted order fast, but inserting new elements slow, whilst a hash table allows for quick insertions and retrievals of specific items, but listing the entire set in order is slow. A Darwinian data structure is one that admits tuning and has multiple implementations, i.e. it is replaceable. The data structure optimisation problem is the problem of finding optimal tuning and implementation for a Darwinian data structure used in an input program.

We aim to help developers perform optimisations cheaply, focusing solving the data structure optimisation problem. We present ARTEMIS, a language-agnostic optimisation cloud-based framework that identifies uses of Darwinian data structures and automatically searches for optimal combinations of implementations and tuning parameters for them, given a test suite. ARTEMIS’ search is multi-objective, seeking to simultaneously improve a program’s execution time, memory usage and CPU usage while passing the test suite. ARTEMIS is the first technique to apply multi-objective optimisation to the Darwinian data structure selection and tuning problem.

ARTEMIS changes the economics of data structure optimisation. Given a set of Darwinian data structures, ARTEMIS can run in the background on the cloud, freeing developers focus on new features, searching for optimal solutions to the Darwinian data structure selection problem. ARTEMIS performs source to source transformations. When it finds a solution, it produces program variants that differ from the original program only at constructor calls and relevant type annotations. Thus, ARTEMIS’ variants are amenable,
by design, to programmer inspection and do not increase technical debt. To ease inspection, ARTEMIS generates for each transformation a report, as a diff, that details the changes.

ARTEMIS achieves substantial performance improvements for every project in a set of Java programs uniformly sampled from GitHub. For execution time, CPU usage, and memory consumption, ARTEMIS finds at least one solution for each project that improves all measures. The median improvement across all these best solutions is 8.38% for execution time, 24.27% for memory consumption and 11.61% for CPU usage. For 8, very popular open source GitHub projects (listed in Section 5.1), ARTEMIS achieves median improvements of 14.14% in execution time, 21.88% in CPU usage, and 38.36% in memory consumption. Looking at the results in detail, we find that ARTEMIS improved the memory consumption of JUnit4, a ubiquitous Java testing framework, by 45.42% memory, while also improving its execution time 2.29% at the cost a 1.25% increase in CPU usage. Jimfs is Google’s in-memory file system, a performance critical project. ARTEMIS improved its execution time by 24.6% and its CPU usage by 30% at the cost of 3.4% increased memory consumption.

ARTEMIS builds on and generalises previous work. Seeds [6] also replaces data structures automatically, but scales poorly since its replacement is exhaustive and focuses exclusively on energy optimisation. ARTEMIS finds inefficient use of data structures as a side-effect of optimizing them. Xu et alia’s work warns users about inefficient use of memory by containers; it does automatically replace them [7]. Chameleon [8] uses online JVM monitoring to suggest alternate data structure implementations. Deep Parameter Optimisation [9] precedes ARTEMIS in applying multi-objective search to parameter tuning; it does not, however, consider data structure optimisation. Further, it injects artificial deep parameters whose meaning and significance to developers can be unclear. ARTEMIS, in contrast, searches the natural, developer-defined, search-space of data structure parameters.

The contributions of this paper follow:

- We formalised the Darwinian data structure selection and optimisation problem DS² (Section 3)
- We implemented ARTEMIS, a language-agnostic optimisation framework that automatically discovers and optimises sub-optimal data structures and their parameters and provide it as a service.
- We conducted a large empirical study on 29 randomly selected projects and 8 popular, well-written projects to provide evidence to the effectiveness of our framework. For all 37 subjects, ARTEMIS can successfully find variants that outperform the original for all three objectives. On extreme cases, ARTEMIS discovered 31% improvement on execution time, 70.68% improvement on memory consumption, and 78.86% improvement on CPU usage.

2 Motivating example

In this section we illustrate the motivation of this research by exemplifying a program google-http-java-client that is developed by Google Inc. The evaluation and analysis of a code snippet (Listing 1) from this program demonstrates why the data structure selection and optimisation problem is important and must be concerned by the software engineering community.

Listing 1: A function from google-http-java-client.

```java
<T> List<T> getAsList(T value) {
  if (value == null)
    return null;
  List<T> result = new ArrayList<T>();
  result.add(value);
  return result;
}
```

Listing 1 (getAsList) is used to package the HTTP headers and is invoked by other methods of this program frequently. In Listing 1, ArrayList is chosen for instantiating the variable result. However, there are other List implementations that share the same functionality while perform differently in non-functional properties. Thereby, changing ArrayList to other List implementations may affect the performance of the program. In this example, a variant program is generated by replacing ArrayList (line 4) with LinkedList and is compared with the original program by executing both against the same test set (for more details on the profiling methodology see Section 4) for 30 runs. The result shows that, by shifting ArrayList to LinkedList in Listing 1 (combined with some other data structure changes), the google-http-java-client can gain 9.9% improvement on execution time and reduce 47.1% memory consumption (see Section 5 for more details). Such results provide motivating evidence for using data structure selection and parameter tuning as ways to improve a program’s runtime performance.

The framework proposed here, ARTEMIS, can automatically discover underperformed data structures and replace them with better choices using search-based techniques. By creating automatically a store of data structures from the language’s Collection API library (see Section 4.1), our framework traverses the program’s AST to identify which of those data structures are used and exposes them as parameters by transforming the code (line 4) into the code shown below. Here D is the tag that refers to the exposed parameter associated with the defined data structure type (more details in Section 4).

```java
<T> result = new D<T>();
```

Additionally, in Listing 1 we also noted that the argument referring to the initial capacity size of the ArrayList was not specified, thereby the default size 10 was used. We also observed that, the instantiated List object only contains one item, leaving the initial capacity size setting may result in unnecessary memory bloat. Tuning the initial capacity size of the ArrayList enables to control the amount of memory pre-allocated. However, an inopportune large value of S may result in memory waste, while an overly small value may force the program to allocate new memory frequently and shortly during the execution, thereby harms to the execution time. Therefore, an appropriate value may be chosen to ease memory bloat while maintaining the time performance simultaneously.

ARTEMIS automatically exposes such arguments as tun-
able parameters, which will be later adjusted to improve the runtime performance of the program. For instance, the code in Listing 1 will be changed into the code below, where \( S \) is a parameter tag.

```java
List<T> l = new ArrayList<>(S);
```

The following command (more details about tool usage in Section 4.5) shows an example of using ARTEMIS:

```
./artemis google-http-java-client output
```

## 3 Problem Formulation

This section formally defines the data structure and parameter optimisation problem we are to solve in this paper.

**Definition 1** (Abstract Data Type). An Abstract Data Type (ADT) [10] is class of objects whose logical behavior is defined by a set of values and a set of operations.

A data structure is an implementation of Abstract Data Type (ADT) that supports a certain set of operations. In an Object-oriented (OO) language, an ADT is usually represented as an interface, therefore, a data structure is usually an implementation or subclass of an interface or a superclass. For example, in Java language, `ArrayList` is an implementation of interface `List`, therefore `ArrayList` is a data structure. Note that, an ADT itself can be a data structure if it is an implementation of another ADT, therefore `List` is also a data structure since it is an implementation of ADT Collection. We shall use \( d \) to refer to a data structure and \( I(d) \) to refer to the ADT it implements. We also use \( D \) to refer to the set of all data structures in the target language.

Before we can formally define the problem, two notations are firstly introduced to describe the properties of a data structure.

**Notation 1**: Given \( \varphi(D) \) is the powerset of \( D \), then \( \varphi : D \mapsto \varphi(D) \) is a function that maps a data structure to a set of interchangeable data structures that are semantically equivalent to it. Formally, the function \( \varphi \) is defined in Eq. 1. For instance, `ArrayList` and `LinkedList` both implement the interface `List`, or \( I(\text{ArrayList}) = I(\text{LinkedList}) = \text{List} \), then we say `ArrayList` and `LinkedList` are semantically equivalent, or `LinkedList \in \varphi(\text{ArrayList})`. In particular, \( d \in \varphi(d) \) for any data structure \( d \), and if \( d_1 \) and \( d_2 \) are semantically equivalent, then \( d_1, d_2 \in \varphi(d_1) = \varphi(d_2) \).

\[
\varphi(d) = \{ d' | I(d') = I(d) \}
\]  

Darwinian data structure: a data structure \( d \) is called Darwinian data structure if \( |\varphi(d)| > 1 \). Note that \( \forall d, d \in \varphi(d) \), thus \( |\varphi(d)| = 1 \) means no other data structures can replace \( d \) and the program cannot be improved by replacing it. Therefore, we only consider Darwinian data structures in this paper whilst all data structures in the remainder of the paper are refereed as Darwinian data structures unless specified otherwise.

**Notation 2**: \( N : D \mapsto \mathbb{N} \) is a function that maps a data structure to the number of numerical arguments it takes when it is initialised. For instance, `ArrayList` takes one argument representing the initial capacity of the list, therefore \( N(\text{ArrayList}) = 1 \).

For a program under optimisation (PUO), there can be multiple identifiable data structures that can be replaced by other semantically equivalent data structures. We use \( \vec{p} = (p_1, p_2, \cdots, p_k) \) to represent a program with a vector of \( k \) data structures identified in a PUO, where \( p_i = (d_i, \vec{x}_i) \) is a pair of data structure \( d_i \) and its numerical arguments \( \vec{x}_i \in \mathbb{R}^{N(d_i)} \).

Because each of the data structures \( d_i \) can be replaced by another data structure \( d'_i \in \phi(d_i) \) and the program remains correct, we can search for different combinations of these data structures in order to improve the program’s non-functional properties (e.g., execution time and memory consumption). Therefore, for a program with original data structures \( \vec{p}_{ori} = (p_{ori,1}, p_{ori,2}, \cdots, p_{ori,k}) \), the search space is formed as Eq. 2:

\[
S(\vec{p}_{ori}) = \{ \vec{p} = (p_1, p_2, \cdots, p_k) \}
\]  

where \( p_i = (d_i, \vec{x}_i), d_i \in \phi(d_{ori,i}), \vec{x}_i \in \mathbb{R}^{N(d_i)} \) for \( i = 1, 2, \cdots, k \).

Suppose we are to optimise a list of replaceable data structures and their arguments in a program, according to a (maximising) fitness function \( f : S \mapsto \mathbb{R} \), we can define our Data Structure and Parameter Optimisation problem as:

**Darwinian Data Structure Selection (DS\(^2\) problem**: given a program with a list of replaceable data structures \( \vec{p}_{ori} \), find the optimal combination of data structures and their arguments \( \vec{p} \in S(\vec{p}_{ori}) \), such that \( \forall \vec{p}' \in S(\vec{p}_{ori}), f(\vec{p}) > f(\vec{p}') \).

## 4 Implementation

Figure 1 illustrates the architecture of the proposed optimisation framework. It consists of three main components: the DARWINIAN DATA STRUCTURES STORE GENERATOR (DDSSG), the EXTRACTOR, and the OPTIMISER. ARTEMIS takes the language’s Collection API library, the user’s application source code and a test suite as input to generate an optimised version of the code with a new combination of data structures. The DDSSG automatically builds a store that contains data structure transformation rules. The EXTRACTOR uses this store to discover potential data structure transformations and exposes them as tunable parameters to the OPTIMISER (see Section 4.2). The OPTIMISER uses a multi-objective search algorithm (NSGA-II [11]) to tune the parameters [9], [12], [13], [14], [15] and to provide optimised solutions (see Section 4.4). A regression test suite is used to maintain the correctness of the transformations and to evaluate the non-functional properties of interest. ARTEMIS uses a built-in profiler that measures execution time, memory consumption and CPU usage.

### 4.1 Darwinian Data Structure Store

Given as input a library from any programming language, the scope is to automatically build a store of Darwinian Data Structures that can be exposed as tunable parameters to the OPTIMISER. In this section, we describe how we build such a store for the Java collection API and how it can be applied for other APIs of other programming languages as well. Note that the DDSSG can expose other equivalent implementations beyond data structures as we will explain next.

The basic assumption of this approach is that the DDSSG takes as input a Graph Class Hierarchy, similar to the one illustrated in Figure 2. This graph shows what are the potential Equivalent implementations of a specific interface and
the hierarchy between those interfaces. The Class Hierarchy is not difficult to generate manually for a specific language, however, in order to be more generic and language agnostic, ARTEMIS can automatically generate the hierarchy graph from the source code of the library (if provided) or from the library documentation.

To get the hierarchy graph from the source code, the DDSSG traverses the AST of each file of the library and looks for class declaration expressions. It extracts the classes and stores them as points of a graph. Whenever it finds a special keyword, such as extends or implements in Java, it creates an edge in the graph that represents this relationship. After the graph construction is finished, a graph traversal is used to automatically generate a store with rules for equivalent implementations of the same interface; e.g., rule1: IList, ArrayList, LinkedList. Those implementations will be considered as replaceable during code execution and will be exposed as parameters to the OPTIMISER.

The store can also be extended with custom user implementations or with implementations from other libraries. Currently, ARTEMIS supports implementations from third-party libraries such as Google Guava Collections³, fastutil⁴ and Apache Commons Collections⁵.

4.2 Discovering Darwinian Data Structures

The EXTRACTOR takes as input the program source code, identifies potential locations of the code that contains DARWINIAN data structures, and provides as output a list of parameters (Extracted Data Structures and Parameters in Figure 1) and a templated version of the code which replaces the data structure with data structure type identifiers (Templated Source Code in Figure 1).

In order to determine which parts of the code contain DARWINIAN data structures, the EXTRACTOR firstly generates an Abstract Syntax Tree (AST) from the input source code. It then traverses the AST to discover potential data structure transformations based on a store of data structures as shown in Table 1. For example, when an expression node of the AST contains a LinkedList expression, the EXTRACTOR marks this expression as a potential DARWINIAN data structure that can take values from the available List implementations: LinkedList or ArrayList. The EXTRACTOR maintains a copy of the AST, referred to as the REWRITER, where it applies transformations, without changing the initial AST. When the AST transformation finishes, the REWRITER produces the final source code which is saved as a new file.

4.3 Code Transformations

While analysing the code of the subjects used during the evaluation (detailed in Section 5), we identified a set of code patterns for data structure usages that may negatively affect the optimisation process (increases the search space). Those patterns contain code that show how users sometimes do not follow good programming practices and write high coupling code when using data structures. A simple example to illustrate such patterns is provided in Listing 2.

---

3. https://github.com/google/guava
4. https://github.com/vigna/fastutil
5. https://github.com/apache/commons-collections
In lines 2 and 8 we see two `LinkedList` variables that could potentially be marked as DARWINIAN and replaced by their equivalent `ArrayList` implementation. In the same lines we notice that the user does not "program to the interface (List)" but uses the concrete type (`LinkedList`) to declare the variable. This is a bad programming habit [16] as it adds dependencies in the code and limits code reuse. More than a single code change are necessary in order to replace the type of the data structure. In addition, we can see that `func3` takes as parameter a `LinkedList` and not the supertype `List`. This means that `func3` will produce a compilation error if the data structure type in `func1` changes.

The approach adopted to extract and transform data structures affects significantly the number of data structures parameters the EXTRACTOR exposes to the OPTIMISER. A bad transformation approach may lead to a huge number of DARWINIAN data structures and many unnecessary solutions that will fail to compile. This may decrease the performance of the optimisation process significantly or in the worst case scenario make it impractical. For example, a bad transformation is to mark as DARWINIAN a data structure that uses a specific method only available for its implementation.

To deal with such patterns, we implemented three transformation modes as follows:

**Exhaustive replacement.** In the simplest case, we could mark every `LinkedList` as a DARWINIAN data structure. This approach however would generate a very big number of parameters for the OPTIMISER and most of the solutions would fail to compile; e.g., in Listing 2, 6 parameters would be generated for all `List` data type occurrences and a total number of $2^6$ combinations of data structure selections, supposing that we have only `LinkedList` and `ArrayList` as equivalent data structures. The advantage of this approach is that it is easy to be implemented as it does not need any additional code analysis.

**Conversion to the supertype.** A better approach is to convert the subclass variable declaration to the more general supertype of the parent class. For example, the code in Line 2, 7, 8 and 11 could use `List<T>` instead of `LinkedList<T>` as variable type because the only methods used are the ones of the parent class `List`. In that case, for Listing 2, the EXTRACTOR would expose only 2 DARWINIAN data structures as 4 of them would be converted to `List`. Providing equivalent implementations for those two exposed parameters would be unlikely to break the code.

To convert the concrete implementation to its super type for all code occurrences, it is necessary to use an API of a re-factor tool (Eclipse IDE's re-factor tool). Also, additional analysis should be performed to check this transformation does not break the code. ARTEMIS aims to be language agnostic without any additional dependencies. Therefore, ARTEMIS performs automatically this transformation through the OPTIMISER by adding the supertype as an equivalent parameter in the store of data structures. Whenever the AST visitor traverses a variable or parameter declaration expression it may replace the DARWINIAN data structure with its supertype.

**Dynamic profiler.** A program may contain a large number of data structures from which only some are DARWINIAN. Moreover, some of those DARWINIAN data structures can affect the performance of the program more than others. There are data structures that store only a few items and be called only a few times during program execution and, as a consequence, changing them will most probably not provide any significant improvement.

In our implementation, we have introduced a preprocessing step that automatically instruments the program to provide profiling details when it is executed the first time. Note that this profiling information can be provided by other Java profiling tools if desired. The instrumented code is run before the optimisation begins and it generates a database with the most costly methods worth optimisation. This information is used by the EXTRACTOR to determine if a data structure is worth being considered as a DARWINIAN data structure. This preprocessing step is mostly useful for very large programs where there is a large number of data structures involved.

### 4.4 Search Based Parameter Tuning

The scope of the OPTIMISER is to find a combination of data structures that improves the performance of the initial program. Practically, we can represent all those data structures as parameters that can be tuned using Search Based Software Engineering approaches [17]. Because we aim to optimise various conflicting performance objectives, we consider this as a multi-objective optimisation problem, thus the OPTIMISER uses a multi-objective Genetic Algorithm [11] to search for optimal solutions.

We use an array of integers to represent the tuning parameters. Each parameter may refer either to an equivalent data structure or to the initial size of that data structure. Note that if the parameter refers to a data structure, its value represents the index in the list of equivalent data structures. The OPTIMISER keeps additional mapping information to distinguish the types of the parameters. For each generation, the NSGA-II progresses by firstly applying tournament selection, followed by a uniform crossover and a uniform mutation operation. In our experiments, we designed fitness functions to capture execution time, memory consumption and CPU usage. After fitness evaluation, a standard NSGA-II non-dominated selection is applied to form the next generation. This process is repeated until the solutions in a generation converge. Finally, all non-dominating solutions in the final population are provided as solutions.

```java
public class Example {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("hello");
        list.add("world");
        System.out.println(list.get(0));
    }
}
```
Two methods are performed to select the subjects: (a) uniformly select 29 subjects from all programs available on GitHub which follow criteria 1 from the criteria list, bellow; and (b) manually pick 8 popular subjects that follow all the following criteria:

1. Project is written in Java and passed the compilation and testing.
2. Project provides adequate test suites, the line coverage ratio must be greater than 70%.
3. Well-known real-world project that receives at least 200 stars on GitHub.
4. The projects have to be diverse.

We chose to optimise code written in Java, because of the popularity of the Java Collection API. ARTEMIS can be easily extended to optimise code written in other languages as it has few dependencies on language specific tools. More specifically, ARTEMIS depends only on the existence of a parser for the language and a profiler that will provide the performance objectives. Moreover, we use ANTLR [18] for generating the AST of the program, which already supports most of the popular programming languages8.

The input projects must contain test suites, for the OPTIMISER to check the correctness of the generated variants and to evaluate the performance of the program. Ideally, each program should have both a regression test suite with high statement coverage for maintaining the correctness of the program and a performance test suite for evaluating the non-functional properties of the program. However, most real world programs do not provide such performance test suite and for this reason we use the regression test suite to evaluate the non-functional properties.

Subjects should be popular real world programs that a lot of people use (we use the official popularity from GitHub9 statistics to measure the popularity of a program). For those well-known projects, they are usually well-written and optimised, where a group of experienced developers may be involved. We include those project to investigate whether they use a better combination of data structures than the one that an automatic tool such as ARTEMIS can provide.

The projects have to be diverse. We would like to apply our approach on a wide range of projects from different areas with different sizes, to demonstrate the generality of this approach.

From Table 2, we can see that the selected subjects have a good diversity, including static analyser tools, testing frameworks, web clients, graph processing applications, etc.. The sizes of the projects vary from 576 to 94K lines of code and the popularity varies from 0 star to 5642 stars. It should be clear that the popular projects have much more stars than the uniformly selected projects in general (more stars means more popular).

5.2 Experimental Set Up

Experiments were conducted in the machines which are all built with Oracle JDK 1.8.0 and Ubuntu 16.04.4 LTS. The configuration of the machines is based on one Intel E5-2673v3 CPU featuring 8 cores and 14GiB of DRAM. To mitigate instability and incorrect results [19], [20], we differentiate

### Table 2: Subject projects studied in this research.

| Project                           | #Star | #Loc | #Test | Coverage (%) |
|-----------------------------------|-------|------|-------|--------------|
| apache/commons-validator           | 52    | 2320 | 527   | 95.9%        |
| BillServ/jopproxy                 | 2     | 2844 | 28    | 91.3%        |
| bced/castorcolour                 | 5     | 11663| 38    | 91.3%        |
| camunda/camunda-bpm-mockito       | 8     | 2857 | 33    | 91.3%        |
| dick-the-deployer/dick-worker     | 0     | 1348 | 18    | 91.3%        |
| fuineng/event-store-commons       | 4     | 12652| 208   | 91.3%        |
| fcrepo4/fcrepo4                    | 77    | 33308| 640   | 91.3%        |
| gabe-alex/HospitalInfectionsMonitoringSystem | 0   | 1033 | 1    | 91.3%        |
| google/google-http-java-client     | 572   | 20637| 636   | 91.3%        |
| HotelsDotCom/plunger              | 26    | 3865 | 175   | 91.3%        |
| imagoj/imagej-ops                 | 19    | 48267| 792   | 91.3%        |
| jhoonewart/shuzai                  | 0     | 680  | 8     | 91.3%        |
| jenkinsci/confluence-publisher-plugin | 13 | 24927| 19    | 91.3%        |
| knor/tap-plugin                   | 4     | 2844 | 34    | 91.3%        |
| linkis/linkis                      | 532   | 13922| 75    | 91.3%        |
| lipsRomani/control-financiero-spring-boot | 0   | 1058 | 1    | 91.3%        |
| Lynchmaniac/poliight              | 2     | 2192 | 35    | 91.3%        |
| Isloan/OpenLRS                     | 0     | 5229 | 28    | 91.3%        |
| myus/uma                          | 3     | 1344 | 18    | 91.3%        |
| nazaryan/shipt                       | 0    | 3207 | 61    | 91.3%        |
| PEXPlugins/PermissionEx            | 356   | 12956| 63    | 91.3%        |
| pfichtner/pranalyzer               | 8     | 14962| 78    | 91.3%        |
| punetaasival/dropwizzard           | 0     | 29116| 873   | 91.3%        |
| rayzeng/queue                     | 0     | 3929 | 10    | 91.3%        |
| roby-rodriguez/rabix-vertifier     | 0     | 576  | 3     | 91.3%        |
| sarris/mnpiparser                  | 81    | 39721| 170   | 91.3%        |
| timmoller/XChange                   | 741   | 94059| 588   | 91.3%        |
| xored/verys-typed-rpc              | 6     | 705  | 16    | 91.3%        |
| zilaiyedaren/zxing                 | 1     | 42521| 378   | 91.3%        |

8. https://github.com/antlr/grammars-v4/
9. https://github.com/trending
VM start-up and steady-state. To assure the accuracy and reduce the bias in the measurement, program profiling period was set as 0.1 seconds, and each generated solution was run for 30 simulations. Also we use Mann Whitney U test [21] to examine if the improvement is statistically significant.

To measure the memory consumption and CPU usage of a subject program, we use the popular JConsole profiler because it uses the stats directly from JDK, and it provides an easy programmable API. We extended JConsole to monitor only those processes that refer each test of the provided test suite. To measure the execution time of the test suite, we use the Maven Surefire plugin, which reports only the execution time of each individual test excluding the measurement overhead that other Maven plugins may introduce.

For the settings of the optimisation algorithm, we used an initial population size of 30 and a maximum number of 900 function evaluations. The tournament selection (based on ranking and crowding distance), simulated binary crossover (with crossover probability 0.8) and polynomial mutation (with the mutation probability 0.1) are used as genetic operators. These settings were determined through a few calibration trials to ensure the maturity of the results. Since the non-deterministic nature of NSGA-II, each experiment was executed for 30 times to obtain statistical power of the results.

5.3 Research Questions and Results Analysis

Our approach automatically transforms the initial source code of the application in order to improve three objectives. The question begs ‘Whether ARTEMIS can improve all three objectives or at least partially?’ This motivates our first research question:

RQ1: What proportion of the programs can ARTEMIS automatically improve?

To investigate RQ1, we collect the non-dominated set of solutions from 30 runs of each subject. In order to compare those multi-objective solutions with the initial objectives of the program we use the terms strictly dominate relation and incomparable relation. These are defined by Zitzler et al. [22] as:

A solution strictly dominates another solution if it outperforms the latter in all measures. A solution is said incomparable with another solution if both outperform each other in at least one of the measures. For brevity, in Table 3 we use strong improvement to represent a strictly dominate relation, and weak improvement for a incomparable relation.

Table 3 provides the summary of the solutions that ARTEMIS found for both uniformly selected programs as well as the popular ones. Column 2 reveals the percentage of programs for which ARTEMIS provides at least one weakly improved solution. Column 3 presents the percentage of programs for which ARTEMIS finds at least one strongly improved solution. We can clearly see that for all programs ARTEMIS provides at least one strongly solution. Column 4 outlines the proportion of strongly improved solutions from the total number of solutions for all subjects. From the table we observe that, there are 35.30% strongly improved solutions for uniformly selected programs and 26.50% for the popular ones.

From the above-mentioned results, we answer RQ1 by clearly stating that, for all subjects, ARTEMIS can successfully provide optimised variants that outperform the original program in execution time, memory and CPU usage at the same time.

After the first question is answered, the second question naturally follows:

RQ2: What is the average improvement that ARTEMIS provides for each program?

Though ARTEMIS aims to improve all measures, because sometimes these measures are conflicting, the improvement of one measure may lead to a decrease in another. Therefore, we expect to observe solutions that slightly improve all measures, as well as solutions that improve one measure drastically at the expense of other measure.

In some domains it is more important to improve significantly one of the measures than to improve slightly all measures; e.g., a high frequency trading application may want to pay the cost of additional memory overhead in order to improve the execution time. Our intuition is that the OPTIMISER will find a large number of pareto-front solutions and there will be for each measure at least one solution that improves the measure significantly.

We answer RQ2 quantitatively using our study for 37 projects. We first report the maximum improvement for each of the three measures (execution time, memory consumption, CPU usage) both for popular (Figure 3) and uniformly selected subjects (Figure 4). We use bar charts to plot the three measures for each program. In Y axis we represent the percentage of improvement for the three measures. A value less than 100% represents an improvement and a value greater than 100% means degradation; e.g., 70% memory consumption means that the solution consumes 70% of the memory used in the initial program.

Figure 3a presents the three measures of the solutions when the execution time is minimised, for each program from the popular GitHub programs. The first thing to notice is that ARTEMIS improves the execution time for every program. The program of which the execution time was improved the most is gson, where the execution time was reduced by 30.99%. We also notice that this execution time improvement did not affect negatively on other measures, but instead the memory consumption was reduced by 7.47% and CPU usage remained almost the same. The other interesting program to notice from this graph is the commons-validator as the execution time was improved by 28.27% but the memory consumption was increased by 57.24%. Note however that ARTEMIS can be instrumented to automatically skip this type of solutions if desired so; the user may specify the limit of

| programs                          | Improved programs | Strongly improved solutions |
|-----------------------------------|-------------------|-----------------------------|
| Uniformly selected programs       | 100%              | 35.30%                      |
| Popular programs                  | 100%              | 26.50%                      |
| All programs                      | 100%              | 32.85%                      |
The program with the most significant improvement is gson with 78.86%, and with least improvement is Graphjet, with 10.81%.

Figure 4a presents the solutions that provide the best improvement in execution time for each program from the uniformly selected GitHub programs. The first thing to notice is that ARTEMIS improves the execution time for every program and the median improvement is 7.96%. The median value for memory consumption is slightly increased by 1.64% and CPU usage decreased by 4.32%. If we neglect programs fqueue, google-http-java-client, polilight and XChange we observe that there was no significant decrease in memory or CPU usage for the majority of the programs. The most significant execution time improvement is achieved by program jobproxy with 30.47% and uma vertex-typed-rpc with 30.32%.

Figure 4b presents the solutions that provide the best improvement in memory consumption for each program from the uniformly selected GitHub programs. ARTEMIS improves the memory consumption for every program and the median improvement is 17.87%. The median value for execution time remained almost the same as it increased by 0.25%. CPU usage was increased by 4.42% because of google-http-java-client and XChange programs, which have significant CPU usage degrading (77.93% and 112.47% respectively). However, those programs are among the ones with the highest memory improvement as google-http-java-client has 62.58% improvement and XChange has 46.16%. Finally, the program with the most significant memory improvement is zxing with 70.78%.

In summary, we can clearly answer RQ2 and mention that ARTEMIS improves all metrics for each of the 37 programs of our study with a median of 8.38% for execution time, 24.27% for memory consumption and 11.61% for CPU usage.

**Research question 3:** What is the cost of using ARTEMIS?

In order for ARTEMIS to be practical and useful in real-world situations, it is important to understand the cost of using ARTEMIS. The aforementioned experimental studies revealed that, even for the popular programs, the selection of the data structure and the setting of its parameters may be suboptimal. Therefore, optimising the data structures and their parameters can still provide significant improvement on the non-functional properties.

To answer this research question, the cost of ARTEMIS for optimising a program is measured by the cost of computational resources it uses. In this study a Microsoft Azure™ D4-v2 machine, which costs £0.41 per hour, was used to conduct all experiments.

The experiments show that an optimisation process takes 3.05 hours on average for all studied subjects. The program degradation for a specific measure (see Section 4.5). Finally, for this set of solutions, the median percentage of execution time improvement is 14.13%, whilst memory consumption is slightly increased by 1.99% and CPU usage is decreased by 3.79%.

Figure 3b shows the solutions that provide the best improvement in memory consumption for the selected popular GitHub programs. We can clearly notice that once again ARTEMIS improves the memory consumption for all programs. We also see that on average memory consumption is improved even more than the execution time in the previous graph, with a median value of 39.36%. The median value of execution time remains almost the same, with a slight decrease of 0.25%, while the median value of CPU usage is increased by 4.42%. We notice that Jafka program has the best improvement by 65.8% and Graphjet the minimum improvement of 7.23%. For all programs the memory improvement is achieved without any significant time impact; if we neglect commons-validator and gson, there is no significant CPU usage impact either.

Figure 3c presents the solutions that provide the best improvement in CPU usage for each program from the popular ones. The median CPU usage improvement is 21.78%. The median value of Execution time was improved by 3.25% and the median value of memory consumption was increased by 4.42%. We see that ARTEMIS improved the CPU usage for all programs and if we neglect truth and gson there was no significant impact on the other measures.
(a) Best execution time of uniformly selected GitHub programs. The median value is 92.04%.

(b) Best memory consumption of the uniformly selected GitHub programs. The median value is 82.63%.

(c) Best CPU usage of the uniformly selected GitHub programs. The median value is 92.22%.

Fig. 4: Answers RQ2. Description.

XChange and jrunalyzer are the most and the least time-consuming programs respectively, among all 37 subjects. ARTEMIS spent 19.16 hours on average to optimise the program XChange whilst only 3.12 minutes for program jrunalyzer. Accordingly, the average cost of applying the proposed approach ARTEMIS for the subjects studied is £1.25, and the range of cost is from £0.02 to £7.86. The experimental results show that overall cost of using ARTEMIS is negligible compared to a human software engineer.

Moreover, ARTEMIS transforms the selection of data structure and sets the parameter on source code level, which means such optimisation does not need to be carried frequently. It enables the software engineer to look into the changes to gain insight about the usage of data structure, and to better understand the characteristics of the program.

In summary, the cost of using ARTEMIS is negligible, with an average of £1.25.

6 THREATS TO VALIDITY

Like any empirical study, there are potential threats to validity for our experimental results:

Internal validity: Threats to internal validity concern issues that may perturb the experimental evaluations. The perturbations may include inappropriate parameter settings, wrong profiling measurements and the implementation of algorithms. Inappropriate parameter settings may lead to premature results or inefficiency of the experiments. To minimise the threat, we conducted a few calibration experiments to adjust the parameters such that, the algorithm converges at a relatively fast speed and stops after the results become stable. For measuring the non-functional properties, we carefully chose JConsole profiler that directly gathers runtime information from JDK, such that the measurement error is minimised. Moreover, we carefully extended JConsole to further improve the precision of the measurements. Therefore the threat from the measurements is minimised. To cater for the stochastic nature of the approximated heuristic algorithm and to provide the statistic power for the results, we run 30 times each experiment.

External validity: Our choice of benchmark programs and their associated test suites influences the generality of our results. Even a good test suite that achieves, for example, high branch coverage, could still differ from real world inputs, in which case the optimised configuration over this test suite may not achieve the best performance.

Regarding the subjects that we used for our experiments, as we described in Section 5.1, we conducted a large-scale sampling for all Java projects in GitHub and we uniformly selected 29 of them. Moreover, we carefully selected 8 popular subjects that come from different areas to achieve a high diversity in our subjects. Therefore, the threat regarding the generality of the subjects is alleviated.

The scalability of our approach is another threat to external validity. The subjects involved in this study vary in size, from hundreds of lines of code to almost a million lines of code. According to our experimental results, the effectiveness of our approach remains the same from small subjects to large subjects. Therefore, it is expected that our
approach will scale up to even larger subjects and this threat is reduced.

In this study, we only included projects written in Java, therefore, whether our tool works for other languages remains a threat. When designing the framework, we implemented the tool in the way that it does not assume any characteristics of specific programming languages and does not rely on any language-specific library. Therefore our tool can be easily extended to other language, and the threat is alleviated.

7 Related Work

Multi-objective Darwinian Data Structure selection and optimisation stands between two areas a) searched based software engineering and data structure performance optimisation.

Search based software engineering optimisation

Previous works have applied Genetic Programming [23] to either improve the functionality (bug fixing) [24] or non-functional properties of a program [13], [14], [25]. Their approaches use existing code as the code base and replace some of the source code in the program under optimisation with the code from the code base. However, their approaches rely on the Plastic Surgery Hypothesis [26], which assumes that the solutions exist in the code base. Our approach, on the other hand, does not rely on the hypothesis but relies on a set of transformation rules. Our approach can automatically generate these transformation rules from the library code or library documentation exhaustively, therefore the approach guarantees a comprehensive set of transformation rules.

Wu et al. [9] introduced a mutation-based method to expose “deep” parameters (similar to the ones we optimised in this paper) from the program under optimisation, and tuned these parameters along with “shallow” parameters to improve the time and memory performance of the program. Though the idea of exposing additional tunable parameter is similar to our approach, their approach did not optimise any data structure selection, which can sometimes be more rewarding than just tuning the parameters. Moreover, their approach was applied on memory management library to affect the time and memory performance of the programs that using this library. Therefore the extent of improvement usually depends on how much a program relies on the library. On the other hand, our approach directly applies to the source code of the program, relying on no assumptions on the program, therefore our approach has a wider applicability.

Data structure optimisation and bloat

There is a group of work [27], [28], [29], [30] that has attempted to identify bloat related with the usage of data structures. In 2009, Shacham et al. [8] introduced a semantic profiler that provides online collection-usage semantics for Java programs. They instrumented Java Virtual Machine (JVM) to gather the usage information of collection data structures. Along with some manually-generated rules, they can automatically provide hints to users when there is a better choice for a data structure in the program. Because the data structure selection rules are manually generated, this approach requires human knowledge as input. Though the rules can be extended by users, if the users do not have enough knowledge about the data structures, they may introduce bias to the rules and jeopardise the effectiveness of the approach. Instead, in our approach, we directly use the performance of a data structure profiled against a set of performance tests to determine the optimal choices of data structures. Therefore our approach does not depend on human knowledges on the data structures, whilst the selection bias is minimised by a set of carefully-chosen performance tests. Furthermore, our approach directly modifies the program instead of providing hints, thus users can use the fine-tuned program provided by our approach without any additional manual adjustment.

In 2014, Manotas et al. [6] introduced a collection data structure replacement and optimisation framework named SEEDS. Their framework replaces the collection data structures in Java applications with other data structures exhaustively and automatically select the most energy efficient one to improve the overall energy performance of the application. In our study, we used a similar but extended approach to optimise both the data structures and additional parameters. We also extended the optimisation objectives from single objective to triple objectives and used Pareto non-dominated solutions to show the trade-offs between these objectives. Due to a much larger search space in our problem, the exhaustive exploration search that used by SEEDS is not practical, therefore we adopted meta-heuristics algorithms to solve the problem. Furthermore, our approach directly transforms the source code of the programs whilst SEEDS transforms the bytecode, therefore our approach can provide human developers more intuitive information about what was changed and encourage them to use more efficient data structures in the future. Moreover, our approach can be more easily applied to other languages as it does not depend on language specific static analysers and refactoring tools such as WALA [31] and Eclipse IDE’s refactoring tools. In order to support another language we just need the grammar of that language; ANTLR currently provides many available grammar languages [32] and to implement a visitor that will just extract the darwinian data structures.

Apart from the novelties mentioned above, this is the largest empirical study to our knowledge in similar works. In the studies mentioned above, only 4 to 7 subjects were included in the experiments. Our study included 29 uniformly selected subjects and 8 well-written popular subjects to show the effectiveness of our approach, therefore our results are statistically more meaningful.

8 Conclusion

Developers frequently use underperformed data structures and forget to optimise them with respect to some critical non-functional properties once the functionalities are fulfilled. In this paper, we introduced ARTEMIS, a novel multi-objective search-based framework that automatically selects and optimises the data structures and their arguments in the given program. ARTEMIS is language agnostic, meaning it can be easily adapted to any programming language. For any specific language, ARTEMIS first automatically generates a data structure store that represents the interchangeable data structures from library source code or documentations. Using the data structure store, it can automatically detect and optimise DARWINIAN data structures along with any additional

10. https://github.com/antlr/grammars-v4/
parameters to improve the non-functional properties of the given program. In a large empirical study on 29 uniformly chosen projects and 8 well-written popular projects, ARTEMIS found strong improvement for all of them. On extreme cases, ARTEMIS found 30.99% improvement on execution time, 70.68% improvement on memory consumption and 78.86% improvement on CPU usage. At last, we estimated the cost of optimising a program in machine hours. With a price of £0.41 per machine hour, the cost of optimising any subject in this study is less than £8, with an average of £1.25. Therefore, we conclude that ARTEMIS is a practical tool for optimising the data structures in large real-world programs.

ACKNOWLEDGMENTS

The authors greatly thank reviewers for their helpful suggestion. The work reported herein was supported by the China Scholarship Council (CSC) and the University College London.

REFERENCES

[1] D. E. Knuth, “Structured programming with go to statements,” ACM Comput. Surv., vol. 6, no. 4, pp. 261–301, Dec. 1974. [Online]. Available: http://doi.acm.org/10.1145/356635.356640

[2] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, “Go with the flow: profiling copies to find runtime bloat,” ACM Sigplan Notices, vol. 44, no. 6, pp. 419–430, 2009.

[3] R. J. Nowling, “Gotchas with Scala Mutable Collections and Large Data Sets,” http://rnowling.github.io/software/engineering/2015/07/01/gotcha-scala-collections.html, 2015. [Online; accessed 18-February-2017].

[4] G. Xu and A. Rountev, “Precise memory leak detection for java software using container profiling,” in Software Engineering, 2008. ICSE’08. ACM/IEEE 30th International Conference On. IEEE, 2008, pp. 151–160.

[5] B. Hardin, “Companies with ‘hacking’ cultures fail,” https://blog.breathrd.in/companies-with-hacking-cultures-fail-b8907a69e3d#.fdkyb1w2, 2016. [Online; accessed 25-February-2017].

[6] I. Manotas, L. Pollock, and J. Clause, “Seeds: A software engineer’s energy-optimization decision support framework,” in Proceedings of the 36th International Conference on Software Engineering (ICSE). ACM, 2015, pp. 1375–1382.

[7] N. Mitchell, G. Sevitsky, and H. Srinivasan, “Modeling runtime behavior in framework-based applications,” in European Conference on Object-Oriented Programming. Springer, 2006, pp. 429–451.

[8] N. Mitchell and G. Sevitsky, “The causes of bloat, the limits of health,” in ACM SIGPLAN Notices, vol. 42, no. 10. ACM, 2007, pp. 245–260.

[9] N. Mitchell, G. Sevitsky, and H. Srinivasan, “Modeling runtime behavior in framework-based applications,” in European Conference on Object-Oriented Programming. Springer, 2006, pp. 429–451.

[10] N. Mitchell, G. Sevitsky, and H. Srinivasan, “Modeling runtime behavior in framework-based applications,” in European Conference on Object-Oriented Programming. Springer, 2006, pp. 429–451.

[11] N. Mitchell, G. Sevitsky, and H. Srinivasan, “Modeling runtime behavior in framework-based applications,” in European Conference on Object-Oriented Programming. Springer, 2006, pp. 429–451.