Darwinian Data Structure Selection

Michail Basios
University College London, UK
michail.basios@cs.ucl.ac.uk

Lingbo Li
University College London, UK
lingbo.li@cs.ucl.ac.uk

Fan Wu
University College London, UK
fan.wu@cs.ucl.ac.uk

Leslie Kanthan
University College London, UK
l.kanthan@cs.ucl.ac.uk

Earl T. Barr
University College London, UK
e.barr@cs.ucl.ac.uk

ABSTRACT

Data structure selection and tuning is laborious but can vastly improve an application's performance and memory footprint. Some data structures share a common interface and enjoy multiple implementations. We call them Darwinian Data Structures (DDS), since we can subject their implementations to survival of the fittest. We introduce ARTEMIS a multi-objective, cloud-based search-based optimisation framework that automatically finds optimal, tuned DDS modulo a test suite, then changes an application to use that DDS. ARTEMIS achieves substantial performance improvements for *every* project in 5 Java projects from DaCapo benchmark, 8 popular projects and 30 uniformly sampled projects from GitHub. For execution time, CPU usage, and memory consumption, ARTEMIS finds at least one solution that improves *all* measures for 86% (37/43) of the projects. The median improvement across the best solutions is 4.8%, 10.1%, 5.1% for runtime, memory and CPU usage.

These aggregate results understate ARTEMIS's potential impact. Some of the benchmarks it improves are libraries or utility functions. Two examples are gson, a ubiquitous Java serialization framework, and xalan, Apache's XML transformation tool. ARTEMIS improves gson by 16.5%, 1% and 2.2% for memory, runtime, and CPU; ARTEMIS improves xalan's memory consumption by 23.5%. *Every* client of these projects will benefit from these performance improvements.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**;

KEYWORDS

Search-based Software Engineering, Genetic Improvement, Software Analysis and Optimisation, Data Structure Optimisation

> *"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical 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 [24]

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. Consider three examples. Selecting an implementation that creates unnecessary temporary objects for the program's workload [46]. Selecting a combination of Scala data structures that scaled better, reducing execution time from 45 to 1.5 minutes [36]. Avoiding the use of poor implementation, such as those in the Oracle bug database that leak memory [48].

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" [18]. In short, optimisation is expensive and its 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. We introduce *Darwinian data structures*, distinct data structures that are interchangeable because they share an abstract data type and can be tuned. The Darwinian data structure optimisation problem is the problem of finding an optimal implementation and tuning for a Darwinian data structure used in an input program.

We aim to help developers perform optimisation cheaply, focusing solving the data structure optimisation problem. We present Artemis, a cloud-based optimisation framework that identifies *Darwinian data structures* and, given a test suite, *automatically* searches for optimal combinations of implementations and parameters for them. Artemis is language-agnostic; we have instantiated it for Java and C++, and present optimisation results for both languages (Section 5). Artemis' search is multi-objective, seeking to simultaneously improve a program's execution time, memory usage, and CPU usage while passing all the test suites. Artemis scales to large code bases because is uses a Genetic algorithm on those regions of its search space with the most solutions (Section 4.4). Artemis is the first technique to apply multi-objective optimisation to the Darwinian data structure selection and tuning problem.

Artemis promises to change the economics of data structure optimisation. Given a set of Darwinian data structures, Artemis can search for optimal solutions in the background on the cloud, freeing developers to focus on new features. Artemis makes economical small optimizations, such as a few percent, that would not pay for the developer time spent realizing them. And sometimes, of course, Artemis, by virtue of being used, will find unexpectedly big performance gains.

Artemis is a source-to-source transformer. When Artemis finds a solution, the program variants it produces 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 [7]. To ease inspection, Artemis generates a diff for each changes it makes. Developers can inspect these diffs and decide which to keep and which to reject.

We report results on 8 popular diverse GitHub projects, on DaCapo benchmark which was constructed to be representative, and a corpus of 30 GitHub projects, filtered to meet Artemis's constraints and sampled uniformly at random. In this study, Artemis achieves substantial performance improvements for all 43 projects in its corpus. In terms of execution time, CPU usage, and memory consumption, Artemis finds at least one solution for 37 out of 43 projects that improves *all* measures. Across all produced optimal solutions, the median improvement for execution time is 4.8%, memory consumption 10.1% and CPU usage 5.1%. This result is for various corpora, but it is highly likely to generalise to arbitrary programs because of the care we took to build a representative corpus (Section 5.1).

These aggregate results understate Artemis's potential impact. Some of our benchmarks are libraries or utilities. All of their clients will enjoy any improvements Artemis finds for them. Three examples are the Apache project's powerful XSLT processor `xalan`, `Google-http-java-client`, the unbiquitous Java library for accessing web resources, and Google's in-memory file system `Jimfs`. Section 5 shows that Artemis improved `xalan`'s memory consumption by 23.5%, while leaving its execution time unchanged; Artemis improved `Google-http-java-client`'s execution time by 46% and its CPU usage by 39.6%; finally, Artemis improved `Jimfs`'s execution time by 14.2% and its CPU usage by 10.7%, while leaving its memory consumption unchanged.

Our principal contributions follow:

```
1  <T> List<T> getAsList(T value) {
2    if (value == null)
3      return null;
4    List<T> result = new ArrayList<T>();
5    result.add(value);
6    return result;
7  }
```
Listing 1: A function from http-java-client.

- We formalise the Darwinian data structure selection and optimisation problem $\mathbf{DS^2}$ (Section 3).
- We implement Artemis, a multi-language optimisation framework that automatically discovers and optimises sub-optimal data structures and their parameters.
- We show the generalizability and effectiveness of Artemis by conducting a large empirical study on a corpus comprising 8 popular GitHub project, 5 projects from the DaCapo benchmark, and 30 Github projects, filtered then sampled uniformly. For all 43 subjects, Artemis find variants that outperforms the original for all three objectives.
- We provide Artemis as a service, along with its code and evaluation artifacts at http://darwinianoptimiser.com.

2 MOTIVATING EXAMPLE

Listing 1 contains a code snippet from `google-http-java-client`[1], a popular Java library for accessing efficiently resources on the web. In the Listing 1, `getAsList` packages HTTP headers and is invoked frequently from other methods because they use it every time they construct an HTTP request. Thus, its functionality is important for the performance of `google-http-java-client`.

Listing 1 uses `ArrayList` to instantiate the `result` variable. However, other `List` implementations share the same functionality but different non-functional properties. Thus, replacing `ArrayList` with other `List` implementations may affect the performance of the program. Considering the variant created when replacing `ArrayList` (Listing 1, line 4) with `LinkedList`, when we compare it with the original program against the same test set for 30 runs (Section 4), we see that the `google-http-java-client` achieves a median 46%, with 95% Confidence Interval [45.6%, 46.3%] improvement in execution time (Section 5).

Artemis, our optimization framework, automatically discovers underperforming data structures and replaces them with better choices using search-based techniques (Section 4.4). First, it *automatically* creates a store of data structures from the language's Collection API library (Section 4.1). Then, Artemis traverses the program's AST to identify which of those data structures are used and exposes them as parameters to the Artemis's OPTIMIZER (Section 4.4) by transforming line 4 into

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

where D is the tag that refers to the exposed parameter associated with the defined data structure type (Section 4).

Listing 1 does not specify the initial capacity size of the `ArrayList`, so the default size 10 was used. If the instantiated `List` object contains less than 10 items, the default capacity can result in memory

[1]https://github.com/google/google-http-java-client

bloat. If the `List` object contains more than 10 items, the default capacity can slow the execution time; more memory realllocation operations will happen. Therefore, an appropriate value must be chosen to find a good tradeoff between memory and execution time.

ARTEMIS automatically exposes such arguments as tunable parameters, then adjusts them to improve the runtime performance of the program. For instance, ARTEMIS changes line 4 to the code below:

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

where S refers to the exposed parameter associated with the amount of pre-allocated memory.

3 DARWINIAN DATA STRUCTURE SELECTION AND TUNING

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

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

A *data structure* concretely implements an ADT. For the corpus of programs C and the ADT a, the data structure extraction function $dse(a, C)$ returns all data structures that implement a in C. This function is integral to the definition that follows.

DEFINITION 2 (DARWINIAN DATA STRUCTURE). *When* $\exists d_0, d_1 \in dse(a, C) \land d_0 \neq d_1 \land d_0$ *and* d_1 *are observationally equivalent modulo* a, d_0 *and* d_1 *are* Darwinian data structures.

In words, Darwinian data structures are *darwinian* in the sense that they can be replaced to produce program mutants whose fitness we can evaluate. The ADT a has Darwinian data structures when it has more than one data structure that are equivalent over the operations the ADT defines. In Java, `List` is an ADT and `ArrayList`, which implements it, is a data structure. `LinkedList` also implements `List`, so both `ArrayList` and `LinkedList` are Darwinian. For the ADT a and the corpus C, Darwinian data structures are interchangeable. Thus, we can search the variants of $P \in C$ formed by substituting one Darwinian data structure for another to improve P's nonfunctional properties, like execution time, memory consumption or CPU usage.

Just as we needed a function to extract an ADT's data structures from a corpus for Definition 2, we need a function that returns the ADT that a data structure implements: when $d = dse(a, C)$, let $adte(d, C) = a$. Let Γ_D bind fully scope-qualified declarations of names to Darwinian data structures in C. We use Γ_D when creating variants of a program via substitution. We are interested not just searching the space of Darwinian data structures, but also tuning them via their constructor parameters. To this end, we assume without loss of generality that a defines a single constructor c and we let $n.c(x)$ denote calling identifier n's constructor c with parameters $x : \tau$.

To create a variant of $P \in C$ that differs from P only in its k bindings of names to Darwinian data structures or their constructor initialization parameter, we define

$$\phi(P, (n, d_i)^k, d_j^k, x_j) =$$
$$\begin{cases} P[(n.c(x_i))^k / (n.c(x_j))^k], & \text{if } \exists d_i, d_j \text{ s.t. } adte(d_i) \neq adte(d_j) \\ P[(n, d_i)^k / (n, d_j)^k][(n.c(x_i))^k / (n.c(x_j))^k], & \text{otherwise} \end{cases}$$

DEFINITION 3 (DARWINIAN DATA STRUCTURE SELECTION AND TUNING). *For the real-valued fitness function* f *over the corpus* C, *the Darwinian data structure and tuning problem is*

$$\underset{(n_i, d_i)^k \in \Gamma_D^k, d_j^k \in adte(d_i, C)^k, x_j \in \tau}{\arg\max} f(\phi(P, (n_i, d_i), d_j, x_j))$$

This vector-based definition simultaneously considers all possible rebinding of names to Darwinian data structures in P; it is also cumbersome, compared to its point-substitution analogue. We could not, however, simply present a scalar definition and then quantify over all potential DDSS substitutions, as doing so would not surface synergies and antagonisms among the substitutions.

4 ARTEMIS

The ARTEMIS's optimisation framework solves the Darwinian Data Structure Selection problem. Figure 1 illustrates the architecture with its 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 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 genetic search algorithm (NSGA-II [13]) to tune the parameters [25–27, 44, 45] 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 and CPU usage.

ARTEMIS relies on testing to define a performance search space and to preserve semantics. ARTEMIS therefore can only be applied to programs with a test suite. Ideally, this test suite would comprise both a regression test suite with high code coverage for maintaining the correctness of the program and a performance test suite to simulate the real-life behaviour of the program and ensure that all of the common features are covered [4]. Even though performance test suites are a more appropriate and logical choice for evaluating the non-functional properties of the program, most real world programs in GitHub do not provide such performance test suite. For this reason, we use the regression test suites to evaluate the non-functional properties of the GitHub projects of this study whenever a performance test suite is not available.

4.1 Darwinian Data Structure Store

ARTEMIS needs a Darwinian data structure store (DDSS) from which to choose when creating variants. Let A be a set of ADTs known to be Darwinian. A developer can augment this set; Figure 2 shows those that ARTEMIS knows by default. For our corpus C of Java benchmarks augmented with JDK libraries over A,

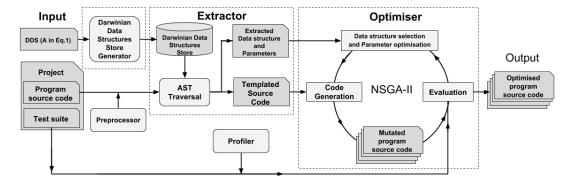$$DDSS = \bigcup_{a \in A} dse(a, C). \tag{1}$$
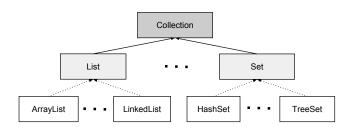
Figure 1: System Architecture of ARTEMIS.

Figure 2: DDS in the Java Collections API.

Table 1: Data structure groups.

Abstract Data Type	Implementation
List	ArrayList, LinkedList
Map	HashMap, LinkedHashMap
Set	HashSet, LinkedHashSet
Concurrent List	Vector, CopyOnWriteArrayList
Concurrent Deque	ConcurrentLinkedDeque,
	LinkedBlockingDeque
Thread Safe Queue	ArrayBlockingQueue, SynchronousQueue,
	LinkedBlockingQueue, DelayQueue,
	ConcurrentLinkedQueue, LinkedTransferQueue

To build the default DDSS for Java, ARTEMIS extracts and traverses each project's class hierarchy, similar to the one illustrated in Figure 2. This hierarchy shows potential Darwinian data structures of a specific interface. When this traversal finishes, ARTEMIS extracts all the implementations of a particular Darwinian data structure; *e.g.,* List, ArrayList, LinkedList. ARTEMIS considers these implementations mutually replaceable. For Java, a default DDSS is provided by ARTEMIS, which the developer can edit. For other languages, the DDSS can be provided manually by the user and this step can be skipped. The OPTIMISER, described next, uses the store during its search.

The developer can also extend the store with custom user-supplied implementations or with implementations from other third-party libraries such as Google Guava Collections[2], fastutil[3] and Apache Commons Collections[4].

4.2 Discovering Darwinian Data Structures

The EXTRACTOR takes as input the program P's source code, identifies Darwinian data structures in P modulo its store (Section 4.1), and outputs a scope-qualified list of names of Darwinian data structures and their constructor parameters (*Extracted Data Structures and Parameters* in Figure 1). For all $a \in$ DDSS, EXTRACTOR's output realises $dse(a, P)$ (Section 3). To mark potential substitutions to the transformer, the EXTRACTOR outputs a templated version of the code which replaces the data structure with data structure type identifiers (*Templated Source Code* in Figure 1).

---

[2]https://github.com/google/guava
[3]https://github.com/vigna/fastutil
[4]https://github.com/apache/commons-collections

To find DARWINIAN data structures, the EXTRACTOR builds an Abstract Syntax Tree (AST) from its 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

When implementing ARTEMIS, we encountered coding practices that vastly increase the search space. Many turn out to be known bad practices [43]. Consider Listing 2. In lines 2 and 8, we see two LinkedList variables that the Extractor marks DARWINIAN and candidates for replacement by their equivalent ArrayList implementation. In these lines, user is violating the precept to "program to the interface", here List, but is, instead, declaring the variable to have the concrete, data structure not ADT, type LinkedList. This bad practice [43] adds dependencies to the code, limiting code reuse. They are especially problematic for ARTEMIS, because they force ARTEMIS to apply multiple transformations to replace and optimise the data structure. Further, func3 takes a LinkedList as a parameter, not List, despite the fact that it only calls the get method defined by List on this parameter. This instance of violating the "program to the interface" precept triggers a compilation error if ARTEMIS naïvely changes func1's type. ARTEMIS transforms

```
1   void func1(){
2     LinkedList<T> v;
3     v = new LinkedList<>();
4     v.add(new T());
5     int value = func3(v);
6   }
7   void func2(LinkedList<T> v){
8     LinkedList<T> v1 = new LinkedList<>();
9     int value = func3(v1);
10  }
11  int func3(LinkedList<T> v){
12    T t = v.get(0);
13    return 2*t.value;
14  }
```

Listing 2: Code to illustrate bad practices.

the code to reduce the OPTIMISER's search space and handle these bad practices. ARTEMIS supports thee transformations - parserless, supertype, and profiler.

The *parserless* mode changes greadily each appearance of a Darwinian implementation. When optimising `List`, it exhaustively tries every implementation of `List` for every `List` variable. It is parserless, since it needs only a regular expression to identify rewritings. This makes it simple, easily portable to other languages, and fast, so it is ARTEMIS' default. However, it generates invalid programs and a large search space.

ARTEMIS' *supertype* transformation converts the type of a Darwinian implementation to that of their Darwinian ADT, for example `LinkedList<T>` → `List<T>` on lines, 2,7,8 and 11. For Listing 2, this tranformation exposes only two DDS to the OPTIMISER and produces only syntactically valid code. To implement this transformation, ARTEMIS invokes Eclipse's re-factoring functionality via its API, then validates the result. ARTEMIS aims to be language-agnostic without any additional dependencies on language specific tools. For this case, ARTEMIS auto performs this transformation 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.

"All data structures are equal, but some data structures are more equal than others" [5]; some DDS affect a program's performance more than others, as when one stores only a few, rarely accessed items. To rank DDS, ARTEMIS *profiles* its input program to identify costly methods. The EXTRACTOR uses this info to identify the subset of a program's DDS worth considering for optimisation. ARTEMIS' instrumentation is particularly important for large programs.

4.4 Search Based Parameter Tuning

The OPTIMISER searches a combination of data structures that improves the performance of the initial program while keeps the original functionality. Practically, we can represent all those data structures as parameters that can be tuned using Search Based Software Engineering approaches [19]. Because of the nature of the

various conflicting performance objectives, the problem we faced here requires a multi-objective optimisation approach to search the (near) optimal solutions.

An array of integers is used to represent the tuning parameters. Each parameter refers either to a Darwinian data structure or to the initial size of that data structure. If the parameter refers to a data structure, its value represents the index in the list of Darwinian data structures. The OPTIMISER keeps additional mapping information to distinguish the types of the parameters. For each generation, the NSGA-II applies 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, ARTEMIS applies standard non-dominated selection to form the next generation. ARTEMIS repeats this process until the solutions in a generation converge. At this point, ARTEMIS returns all non-dominated solutions in the final population.

Search Space size: We used GA because the search space is huge. Let D be the definitions of darwinian data structures in program P. Let I be the number of implementations for a particular $d \in D$. The size of the search space is:

$$\prod_{d \in D} I(d) * |dom(d.c)|, \text{ where } d.c \text{ is } d\text{'s constructor.} \quad (2)$$

4.5 Deployability

ARTEMIS provides optimisation as a cloud service. To use the service, developers only need to provide the source code of their project in a Maven build format and a performance test suite invoked by `mvn test`. ARTEMIS returns the optimised source code and a performance report. ARTEMIS exposes a RESTful API that developers can use to edit the default store of Darwinian data structures. The API also allows developers to select other Search Based algorithms; the OPTIMISER uses NSGA-II by default. To use our tool from the command line, a simple command is used:

```
./artemis input−program−src
```

where this command defaults to ARTEMIS's built in DDSSG. ARTEMIS writes the source of an optimized variant of its input for each measure. ARTEMIS also supports optional parameters to customise its processing.

5 EVALUATION

To demonstrate the performance improvements that ARTEMIS automatically achieves and its broad applicability, we applied it to three corpora: 8 popular GitHub projects, 5 projects from the Dacapo Benchmark, and 30 projects, filtered to meet ARTEMIS's requirements, then sampled uniformly at random from Github. To show also that ARTEMISis language-agnostic, we applied it to optimise Guetzli[6] (Section 5.3), a JPEG encoder written in C++.

5.1 Corpus

ARTEMIS requires projects with capable build systems and an extensive test suites. These two requirements entail that ARTEMIS be able to build and run the project against its test suite. ARTEMIS is

---

[5]Adapted from "Animal Farm" by George Orwell

language-agnostic but is currently only instantiated for Java and C++, so it requires Java or C++ programs.

Our first corpus comprises eight popular GitHub projects. We selected these eight to have good test suites and be diverse. We defined popular to be projects that received at least 200 stars on GitHub. We deemed a test suite to be good if its line coverage met or exceeded 70%. This corpus contains projects, usually well-written, optimised and peer code-reviewed by experienced developers. We applied ARTEMIS on those projects to investigate whether it can provide a better combination of data structures than those selected by experienced human developers.

This first corpus might not be representative, precisely because of the popularity of its benchmarks. To address this threat to validity, we turned to the DaCapo benchmarks [5]. The authors of DaCapo built it, from the ground up, to be representative. The goal was to provide the research community with realistic, large scale Java benchmarks that contain a good methodology for Java evaluation. Dacapo contains 14 open source, client-side Java benchmarks (version 9.12) and they come with built-in extensive evaluation. Each benchmark provides accurate measurements for execution time and memory consumption. DaCapo first appeared in 2006 to work with Java v.1.5 and has not been further updated to work with newer versions of Java. For this reason, we faced difficulties in compiling all the benchmarks and the total number of benchmarks were reduced to 5 out of 14. In this corpus we use the following five: `fop`, `avrora`, `xalan`, `pmd` and `sunflow` (Figure 4).

Because of its age and the fact that we are only using subset of it, our DaCapo benchmark may not be representative. To counter this threat, we uniformly sampled projects from GitHub. We discarded those that did not meet ARTEMIS's constraints, like being equipped with a build system, until we collected 30 projects. Those projects are diverse, both in domain and size. The selected projects include static analysers, testing frameworks, web clients, and graph processing applications. Their sizes vary from 576 to 94K lines of code with a median of 14881. Their popularity varies from 0 to 5642 stars with a median of 52 stars per project. The median number of tests is 170 and median line coverage ratio is 72%.

Collectively, we systematically built these corpora to be representative in order to demontrate the general applicably of the ARTEMIS' optimization framework. The full list of the programs used in this experimental study are available online[7] in the project's website.

5.2 Experimental Setup

Experiments were conducted using Microsoft Azure[TM] D4-v2 machines with one Intel E5-2673v3 CPU featuring 8 cores and 14GB of DRAM and built with Oracle JDK 1.8.0 and Ubuntu 16.04.4 LTS.

Performance measurements may lead to incorrect results if not handled carefully [1]. Thus, a statistical rigorous performance evaluation is required [16, 23, 28]. To mitigate instability and incorrect results, we differentiate VM start-up and steady-state. We ran our experiments in a fresh Azure VM that contained only the JVM and the subject. We use JUnit, which runs an entire test suite in a single JVM. We manually identified and dropped startup runs, then we spot-checked the results to confirm that the rest of the runs achieved a steady state and were exhibiting low variance. All

of the means and medians we reported fall within the computed interval with 95% confidence. 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 more than 30 simulations. Also we use Mann Whitney U test [15] 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[8] because it directly handles JDK statistics and provides elegant API. We extended JConsole to monitor only those system processes belonging to the test suite. We use Maven Surefire plugin[9] to measure the test suite's execution time because it reports only the execution time of each individual test, excluding the measurement overhead that other Maven plugins may introduce.

For the OPTIMISER, we chose an initial population size of 30 and a maximum number of 900 function evaluations. We used 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). We determined these settings from calibration trials to ensure the maturity of the results. Since NSGA-II is stochastic, we ran each experiment 30 times to obtain statistical significant results.

5.3 Research Questions and Results Analysis

ARTEMIS aims to improve all objectives at the same time. Therefore the first research question we would like to answer is:

RQ1: *What proportion of programs does ARTEMIS improve?*

To answer RQ1, we applied ARTEMIS to our corpus. We inspected the generated optimal solutions from 30 runs of each subject by examining the dominate relation between the optimal and inital solutions regarding the optimisation objectives. We introduce the terms *strictly dominate relation* and *non-dominated relation* to describe the relation. Defined by Zitzler et al. [49], a solution *strictly dominates* another solution if it outperforms the latter in all measures. A solution is *non-dominated* with another solution if both outperform the other in at least one of the measures.

For DaCapo, ARTEMIS found at least one strictly dominant solution for 4 out of 5 projects; it found no such solution for `sunflow`. It found 1072 solutions, from which 3% are strictly dominant (median is 5.5 solutions per project) and 64% are non-dominated (median is 18 solutions per project).

For the popular Github projects, ARTEMIS found at least one strictly dominant solution for all 8 projects. The total number of solutions found is 10218 and 16% of them are strictly dominant (median is 50 solutions per project) and 59% are non-dominated (median is 749.5 solutions per project).

For the sampled Github projects, ARTEMIS found a strictly dominant solution for 25 out of 30 projects, but found no solution for projects `rubix-verifier`, `epubcheck`, `d-worker`, `telegrambots` and `fqueue`. It found 27503 of which 10% of them are strictly dominant (median is 24 solutions per project) and 66% are non dominant

---

(median is 125 solutions per project). With these results, we answer *RQ*1 affirmatively:

> **Finding1:** ARTEMIS finds optimised variants that outperform the original program in at least one measure for *all* programs in our representative corpus.

This finding understates ARTEMIS's impact. Not only did it improve at least one measure for *all programs*, ARTEMIS found solutions that improve *all measures* for 88% of the programs.

Having found that ARTEMIS finds performance improvements, we ask "How good are these improvements" with:
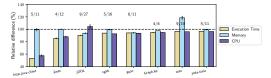
RQ2: *What is the average improvement that ARTEMIS provides for each program?*

Though ARTEMIS aims to improve all candidate's measures, it cannot achieve that if improvements are antagonistic. In some domains, it is more important to significantly improve 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 many solutions on the Pareto-front and at least one of them will improve each measure significantly.
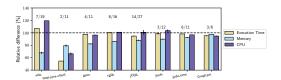
We answer *RQ*2 quantitatively. We report the maximum improvement (median value with 95% confidence interval) for execution time, memory and CPU usage for each subject of the three corpora. We use bar charts with error bars to plot the three measures for each program. In Y axis, we represent the percentage of improvement for each measure. A value less than 100% represents an improvement and a value greater than 100% means degradation; *e.g.,* 70% memory consumption implies that the solution consumes 70% of the memory used in the input program.

Selected popular GitHub programs. Figure 3a presents the three measures of the solutions when the execution time is minimised, for each program from the popular GitHub programs. We observe that ARTEMIS improves the execution time of *every* program. google-http-java-client's execution time was improved the most; its execution time was reduced by M=46%, 95% CI [45.6%, 46.3%]. We also notice that this execution time improvement did not affect negatively the other measures, but instead the CPU usage was reduced by M=41.6%, 95% CI [39.6%, 43.6%] and memory consumption remained almost the same. The other interesting program to notice from this graph is solo, a blogging system written in Java; its execution time improved slightly by 2% but its memory consumption increased by 20.2%. Finally, for this set of solutions, the median execution time improvement is 14.13%, whilst memory consumption slightly increased by 1.99% and CPU usage decreased by 3.79%. For those programs, ARTEMIS extracted a median of 12 data structures and the optimal solutions had a median of 4 data structures changes from the original versions of the program.
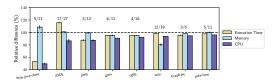
Figure 3b shows the solutions optimised for memory consumption. We notice that ARTEMIS improves the memory consumption for all programs, with a median value of 14%. The execution time was improved by a median value of 2.8% for these solutions, while the median value of CPU usage is slightly increased by 0.4%. We notice that solo has the best improvement by M=31.1%, 95% CI

(a) Best execution time of popular GitHub programs. The median value is 93.3%, mean is 86.4%. Median number of DDS is 12 and mean is 14.6. Median number of DDS changes is 4 and mean is 5.

(b) Best memory consumption of popular GitHub programs. The median value is 86% and mean is 84%. Median number of DDS is 12 and mean is 14.6. Median number of DDS changes is 4 and mean is 5.85.

(c) Best CPU usage of popular GitHub programs. The median value is 90.3% and mean is 84.6%. Median number of DDS is 12 and mean is 14.6. Median number of DDS changes is 5 and mean is 7.42.
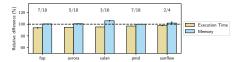
Figure 3: Answers *RQ2*. Description.

[29.3%, 33%], but with an increase of M=8.7%, 95% CI [8.5%, 8.9%] in execution time and M=21.3%, 95% CI [20.6%, 22%] in CPU usage. Graphjet, a real-time graph processing library, has the minimum improvement of M=0.9%, 95% CI [0.6%, 1.1%]. The optimal solutions had a median of 4 data structures changes per solution.
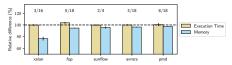
Figure 3c presents solutions optimised for CPU usage. The median CPU usage improvement is 9.7%. The median value of execution time improved by 5.2% and the median value of memory consumption improved by 2.3%. The program with the most significant improvement in CPU is http-java-client with M=49.7%, 95% CI [48%, 51.4%], but with a decrease in memory of M=9.8%, 95% CI [7.5%, 12.9%]. The optimal solutions make a median of 5 data structures changes to the original versions of the program.

DaCapo. Figure 4 presents all solutions optimised for execution time and memory consumption for the DaCapo benchmark. We used only two measures for the DaCapo benchmark as those were the ones built in the benchmark suite. We chose not to extend or edit the profiling method of DaCapo, to avoid the risk of affecting the validity of its existing, well tested profiling process.

ARTEMIS found solutions that improve the execution time for every program without affecting significantly the memory consumption, except project xalan which had improvement (M=4.8%, 95% CI [4.6%, 5.7%] in execution time but with an increase (5.8%, 95% CI [3.5%, 7%]) in memory consumption. All solutions for optimised memory consumption did not affect execution time, except for a slight increase for program fop. Finally, for this set of solutions, the median percentage of execution time improvement is 4.8%, and 4.6% for memory consumption. For this set of programs,

(a) Best execution time of the Dacapo benchmark. The median value is 95.20% and mean is 95.6%. Median number of DDS is 18 and mean is 14.8. Median number of DDS changes is 5 and mean is 4.8.

(b) Best memory consumption of the Dacapo benchmark. The median value is 95.7% and mean is 92.1% . Median number of DDS is 18 and mean is 14.8. Median number of DDS changes is 3 and mean is 4.2.

Figure 4: Answers *RQ2*. Description.

ARTEMIS extracted a median of 18 data structures per program, and the optimal solutions had a median of 5 data structures changes for the execution time optimised solutions and 4 for the memory optimised solutions.

Sampled GitHub programs. Figure 5 presents all solutions optimised for execution time, memory consumption and CPU usage for the sampled GitHub programs. As with the previous corpora, ARTEMIS found solutions that improved each measure significantly. ARTEMIS improves the median value of execution time across all projects by 4.6%, memory consumption by 11.4% and CPU usage by 4.6%.

ARTEMIS found solutions with antagonistic improvement for projects jafka and documents4j. ARTEMIS found a solution that improves the execution time of jafka, a distributed publish-subscribe messaging system, by M=12%, 95% CI [11.2, 13.6%], but also increases its memory consumption by M=23.6%, 95% CI [21.4, 25.7%]. It also found a solution that improves the memory consumption of documents4j (M=38%, 95% CI [38%, 41%]) but introduced extra CPU usage M=26.1%, 95% CI [24.2, 28%]. A median of 9.5 data structures were extracted and the optimal solutions had a median of 5 data structures changes from the original versions of the program.

Observing again the numbers across the three corpora, we can say that they are quite consistent, showing that ARTEMIS finds optimal solutions that improve significantly the different optimisation measures. We also see that the number of Darwinian Data structures extracted (between 9.5 and 18) and the optimal solutions DDS changes (between 4 and 5) are quite similar for the three corpora.

Analysing all results from the 3 corpora we conclude the discussion of RQ2 with:

> **Finding2:** ARTEMIS improves the median across all programs in our corpus by 4.8% execution time, 10.2% memory consumption, and 5.1% CPU usage.

RQ3: *Which Darwinian data structures does ARTEMIS find and tune?*

We ask this question to understand which changes ARTEMIS makes to a program. Table 2 contains the transformations ARTEMIS

Table 2: DDS changes for optimal solutions across all measures.

Tranformation	Time	Memory	CPU
HashMap -> LinkedHashMap	60	53	57
LinkedList -> ArrayList	16	13	18
HashSet -> LinkedHashSet	22	21	21
LinkedBlockingQueue -> LinkedTransferQueue	1	2	2
ArrayList -> LinkedList	91	86	87
LinkedHashSet -> HashSet	7	8	5
Vector -> CopyOnWriteArrayList	1	0	2
LinkedHashMap -> HashMap	17	23	19

applied across all optimal solutions. We see that the most common transformation for all measures is replacing `ArrayList` with `LinkedList`, it appears 91, 86 and 87 times respectevely across all measures. This transformation indicates that most developers prefer to use `ArrayList` in their code, which in general is considered to be faster, neglecting use cases in which `LinkedList` performs better; *e.g.,* when the program has many list insertion or removal operations. Except `HashMap` to `LinkedHashMap`, the other transformations happen relatively rare in the optimal solutions. Last, the median number of lines Artemis changes is 5.

> **Finding3:** ARTEMIS extracted a median of 12 Darwinian data structures from each program and the optimal solutions had a median of 5 data structure changes from the original versions of the program.
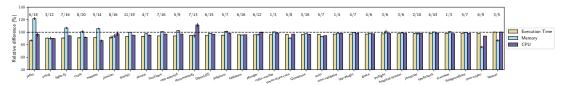
RQ4: *What is the cost of using ARTEMIS?*

In order for ARTEMIS to be practical and useful in real-world situations, it is important to understand the cost of using it. The aforementioned experimental studies reveal that, even for the popular programs, the existing selection of the data structure and the setting of its parameters may be sub-optimal. Therefore, optimising the data structures and their parameters can still provide significant improvement on 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, we used a Microsoft Azure[TM] D4-v2 machine, which costs £0.41 per hour at a standard Pay-As-You-Go rate[10], to conduct all experiments.
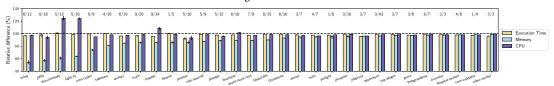
The experiments show that an optimisation process takes 3.05 hours on average for all studied subjects. The program `GraphJet` and `jimfs` are the most and the least time-consuming programs respectively, with 19.16 hours and 3.12 minutes optimisation time. Accordingly, the average cost of applying ARTEMIS for the subjects studied is £1.25, with a range from £0.02 to £7.86. The experimental results show that overall cost of using ARTEMIS is negligible compared to a human software engineer, with the assumption that a competent software engineer can find those optimisation in a reasonable time.

ARTEMIS transforms the selection of data structure and sets parameters by rewriting source code, thereby allowing human developers to easily investigate its changes and gain insight about the usage of data structures and the characteristics of the program.
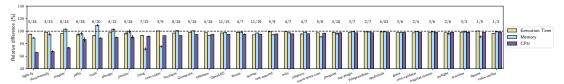
---

[10]https://azure.microsoft.com/en-gb/pricing/

(a) Best execution time of uniformly selected GitHub programs. The median value is 95.4% **and mean is** 94.7%. **Median number of** DDS **is** 9.5 **and mean is** 11.6. **Median number of** DDS **changes is** 5 **and mean is** 4.8.

(b) Best memory consumption of the uniformly selected GitHub programs. The median value is 89.1% **and mean is** 86.8%. **Median number of** DDS **is** 9.5 **and mean is** 11.6. **Median number of** DDS **changes is** 5 **and mean is** 4.6.

(c) Best CPU usage of the uniformly selected GitHub programs. The median value is 5.1% **and mean is** 8%. **Median number of** DDS **is** 9.5 **and mean is** 11.6. **Median number of** DDS **changes is** 5 **and mean is** 4.5.

Figure 5: Answers *RQ2*. Description.

Finding4: The cost of using ARTEMIS is negligible, with an average of £1.25 per project, providing engineers with insights about the optimal variants of the project under optimisation.

To show the versatility of the ARTEMIS framework, we ask RQ2, RQ3 and RQ4 over Google guetzli, a very popular JPEG encoder written in C++. We used the STL containers and their operations as Darwinian data structures. More specifically, we considered the push_back and emplace_back as equivalent implementations of the same functionality and exposed those as tunable parameters to ARTEMIS's optimiser. We collected a random sample of images (available online [11]) and used it to construct a performance suite that evaluates the execution time of guetzli.

We answer RQ2 by showing that ARTEMIS found an optimal solution that improves execution time by 7%. We answer RQ3 by showing that ARTEMIS extracted and tuned 25 parameters and found an optimal solution with 11 parameter changes. ARTEMIS spent 1.5 hours (costs £0.62) to find optimal solutions which is between the limits reported in RQ4. Last, we spent approximately 4 days to extend ARTEMIS to support C++, using the *parserless* mode.

6 THREATS TO VALIDITY

Section 5.1 discusses the steps we took to address the threats to the external validity of the results we present here. In short, we built three subcorpora, each more representative than the last, for a total of 43 programs, diverse in size and domain. The biggest threat to the internal validity of our work is the difficulty of taking accurate performance measurements of applications running on VM, like the JVM. Section 5.2 details the steps, drawn from best

---

practice, we took to address this threat. In essence, we conducted calibration experiments to adjust the parameters such that the algorithm converges quickly 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 tuned JConsole to further improve the precision of the measurements by maximising its sampling frequency such that it does not miss any measurements while minimising the CPU overhead. To cater for the stochastic nature of ARTEMIS and to provide the statistic power for the results, we ran each experiment 30 times and manually checked that experiments had a steady state and exhibited low variance.

7 RELATED WORK

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

Search-Based Software Engineering Previous work applies Genetic Programming [21, 29, 30, 35, 37, 39] to either improve the functionality (bug fixing) [11, 17] or non-functional properties of a program [8, 21, 25, 35, 38, 44]. 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, many of these approaches rely on the Plastic Surgery Hypothesis [2], which assumes that the solutions exist in the code base. ARTEMIS, on the other hand, does not rely on this hypothesis. ARTEMIS can harvest Darwinian data structures both from the program, but also from external code repositiories; further, ARTEMIS relies on a set of transformation rules that it can automatically exhaustively extract from library code or documentation.

Wu et al. [45] introduced a mutation-based method to expose "deep" parameters, similar to those we optimise 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 ARTEMIS, their approach did not optimise data structure selection, which can sometimes be more rewarding than just tuning the parameters. Moreover, they applied their approach to a memory management library to benefit that library's clients. The extent of improvement usually depends on how much a program relies on that library. In contrast, ARTEMIS directly applies to the source code of the program, making no assumptions about which libraries the program uses, affording ARTEMIS much wider applicability.

Data Structure Optimisation and Bloat   A body of work [3, 14, 32–34, 42, 47] has attempted to identify bloat arising from data structures. In 2009, Shacham et al. [40, 41] introduced a semantic profiler that provides online collection-usage semantics for Java programs. They instrumented Java Virtual Machine (JVM) to gather the usage statistics of collection data structures. Using heuristics, they suggest a potentially better choice for a data structure for a program.

Though developers can add heuristics, if they lack sufficient knowledge about the data structures, they may bias the heuristics and jeopardise the effectiveness of the approach. ARTEMIS directly uses the performance of a data structure profiled against a set of performance tests to determine the optimal choices of data structures. Therefore, ARTEMIS does not depend on expert human knowledge about the internal implementation and performance differences of data structures to formulate heuristics. Instead. ARTEMIS relies on carefully-chosen performance tests to minimse bias. Furthermore, ARTEMIS directly modifies the program instead of providing hints, thus users can use the fine-tuned program ARTEMIS generates without any additional manual adjustment.

Other frameworks provide users with manually or automatically generated selection heuristics to improve the data structure selection process. JitDS [12] exploits declarative specifications embedded by experts in data structures to adapt them. CollectionSwitch [9] uses data and user-defined performance rules to select other data structure variants. Brainy [22] provides users with machine learning cost models that guide the selection of data structures. ARTEMIS does not require expert annotations, user-defined rules or any machine learning knowledge. Storage strategies [6] changes VMs to optimize their performance on collections that contain a single primitive type; Artemis rewrites source code and handles user-defined types and does not make VM modifications.

In 2014, Manotas et al. [31] 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. Conceptually ARTEMIS extends this approach to optimise both the data structures and their initialization parameters. ARTEMIS also extends 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-heuristic search.

Furthermore, ARTEMIS directly transforms the source code of the programs whilst *SEEDS* transforms the bytecode, so ARTEMIS provides developers more intuitive information about what was changed and teaches them to use more efficient data structures. Moreover, ARTEMIS can be more easily applied to other languages as it does not depend on language specific static analysers and refactoring tools such as WALA [20] and Eclipse IDE's refactoring tools. In order to support another language we just need the grammar of that language and to implement a visitor that extracts a program's Darwinian data structures. We note that ANTLR, which ARTEMIS uses, currently provides many available grammar languages [12].

Apart from the novelties mentioned above, this is the largest empirical study to our knowledge compared to similar work. In the studies mentioned above, only 4 to 7 subjects were included in the experiments. Our study included the DaCapo benchmark, 30 sampled Github subjects and 8 well-written popular subjects to show the effectiveness of ARTEMIS, 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 multi-language search-based framework that automatically selects and optimises Darwinian data structures and their arguments in a given program. ARTEMIS is language agnostic, meaning it can be easily adapted to any programming language; extending ARTEMIS to support C++ took approximately 4 days. Given as input a data structure store with Darwinian implementations, it can automatically detect and optimise them along with any additional parameters to improve the non-functional properties of the given program. In a large empirical study on 5 DaCapo benchmarks, 30 randomly sampled projects and 8 well-written popular Github projects, ARTEMIS found *strong* improvement for all of them. On extreme cases, ARTEMIS found 46% improvement on execution time, 44.9% improvement on memory consumption, and 49.7% improvement on CPU usage. ARTEMIS found such improvements making small changes in the source code; the median number of lines ARTEMIS changes is 5. Thus, ARTEMIS is practical and can be easily used on other projects. 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 optsimising 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.

---

[12] https://github.com/antlr/grammars-v4/

REFERENCES

[1] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *ACM SIGPLAN Notices*, volume 37, pages 111–129. ACM, 2002.

[2] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317, New York, NY, USA, 2014. ACM.

[3] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr. Optimising darwinian data structures on google guava. In *International Symposium on Search Based Software Engineering*, pages 161–167. Springer, 2017.

[4] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 2000.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[6] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 167–182, New York, NY, USA, 2013. ACM.

[7] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[8] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, New York, NY, USA, 2015. ACM.

[9] D. Costa and A. Andrzejak. Collectionswitch: a framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 16–26. ACM, 2018.

[10] N. Dale and H. M. Walker. *Abstract data types: specifications, implementations, and applications.* Jones & Bartlett Learning, 1996.

[11] H. Dan, M. Harman, J. Krinke, L. Li, A. Marginean, and F. Wu. Pidgin crasher: searching for minimised crashing gui event sequences. In *International Symposium on Search Based Software Engineering*, pages 253–258. Springer, 2014.

[12] M. De Wael, S. Marr, J. De Koster, J. B. Sartor, and W. De Meuter. Just-in-time data structures. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 61–75. ACM, 2015.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[14] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 59–70. ACM, 2008.

[15] M. P. Fay and M. A. Proschan. Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys*, 4:1, 2010.

[16] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[17] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13, June 2012.

[18] B. Hardin. Companies with hacking cultures fai. https://blog.bretthard.in/companies-with-hacking-cultures-fail-b8907a69e3d#.ffdkyb1w2, 2016. [Online; accessed 25-February-2017].

[19] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[20] IBM. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page, 2009. [Online; accessed 18-February-2017].

[21] Y. Jia, F. Wu, M. Harman, and J. Krinke. Genetic improvement using higher order mutation. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 803–804, 2015.

[22] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: effective selection of data structures. In *ACM SIGPLAN Notices*, volume 46, pages 86–97. ACM, 2011.

[23] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *ACM SIGPLAN Notices*, volume 48, pages 63–74. ACM, 2013.

[24] D. E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, Dec. 1974.

[25] W. B. Langdon, M. Modat, J. Petke, and M. Harman. Improving 3d medical image registration cuda software with genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 951–958. ACM, 2014.

[26] L. Li. Exact analysis for next release problem. In *Requirements Engineering Conference (RE), 2016 IEEE 24th International*, pages 438–443. IEEE, 2016.

[27] L. Li. *Exact analysis for requirements selection and optimisation.* PhD thesis, UCL (University College London), 2017.

[28] L. Li, M. Harman, E. Letier, and Y. Zhang. Robust next release problem: handling uncertainty during optimization. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1247–1254. ACM, 2014.

[29] L. Li, M. Harman, F. Wu, and Y. Zhang. Sbselector: Search based component selection for budget hardware. In *International Symposium on Search Based Software Engineering*, pages 289–294. Springer, 2015.

[30] L. Li, M. Harman, F. Wu, and Y. Zhang. The value of exact analysis in requirements selection. *IEEE Transactions on Software Engineering, PP (99)*, pages 1–1, 2016.

[31] 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 2014, pages 503–514, New York, NY, USA, 2014. ACM.

[32] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *ACM SIGPLAN Notices*, volume 42, pages 245–260. ACM, 2007.

[33] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming*, pages 429–451. Springer, 2006.

[34] F. Nagel, G. M. Bierman, A. Dragojevic, and S. Viglas. Self-managed collections: Off-heap memory management for scalable query-dominated collections. In *EDBT*, pages 61–71, 2017.

[35] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke. Mutation testing of memory-related operators. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.

[36] 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].

[37] J. Petke, S. Haraldsson, M. Harman, D. White, J. Woodward, et al. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.

[38] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *European Conference on Genetic Programming*, pages 137–149. Springer, 2014.

[39] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza. *A field guide to genetic programming.* Lulu. com, 2008.

[40] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM.

[41] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. *SIGPLAN Not.*, 44(6):408–418, June 2009.

[42] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. *ACM Sigplan Notices*, 43(10):127–142, 2008.

[43] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[44] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke. Memory mutation testing. *Information & Software Technology*, 81:97–111, 2017.

[45] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382. ACM, 2015.

[46] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. *ACM Sigplan Notices*, 44(6):419–430, 2009.

[47] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. *ACM Sigplan Notices*, 45(6):174–186, 2010.

[48] 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*, pages 151–160. IEEE, 2008.

[49] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, April 2003.