View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCL Discovery

# Optimising Darwinian Data Structures on Google Guava

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

Department of Computer Science,
University College London, Malet Place, London, WC1E 6BT, UK.
{m.basios,lingbo.li,fan.wu,l.kanthan,e.barr}@cs.ucl.ac.uk

**Abstract.** Data structure selection and tuning is laborious but can vastly improve application performance and memory footprint. In this paper, we demonstrate how ARTEMIS, a multiobjective, cloud-based optimisation framework can *automatically* find optimal, tuned data structures and how it is used for optimising the Guava library. From the proposed solutions that ARTEMIS found, 27.45% of them improve *all* measures (execution time, CPU usage, and memory consumption). More specifically, ARTEMIS managed to improve the memory consumption of Guava by up 13%, execution time by up to 9%, and 4% CPU usage.

**Keywords:** Search-based software engineering; Genetic improvement; Software analysis and optimisation; Multi-objective optimisation

## 1 Introduction

Under the immense time pressures of industrial software development, developers tend to avoid early-stage optimisations, yet forget to do so later. 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 [11]. For performance, examples include the selection of an implementation that created unnecessary temporary objects for the program's workload [13] or selecting a combination of Scala data structures that scaled better, reducing execution time from 45 to 1.5 minutes [10]; 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 [14].

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" [6] and "Hack until it works" [4]. In short, optimisations are expensive and their benefits unclear for many projects. Developers need automated help.

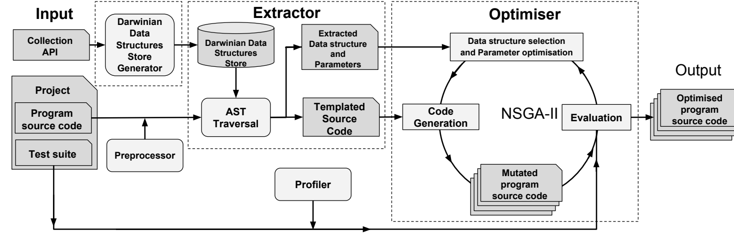


Fig. 1: System Architecture of ARTEMIS.

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 a 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* [9] is on 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.

In this paper, we aim to help developers perform optimisations cheaply, focusing solving the data structure optimisation problem. We present ARTEMIS, a cloud-based language-agnostic optimisation 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.

## 2 Proposed Solution

**Darwinian Data Structure Selection ($DS^2$) problem**: given a program with a list of replaceable data structures, find the optimal combination of data structures and their arguments, such that the runtime performance of the program is optimised.

In order to solve the $DS^2$ problem, we proposed a language-agnostic optimisation framework, ARTEMIS. Figure 1 illustrates the architecture of ARTEMIS. 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. 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.

**DARWINIAN DATA STRUCTURE STORE GENERATOR (DDSSG)** automatically builds a store of *Darwinian Data Structures* that can be exposed as tunable parameters to the OPTIMISER. DDSSG uses a hierarchy graph to represent the inheritance relations between classes, then it groups the replaceable classes together, as its output. 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, 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 of equivalent implementations for each interface; *e.g.,* {`List`, `ArrayList`, `LinkedList`}. Those implementations will be considered as replaceable during code execution and will be exposed as parameters to the OPTIMISER.

**EXTRACTOR** takes as input the program source code, identifies potential locations of the code that contain 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 generated from DDSSG. 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 generates a templated copy of the AST, with all discovered DARWINIAN data structures replaced by template identifiers (holes).

**OPTIMISER** is to find a combination of data structures that improves the performance of the initial program. 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 [2] to search for optimal solutions [12,7,1,8].

We use an array of integers to represent the tuning parameters. Each parameter may refer either to an equivalent data structure or a parameter of the data structure such as the initial size. Together with the templated AST generated from the OPTIMISER, ARTEMIS can rebuild the program with a different set of data structures. For each iteration of the algorithm, 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 converge. Finally, all non-dominating solutions in the final population are provided as solutions.

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. The instrumented code is run before the optimisation begins and it generates a database with the most costly parts of codes worth optimising. 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.

## 3 Experiments and Results

To assess how effectively ARTEMIS can improve a program's performance, we used Guava[1] as an instance of its application. Guava is an open-source set of common libraries for Java. It consists of 252,688 Lines of Code, which are tested by 1,674,425 test cases with 61.7% branch coverage. We conducted experiments with Oracle JDK 1.8 and Ubuntu 16.04 on top of machines featuring 8 cores and 14GB of DRAM. We used JVM profiling tools for performance measurements. To mitigate instability and incorrect results [3], we differentiate VM start-up and steady-state. We do repeated measurements for 30 times and record the measurements before and after doing the optimisations. Also we use Mann Whitney U test to examine if the improvement is statistically significant. For the settings of the optimisation algorithm, we used an initial population size of 30 and a maximum number of 900 function evaluations. These numbers were chosen after a few calibration experiments to ensure the best performance of the algorithm.

For the rest of this section, we asked four Research Questions and provided answers with supportive results in each of the following paragraphs.

***RQ1*** What is the improvement that ARTEMIS provides for each objective?

We ask this question to understand how much improvement ARTEMIS can achieve on each of the objectives and how the other objectives are affected. Firstly, we compute the mean response time and report the 95% confidence interval. We use effect size [5] for measuring the performance impact. To quantify the effects we use Cohen's d [5] strength values: small ($0.2 < d \leq 0.5$), medium ($0.5 < d \leq 0.8$) and large ($0.8 < d$). In Figure 2 we plot the mean values for the optimal solutions that contain at least one large improvement for one of the three measurements. The maximum improvement for each measure is 9% execution time, 13% memory usage and 4% CPU usage.

***RQ2*** How many provided solutions strictly dominate the original program?

[1] `https://github.com/google/guava`

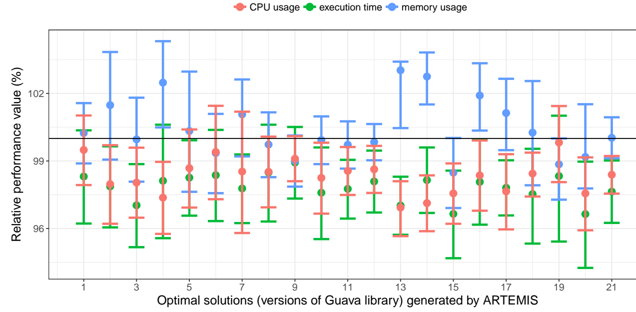


Fig. 2: Optimal solutions with large improvement in at least one measure.

A solution is said to strictly dominate another if it outperforms the other in all measures. If ARTEMIS can provide solutions that strictly dominate the original program, those solutions can be very valuable because they represent options to improve the program without sacrificing anything. The number of strictly dominating solution for Guava was 14 out of 51 final solutions. Those 14 solutions provide a wide range of options for users to choose depending on their favour of different objectives.

***RQ3*** What is the cost of using ARTEMIS?

This question asks about the computational cost of ARTEMIS. An extremely high computational cost may make the system impractical to use in real-world situations. Therefore we measured its cost on Guava subject in terms of machine hours. In this study, a Microsoft Azure D4-v2 machine, which costs £0.41 per hour[2], was used to conduct all experiments. This cost of using 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.

***RQ4*** How many Darwinian data structures does ARTEMIS optimise?

We ask this question to understand what changes have been made to the program. To minimise the search space we applied ARTEMIS only to the most used code in Guava as identified by the preprocessor. As a result, ARTEMIS extracted only 6 Darwinian data structures in total from the Guava library. Across all the optimal solutions ARTEMIS produced, 1 to 6 data structures were changed in each, with a median of 3 data structures uses. For instance, ARTEMIS replaced `HashMap` with `LinkedHashMap` in 42 of the 135 changes across all optimal solutions.

## 4 Conclusions

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 a given program. ARTEMIS is language agnostic, meaning it can be easily adapted to any programming language. On a large real-world system,

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

Guava, ARTEMIS found 9% improvement on execution time, 13% improvement on memory consumption and 4% improvement on CPU usage separately, and 27.45% of the final solutions provides improvement without sacrificing other objectives. Lastly, we estimated the cost of optimising Guava in machine hours. With a price of £0.41 per machine hour, the cost of optimising a real-world system such as Guava in this study is less than £7.85. Therefore, we conclude that ARTEMIS is a practical tool for optimising the data structures in large real-world programs.

## References

1. Haitao Dan, Mark Harman, Jens Krinke, Lingbo Li, Alexandru Marginean, and Fan Wu. *Pidgin Crasher: Searching for Minimised Crashing GUI Event Sequences*, pages 253–258. Springer International Publishing, Cham, 2014.
2. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.
3. Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.
4. Brett Hardin. Companies with "hacking" cultures fail. `https://blog.bretthard.in/companies-with-hacking-cultures-fail-b8907a69e3d`, 2016. [Online; accessed 25-February-2017].
5. Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.
6. Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974.
7. William B Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3d medical image registration cuda software with genetic programming. In *Proceedings of the 2014 GECCO*, pages 951–958. ACM, 2014.
8. Lingbo Li, Mark Harman, Fan Wu, and Yuanyuan Zhang. *SBSelector: Search Based Component Selection for Budget Hardware*, pages 289–294. Springer International Publishing, Cham, 2015.
9. Basios Michail, Lingbo Li, Fan Wu, Leslie Kanthan, Donald Lawrence, and Earl Barr. Darwinian data structure selection. *arXiv preprint arXiv:1706.03232*, 2017.
10. Ronald 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].
11. Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In *ACM Sigplan Notices*, volume 44, pages 408–418. ACM, 2009.
12. Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382. ACM, 2015.
13. Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: profiling copies to find runtime bloat. *ACM Sigplan Notices*, 44(6):419–430, 2009.
14. Guoqing Xu and Atanas 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.