Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava

Nathan Burles[1], Edward Bowles[1], Alexander E. I. Brownlee[2], Zoltan A. Kocsis[2], Jerry Swan[1], and Nadarajen Veerapen[2]

[1] University of York, York, YO10 5DD, United Kingdom
[2] University of Stirling, Stirling, FK9 4LA, United Kingdom

Abstract. In this work we use metaheuristic search to improve Google's Guava library, finding a semantically equivalent version of `com.google.common.collect.ImmutableMultimap` with reduced energy consumption. Semantics-preserving transformations are found in the source code, using the principle of subtype polymorphism. We introduce a new tool, Opacitor, to deterministically measure the energy consumption, and find that a statistically significant reduction to Guava's energy consumption is possible. We corroborate these results using Jalen, and evaluate the performance of the metaheuristic search compared to an exhaustive search—finding that the same result is achieved while requiring almost 200 times fewer fitness evaluations. Finally, we compare the metaheuristic search to an independent exhaustive search at each variation point, finding that the metaheuristic has superior performance.

Keywords: Genetic Improvement, Object-Oriented Programming, Subclass substitution, ov Substitution Principle, Energy Profiling

1 Introduction

Across all scales of computing, from mobile devices to server farms, there is widespread interest in minimizing energy requirements. For a given program, it is likely that there are many functionally-equivalent programs exhibiting a variety of different non-functional properties. Previous work by Sahin et al. [13] has measured the effect of 6 popular refactorings on 9 real Java programs, concluding that the effect of these refactorings on energy usage are highly end-application dependent and that commonly-applied predictive metrics are of little practical use. This is therefore a strong motivator for the application of techniques from Search Based Software Engineering (SBSE). In this article, we use metaheuristic search to find semantically equivalent programs with reduced energy consumption. Semantics preserving transformations are achieved via the behavioral equivalence that is central to object-orientation.

Related Work. Although there are a number of works in Genetic Programming (GP) and Grammatical Evolution (GE) that claim to be 'Object Oriented' [1, 2, 8, 12, 15], we are not aware of any concerned with the central pillar of Object Orientation, viz. the Liskov Substitution Principle (LSP) [7] as exemplified by subtype polymorphism. In this respect, the closest work we are aware of is that of the SEEDS framework of Manotas et al. [9], in which alternative subtypes of container classes are substituted into bytecode in order to minimize power

consumption. The search mechanism that is employed in SEEDS is that of a separate exhaustive search at each object allocation location: our approach differs by using a genetic approach to assign subclasses to constructor invocations. It is certainly therefore the first work deserving of the title of 'Object-Oriented Genetic Improvement'. Related work in a non-SBSE context is the interesting use of strong-typing corresponding to different operating modes (e.g. 'battery level high') [4], allowing the programmer to delineate differing responses to operating conditions (e.g. opting to render a low resolution image when energy is low).

2 Implementation

In outline, the improvement process is as follows:

1. Parse the source file designated for improvement, yielding an Abstract Syntax Tree (AST). Identify *variation points*, i.e. source nodes in the AST corresponding to the creation of Guava container objects.
2. Obtain the complete set of possible target substitutions T (i.e. all the container classes within the Guava, Apache Collections[1], and the Java 8 `util` package). For each source node S_i in the AST, find the subset of possible target substitutions $t(S_i) \subseteq T$ which are actually valid.
3. Given a sequence of the k source nodes $[S_1, \ldots, S_k]$ from the AST, the search space is then given by all combinations from $[t(S_1), \ldots, t(S_k)]$. The solution representation is thus an assignment $i \mapsto s \in t(S_i), 1 \leq i \leq k$, represented as an element $r \in \mathbb{Z}^k$, with constraints $0 \leq r_i < \left|t(S_i)\right|$.
4. Given such an assignment, the AST node for each variation point can be replaced with its target substitution and the correspondingly mutated source file can be written out to disk.
5. The program containing the mutated source is then compiled and evaluated by a measure related to its energy consumption. By this means, combinatorial search is performed in the space of these representations using the Genetic Algorithm metaheuristic [6], in order to find the sequence of substitutions which minimize energy consumption.

Variation Points. The open-source Java framework Google Guava implements a variety of concrete subclasses of `java.util.Collection`. There are well-known tradeoffs in performance characteristics between different collection subclasses: for example, finding a specific element in a linked list is $\mathcal{O}(n)$ but in a hash-set it is $\mathcal{O}(1)$. We selected `com.google.common.collect.ImmutableMultimap` as a testcase for improvement, since it features a number of instantiations of Collection subclasses. Source file parsing (and subsequent re-generation, described below) was done with a popular open-source library `com.github.javaparser.JavaParser`[2]. Within this file, three types of syntax fragments were identified for use as variation points: calls to constructors (e.g. `new LinkedHashMap<>()`); calls to Guava factory classes (e.g. `Maps.newHashMap()`); and calls to Guava static creator methods (e.g. `ImmutableList.of()`). These were filtered to include only fragments

[1] Guava v18.0, Apache Collections v4.0

[2] During development we discovered and fixed a bug in the hashCode implementation of the Node class in the `com.github.javaparser.JavaParser` library.

creating a collection object, i.e., one implementing `java.util.Collection`, `java.util.Map` or `com.google.common.collect.Multimap`. In total, 5 variation points were found, with between 5 and 45 possible substitutions each.

Mutating the Source Code. For each variation point, one of three approaches to determining the interface of the created object was used: the method's return type for return statements; the declared type for variable declarations. For other expressions, the least-general interface implemented by the class was used: one of `Map,Set,List,Multimap,Multiset` or `Collection`. The potential substitutions for a variation point are classes implementing the appropriate interface in Guava, Apache Collections, `java.util`, and `java.util.concurrent`. Excluded were abstract and inner classes, and those expecting a particular type (e.g. `Tree` collections require elements implementing `Comparable`). A modified version of JavaParser's `SourcePrinter` performed the required code substitution at each variation point in `ImmutableMultimap`'s source. A careless programmer might depend on functionality not guaranteed by the LSP, e.g. relying on a collection being sorted, despite this not being part of the superclass type specification that is actually visible at the point at which such reliance is made. This is a logical error on behalf of the programmer, and strictly speaking necessitates that they rewrite the code. Although we do not cater for such programming errors in our experiments, the best that can be done in such cases is to include any existing unit tests as a constraint on the search.

Measuring Energy Consumption. During the evolutionary process the fitness function was the energy consumption measured using a new tool, Opacitor, designed to make measurements deterministic. Opacitor traces the execution of Java code, using a modified version of OpenJDK, generating a histogram containing the number of times each Java opcode was executed—allowing very similar programs to be distinguishable. A model of the energy costs of each Java opcode, created by Hao et al. [5], is then used to calculate the total number of Joules used. As the Just-In-Time compilation (JIT) feature of the Java Virtual Machine (JVM) is non-deterministic, it is disabled during evolution. Similarly, Garbage Collection (GC) is non-deterministic and so the JVM is allocated enough memory to avoid GC. During the final testing, after evolution has completed, these features are re-enabled to ensure that the results remain valid on an unmodified JVM. It should be noted that a significant benefit of Opacitor, compared to other approaches which require timing or physical energy measurement, is that it is unaffected by anything else executing on the experimental system. This means that it can be parallelised, or executed simultaneously with other programs, without difficulty. In previous work [14] we successfully used Jalen [11] to calculate the energy required, and compare two algorithms; during the final testing we have used this technique as a corroboration that Opacitor is effective and generates reliable measurements.

3 Experiments

The first experiment performed was the use of a metaheuristic to search for a solution with reduced energy consumption. We elected to use a Genetic Algorithm (GA) [6] to search the space of solutions, since this is known to be a useful approach for a variety of assignment problems [3]. The solution representation

used is a vector of integers $r \in \mathbb{Z}^k$. The representation itself is constrained with the required constraints $0 \leq r_i < \left|t(S_i)\right|$.

The GA was configured with a population of 500, running for 100 generations. New populations were generated using an elitism rate of 5%, single-point crossover with a rate of 75%, and one-point mutation with a rate of 50% with candidates selected using tournament selection with arity 2. These parameters were selected, after preliminary investigations, in order to provide sufficient genetic diversity to the evolution without requiring an excessive number of fitness evaluations as each evaluation takes in the order of 10 seconds on a 3.25GHz CPU. During experimentation we ran the GA five times, with a different seed to the random number generator, in order to test the robustness of the evolution.

A second experiment ran an exhaustive search on the entire search space of 674,325 possible solutions, using 32 3.25GHz cores to allow it to finish within 2.5 days, in order to determine how close the GA came to finding an ideal solution.

The final experiment ran an exhaustive search independently on each variation point, following the example of Manotas et al. [9] but on source code instead of bytecode, before combining each of the substitutions at the end.

4 Results

Statistical testing was carried out using the Wilcoxon / Mann Whitney U Statistical Tests and Vargha-Delaney Effect size tests (as implemented in the Astraiea framework [10]). The results were obtained with 100 samples in each dataset. The result of each of the runs of the GA was the same set of substitutions, and so this set was used thereafter.

When considering a full exhaustive search of the entire problem space of 674,325 possible combinations of substitutions, performed to gauge the effectiveness of the GA, the best set of substitutions were the same as those found by the GA. The use of a GA therefore provided a speed-up of almost 200 while still successfully finding the best possible result.

The combined results of the original library, our improved version, and the version using only independent exhaustive search are shown together in Table 1. During the evolution, with JIT disabled and GC avoided, the best solution found used 216.49 J. This compares with 298.58 J required by the original, and 266.43 J

Table 1. Energy (J) required to exercise the various methods provided by the `ImmutableMultimap` class 10,000 times (mean of 100 runs, and standard deviation σ), as well as the p-values (p) and effect size measures (e) comparing our result to the original or the result of an independent exhaustive search at each variation point.

Measurement technique	GA	Original			Independent Exhaustive		
	J	J	p	e	J	p	e
Opacitor	216.49	298.58	–	–	266.43	–	–
Opacitor with JIT and GC	11.15 σ2.06	14.75 σ1.13	<.001	0.93	13.45 σ1.15	<.001	0.85
Jalen	11.81 σ2.18	15.25 σ1.00	<.001	0.94	13.46 σ0.66	<.001	0.82

required by the solution found using an exhaustive search independently on each variation point. As the measurements in this case are deterministic, and thus generate only one observation for each version, no statistical tests are necessary.

The GA required approximately 3,500 fitness evaluations to find its best solution (the number of evaluations varied slightly between different evolutionary runs), while the exhaustive search at each variation point required only 105 evaluations. Although using the GA was therefore significantly slower than the approach used in similar work [9], the final result is also significantly better.

More interestingly, similarly impressive results were also obtained when non-determinism was reintroduced post-evolution—with JIT enabled and the JVM's memory allocation unmodified (allowing for GC when necessary). In this case the GA's solution required 11.15 J, compared with 14.75 J for the original and 13.45 J for the independent exhaustive search. As the energy measurement is no longer deterministic, the p-values and effect size measures vary accordingly. Vargha and Delaney suggest that a value of 0.71 indicates a large difference between data sets, and so the results demonstrate that the GA's solution provides a significant improvement over both the original and the independent exhaustive search.

To help corroborate that the model-based energy measurement provides realistic results, we used JALEN (with JIT and GC) to compare the three versions of `ImmutableMultimap`. The results support the assertion that OPACITOR provides realistic and reliable energy measurements, as well as the hypothesis that performing the evolution using the deterministic measure would map correctly to results generated in a non-deterministic, realistic environment.

5 Conclusion

We have introduced 'Object-Oriented Genetic Improvement', a technique by which non-functional properties such as time or energy consumption may be optimised by substituting suitable alternative subclasses to constructor invocations. By virtue of subclass adherence to the 'Liskov Substitution Principle' [7], we can make semantics-preserving changes to source code in order to take advantage of the vastly different performance characteristics displayed by different collection implementations. We applied this technique to the `com.google.common.collect.ImmutableMultimap` class, part of Google's Guava library, using a new tool, OPACITOR, to evaluate the energy consumption of candidate solutions.

Our results showed that significant improvements could be made, with the best solution providing a saving of approximately 24%. The results generated by OPACITOR were corroborated using JALEN, which uses time and CPU utilisation as a proxy for energy consumption. Thus, the results show that the substitutions improve both the energy consumption and the execution time of the class. We further compared the results of our technique to those obtained using an approach used in related work [9]—a separate exhaustive search at each variation point—and found that although the number of fitness evaluations increased using a GA, the performance of the final result was significantly improved. This shows that the variation points within code are not always independent. This is intuitively the case for `ImmutableMultimap`—two of the identified variation points instantiate the `BuilderMultimap` private class which exists within `ImmutableMultimap`, while other variation points exist within the `BuilderMultimap`

sub-class. An independent exhaustive search at each variation point may therefore decide to substitute `BuilderMultimap` for a more efficient alternative, even though more efficient subclass substitutions can be made within the private class.

Acknowledgement. Work funded by UK EPSRC grant EP/J017515/1.

References

1. Abbott, R.J.: Object-oriented genetic programming, an initial implementation. In: Proceedings of the 6th International Conference on Computational Intelligence and Natural Computing. North Carolina USA (2003)
2. Bruce, W.S.: Automatic generation of object-oriented programs using genetic programming. In: Proceedings of the 1st Annual Conference on Genetic Programming. pp. 267–272. MIT Press, Cambridge, MA, USA (1996)
3. Chu, P.C., Beasley, J.E.: A genetic algorithm for the generalised assignment problem. Comput. Oper. Res. 24(1), 17–23 (Jan 1997)
4. Cohen, M., Zhu, H.S., Senem, E.E., Liu, Y.D.: Energy types. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 831–850. OOPSLA '12, ACM, NY, USA (2012)
5. Hao, S., Li, D., Halfond, W.G., Govindan, R.: Estimating mobile application energy consumption using program analysis. In: 35th International Conference on Software Engineering. pp. 92–101. IEEE (2013)
6. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press, Cambridge, MA, USA (1992)
7. Liskov, B.: 'Data Abstraction and Hierarchy' (keynote address). SIGPLAN Not. 23(5), 17–34 (Jan 1987)
8. Lucas, S.M.: Exploiting reflection in object oriented genetic programming. In: Genetic Programming, pp. 369–378. Springer (2004)
9. Manotas, I., Pollock, L., Clause, J.: Seeds: A software engineer's energy-optimization decision support framework. In: Proceedings of the 36th International Conference on Software Engineering. pp. 503–514. ACM, NY, USA (2014)
10. Neumann, G., Swan, J., Harman, M., Clark, J.A.: The executable experimental template pattern for the systematic comparison of metaheuristics. In: Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion. pp. 1427–1430. ACM (2014)
11. Noureddine, A., Bourdon, A., Rouvoy, R., Seinturier, L.: Runtime monitoring of software energy hotspots. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 160–169. IEEE (2012)
12. Oppacher, Y., Oppacher, F., Deugo, D.: Evolving java objects using a grammar-based approach. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation. pp. 1891–1892. ACM, NY, USA (2009)
13. Sahin, C., Pollock, L., Clause, J.: How do code refactorings affect energy usage? In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 36:1–36:10. ACM, NY, USA (2014)
14. Swan, J., Burles, N.: Templar–a framework for template-method hyper-heuristics. In: Genetic Programming, pp. 205–216. Springer (2015)
15. White, T., Fan, J., Oppacher, F.: Basic object oriented genetic programming. In: Mehrotra, K., Mohan, C., Oh, J., Varshney, P., Ali, M. (eds.) Modern Approaches in Applied Intelligence, pp. 59–68. LNCS 6703, Springer (2011)