

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

Nathan Burles¹, Edward Bowles¹,
Alexander E. I. Brownlee², Zoltan A. Kocsis²,
Jerry Swan¹, Nadarajen Veerapen²

¹ Department of Computer Science, University of York

² Computing Science and Mathematics, University of Stirling

What actually *is* OO?

The following have historically been defined as being central to Object-orientation [1]:

- Encapsulation.
- Inheritance.
- Abstraction via Subtype Polymorphism.

Of these, the latter is then most important distinguishing feature: it is what allows the construction of large and re-useable frameworks via Meyer's *open-closed* principle [6]. This is only really possible if the *Liskov Substitution Principle* is followed [4].

The Liskov Substitution Principle

The LSP is a form of *design by contract*, constraining derived classes as follows [6]:

... when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

Exploiting the LSP for GI

The LSP allows us to safely substitute subclasses for superclasses. The original intent was that this substitution be performed by human software engineers, but it is obviously very useful for larger-scale transformations than have traditionally been used in GI.

Pitfall: Java collections aren't always strictly LSP-conformant (e.g. throwing 'UnsupportedOperationException'). Hence, in practice, testing **is** required.

Implementation Outline

In outline, the process is as follows:

- Parse the source file, obtaining an abstract syntax tree—identify *variation points* within this AST.
- Find the subset of possible target substitutions (from all container classes within Guava, Apache Collections, and Java 8's `util` package) which are valid for each variation point.
- The search space is then all combinations of valid substitutions over the identified variation points.
- Select an assignment, replace the variation point with its target substitution, and write the mutated source to disk.
- Compile, and evaluate this source in terms of its energy consumption.

Variation Points

Google Guava implements a variety of concrete subclasses of `java.util.Collection`.

Well-known tradeoffs in performance between different subclasses: e.g. finding an element in a linked list is $\mathcal{O}(n)$, in a hash-set $\mathcal{O}(1)$.

We selected `ImmutableMultimap` as a test-case, as it contains a number of instantiations of `Collections` subclasses. Three types of instantiations were identified as variation points:

- Calls to constructors (e.g. `new LinkedHashMap<>()`).
- Calls to factory classes (e.g. `Maps.newHashMap()`).
- Calls to static creator methods (e.g. `ImmutableList.of()`).

Mutating the Source Code

At each variation point, the interface of the created object was determined by one of three methods:

- Return statements: the method's return type.
- Variable declarations: the declared type.
- Other expressions: the least-general interface implemented by the class, one of `Map`, `Set`, `List`, `Multimap`, `Multiset`, or `Collection`.

Note: Careless programmers may e.g. rely on a collection being sorted, despite this not being part of the interface. This is an error, and should be fixed—the best that can be done is to use existing unit tests during the search.

Measuring Energy Consumption: Existing methods

Hardware tools exist, such as the data provided by an Uninterruptible Power Supply or electronic watt-meter. These are useful for a *coarse-grained overview*. . . their precision and accuracy is too low for comparing very similar algorithms.

Software alternatives such as:

- JALEN [7]—targets Java; estimates power consumption as a function of execution time, CPU utilisation, and clock frequency (low precision as reliant on timing: experiments must be repeated multiple times within a measurement).
- Wattch [2]—cycle-level simulator, provides ability to distinguish between very similar programs. . . requires a parameterised model of CPU, and doesn't support Java

OPACITOR (1)

OPACITOR traces the execution of Java code, using a modified version of OpenJDK. This JVM generates a histogram counting the number of times each Java opcode was executed.

Uses a model of energy costs of each Java opcode created by Hao et al. [3] in their eLens work to calculate the energy consumption.

It provides ability to distinguish programs down to *single instruction*. . . but accuracy depends on the model used!

OPACITOR (2)

Most important in this work is comparative accuracy, e.g. $a < b$ is correct, rather than $a = 3.78543J$. . . this depends on the variability of energy consumption by opcodes, particularly those dependent on operands.

A further complication is the significant **variability between different runs** of the exact same Java program: both *Garbage Collection* and *Just-in-Time compilation* are non-deterministic, which greatly affects execution time and power consumption.

OPACITOR (3)

In OPACITOR, therefore, during evolution GC and JIT are both disabled. This allows runs to be repeatable.

In final testing, all features are enabled to ensure final results are valid on a standard JVM.

This leads to an important benefit of OPACITOR, compared to timing or wall-power measurements—its determinism means that it can be executed in parallel or concurrently with other software.

Experiments

- 1 Using a GA to search the space of solutions (674,325), repeated 5 times with different seeds.
- 2 Exhaustive search on the entire space (parallelised to allow completion in reasonable time—approximately 10 seconds per evaluation).
- 3 Exhaustive search independently on each variation point, following the example of Manotas et al. [5] (on source rather than bytecode).

Results (1)

Measurement technique	GA	Original		
	J	J	p	e
OPACITOR	216.49	298.58	–	–
OPACITOR with JIT and GC	11.15 σ 2.06	14.75 σ 1.13	<.001	0.93
JALEN	11.81 σ 2.18	15.25 σ 1.00	<.001	0.94

Table: Energy used by test program as measured by OPACITOR, OPACITOR with JIT and GC enabled, and JALEN. Each measurement shows the mean and standard deviation over 100 runs of two versions of `ImmutableMultimap`—the GA result vs. original.

Results (2)

Measurement technique	GA	Independent Exhaustive		
	J	J	p	e
OPACITOR	216.49	266.43	–	–
OPACITOR with JIT and GC	11.15 σ 2.06	13.45 σ 1.15	<.001	0.85
JALEN	11.81 σ 2.18	13.46 σ 0.66	<.001	0.82

Table: Energy used by test program as measured by OPACITOR, OPACITOR with JIT and GC enabled, and JALEN. Each measurement shows the mean and standard deviation over 100 runs of two versions of `ImmutableMultimap`—the GA result vs. the “Independent Exhaustive Search” result.

Conclusions (1)

We have introduced ‘Object-Oriented Genetic Improvement’, a technique to optimise non-functional properties, e.g. time or energy, by subclass substitution and applied this technique to Google Guava’s `ImmutableMultimap` class, using OPACITOR to evaluate energy consumption.

The results showed that energy savings of approximately **24%** could be made optimising this class alone.




These results were corroborated by JALEN, using time and CPU utilisation as a proxy to estimate energy consumption.

Conclusions (2)




The exhaustive search showed that our GA found the best possible solution, using 3,500 fitness evaluations (vs. space of 674,325 candidates—almost 200 times faster than exhaustive).

The results of the approach used in [5] shows that variation points within code are **not always independent** (although it only required 105 fitness evaluations). In this case, two of the variation points instantiate the private `BuilderMultimap` class, where other variation points exist within that class. . . substituting `BuilderMultimap` for an alternative may miss a better substitution inside the private class.

References I

-  Grady Booch et al. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. 3rd ed. Addison-Wesley Professional, Apr. 2007. ISBN: 020189551X.
-  David Brooks, Vivek Tiwari, and Margaret Martonosi. “Wattch: a framework for architectural-level power analysis and optimizations”. In: *27th Annu. Int. Symp. on Comput. Archit., ISCA '00*. ACM, 2000, pp. 83–94.
-  Shuai Hao et al. “Estimating mobile application energy consumption using program analysis”. In: *35th International Conference on Software Engineering*. IEEE, 2013, pp. 92–101.

References II

-  Barbara Liskov. “Data Abstraction and Hierarchy’ (keynote address)”. In: *SIGPLAN Not.* 23.5 (Jan. 1987), pp. 17–34. ISSN: 0362-1340.
-  Irene Manotas, Lori Pollock, and James Clause. “SEEDS: A Software Engineer’s Energy-optimization Decision Support Framework”. In: *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, India: ACM, 2014, pp. 503–514. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568297.
-  Bertrand Meyer. *Object-Oriented Software Construction*. 2nd. Prentice Hall PTR, 1997. ISBN: 0136291554.

References III



Adel Nouredine et al. “Runtime monitoring of software energy hotspots”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2012, pp. 160–169.