Search-Based Energy Optimization of Some Ubiquitous Algorithms

Alexander Edward Ian Brownlee, Nathan Burles, and Jerry Swan

***Abstract*—Reducing computational energy consumption is of growing importance, particularly at the extremes (i.e., mobile devices and datacentres). Despite the ubiquity of the Java virtual machine (JVM), very little work has been done to apply search-based software engineering (SBSE) to minimize the energy consumption of programs that run on it. We describe OPACITOR, a tool for measuring the energy consumption of JVM programs using a bytecode level model of energy cost. This has several advantages over time-based energy approximations or hardware measurements. It is 1) deterministic, 2) unaffected by the rest of the computational environment, 3) able to detect small changes in execution profile, making it highly amenable to metaheuristic search, which requires locality of representation. We show how generic SBSE approaches coupled with OPACITOR achieve substantial energy savings for three widely used software components. Multilayer perceptron implementations minimizing both energy and error were found, and energy reductions of up to 70% and 39.85% were obtained over the original code for Quicksort and object-oriented container classes, respectively. These highlight three important considerations for automatically reducing computational energy: tuning software to particular distributions of data; trading off energy use against functional properties; and handling internal dependencies that can exist within software that render simple sweeps over program variants sub-optimal. Against these, global search greatly simplifies the developer's job, freeing development time for other tasks.**

***Index Terms*—Energy, Java, search based software engineering.**

I. Introduction

ENERGY consumption related to program execution is of growing importance at all scales, from mobile and embedded devices through to datacentres [1]–[3]. For mobile applications, battery life is a critical aspect of user experience and while several factors (such as the display, wifi and GPS) have a large impact on power consumption, CPU usage is still a major factor [4]. For example, the maximum CPU power of a Samsung Galaxy S3 is 2,845 mW, 2.53× the maximum power consumption of the screen and 2.5× that of the 3G hardware [5]. At the other end of the scale, the repeated execution of specific subroutines by server farms offers the potential for considerable energy saving via the identification of energy-intensive 'hotspots' in the code. The electricity consumption by servers was estimated at between 1.1 and 1.5% of global electricity production in 2010 [6], with energy consumption reaching 50-100% of the purchase cost of the hardware over its lifetime [7]. These facts motivate the use of automated methods to reduce the energy consumption attributable to software, allowing computational search to relieve developers of the burden of finding energy-optimal software implementations. Predictive 'decision support' metrics, recently shown to be of little practical use [8] can then be replaced with automatic tuning and refactorings that demonstrably reduce energy consumption. The ultimate goal is to obtain general methods that can reduce energy consumption, analogously to the familiar process of automatic compiler optimization for execution time. This is an ideal application for Search-Based Software Engineering (SBSE), the use of search based optimisation for software engineering problems [9], with initial work in this area showing promising results [2], [10]–[14].

Given the prevalence of the Java Virtual Machine (JVM)[1], it is surprising that there has been little previous work on energy optimization of Java applications. In this paper, we describe OPACITOR, a tool implementing a new approach to measuring the energy consumption of applications running on a JVM. Previously, run time [15] or system calls [16], [17] have been used as proxies for energy use, but it has been shown [18] that, independent of the total number of CPU cycles, there can be large differences in the energy consumed by different opcodes. An alternative, high-level hardware measurements [19], [20], are non-trivial to implement and subject to noise that hinders a search process. In contrast, OPACITOR uses a predefined cost model of the energy consumed by each JVM opcode, thereby providing a deterministic (and so noise-free) approximation of the total energy consumption. It also has the advantage of being unaffected by the rest of the computing environment, and can detect small changes in execution profile, down to opcode level.

We use OPACITOR to measure the effectiveness of three distinct SBSE [9] approaches for minimising energy consumption of some very widely-used algorithms, and highlight three important aspects of SBSE applied to energy consumption. The focus is on Java implementations, but the techniques are general and

Manuscript received November 5, 2016; revised April 10, 2017; accepted April 13, 2017. Date of current version May 23, 2017. This work was supported by UK Engineering and Physical Sciences Research Council under Grant EP/J017515/1 (DAASE). *(Corresponding author: Alexander Edward Ian Brownlee.)*

A. E. I. Brownlee is with the Division of Computing Science and Mathematics, University of Stirling FK9 4LA, U.K. (e-mail: sbr@cs.stir.ac.uk).

N. Burles is with the IBM United Kingdom, Ltd., York YO10 5GA, U.K. (e-mail: nathan.burles@york.ac.uk).

J. Swan is with the York Centre for Complex Systems Analysis, University of York, York YO10 5DD, U.K. (e-mail: jerry.swan@york.ac.uk).

Digital Object Identifier 10.1109/TETCI.2017.2699193

[1]Users are reminded at each installation that over 3 billion devices use Java: http://www.java.com/en/about/ , and Java is regularly reported as the most popular programming language: https://www.tiobe.com/tiobe-index/

are applicable across a wide range of programming languages. The three approaches are:

1) Genetic Programming (GP) to obtain better pivot functions for Quicksort, a ubiquitous algorithm for in-place sorting of a sequence in random access memory, highlighting that energy consumption can be tuned to the specific application of the code – the datasets in this case – motivating automated methods to perform this tuning.
2) Hyperparameter search over the efficiency/accuracy tradeoff of Multi-Layer Perceptrons, a popular neural net architecture for classification, incorporating novel activation functions via Automatic Differentiation, demonstrating that energy consumption can be traded-off against a functional property – classification accuracy in this case.
3) Genetic Algorithms (GA) using the constraints of Object Oriented Programming for Guava and Apache Commons collection classes (OO-GI), library implementations of well-known data structures, revealing that different software components can exhibit subtle interactions with respect to energy, vastly increasing the search space and justifying the use of SBSE to reduce developer effort.

The first two of these approaches are applicable to any language, further extending their potential impact. OO-GI is applicable to any OO code beyond the collections classes that we have chosen as our focus and, more generally, wherever program behaviour is specified via ‘Design by Contract’ [21].

This work makes four major contributions. Firstly, we describe and demonstrate OPACITOR, a new tool for measuring the energy consumption of Java programs. Secondly, we extend work from two earlier papers [22], [23] (respectively, proposing evolution of pivot functions for Quicksort and introducing OO-GI) with more extensive experiments and analysis, demonstrating the wider applicability of both approaches. Thirdly, we propose a combination of automatic-differentiation and topology parameterisation to allow search-based improvement of multi-layer perceptrons. Finally, we show that the energy consumption of three very widely-used software components can readily be improved. The results demonstrate that global search methods allow improvements in energy to be targeted to specific datasets, be traded off against other functionality, and can accommodate hard-to-determine interactions within the code. All three of these would otherwise result in lengthy, repetitive exploration through large spaces of alternative designs. The importance to industry of these results is that automated search methods reduce the required developer effort: the SBSE approaches we describe can achieve significant reductions in energy use for little development cost. Given the ubiquity of these components, it demonstrates the potential energy savings for a very wide range of applications.

We begin by summarising related work in Section II. We describe OPACITOR in Section III. Sections IV–VI describe the experiments with each of the three studied algorithms (Quicksort, MLP and OO collections), each including implementation details, results and discussion. We consider possible threats to the validity of the work in Section VII before drawing our conclusions and suggesting future work in Section VIII.

II. RELATED WORK

The energy consumption attributable to software is becoming increasingly important to software developers [1], particularly as software development moves away from the ‘mid-range’ of the desktop (and the relatively homogeneous architectural configuration of PCs) to the extremes of both large- and small- scale computing [2].

A variety of approaches have been described for measuring the energy consumption due to software. Often, CPU time is used as a proxy for energy use [15], but it has been found that this can be inaccurate, particularly because this omits CPU idle states [24]. As one alternative, both [16] and [17] used a regression model built on the number of a system calls made by a running program. Static or dynamic analysis of program paths with respect to known power consumption of hardware components can also be used [25]. Several pieces of research have measured power consumption directly using instrumentation hardware. This can be in terms of the overall system power [26], which can be measured by the current drawn from the battery on mobile devices, such as in [19] and in the “Green Miner” platform [20]. Power consumption due to the CPU can also be determined via the Intel Power Gadget API [2], [27]. It is possible to make such high-level measurements with a frequency high enough that energy use patterns can be matched to particular source code lines [28], to the use of particular libraries [29], or specific API calls [30]. Obvious issues with hardware measurements are the need for support from the existing system components, or the non-trivial task of adding physical probes to existing circuitry. Furthermore, hardware solutions are typically course-grained, measuring the consumption of the whole system. Background processes and changing environmental conditions then lead to noisy, non-repeatable measurements. These must then be handled by the SBSE algorithm, often by repeating the measurements, extending run-times and reducing confidence in the results. A deterministic measurement of energy consumption would avoid this issue. One such approach is the static analysis of energy consumption attributable to specific instructions [31], in this case targeted to intermediate compiler representations of a program in the LLVM toolchain. While this is similar in concept to OPACITOR, the latter measures energy consumption due to each bytecode at runtime, which may be more useful where the execution pathway is highly dependent on the input. Also, to the best of our knowledge, OPACITOR is the only broadly comparable tool that is targeted at the JVM.

There are currently only a few publications which use SBSE to reduce energy consumption. One of the earliest applied multi-objective GP for generating random number generators [10]. More generally, Genetic Improvement (GI) has been applied the C source code of the boolean satisfiability solver MiniSAT [2], with energy measured by the Intel Power Gadget. The solver was specialised to three different application domains with reductions in energy consumption of 5 to 25%. GI has also been applied to x86 assembly code with a fitness function using hardware performance counters (special hardware registers available on most modern CPUs) [11], achieving energy reductions of up to 20%. *Code perforation* (finding parts of the code that can be skipped, such as some loop iterations) was applied to the open source C implementation of the h.264 codec in [12]. A speed up of 2-3 times was achieved, with a corresponding reduction in energy consumed by the CPU. More recently, Cartesian GP was applied to an assembly-code implementation of a median-finding algorithm [13], reducing energy consumption in micro-controllers for a small, fixed number of inputs, using execution time as a proxy for energy. Another application considered parameter tuning to find the multi-objective trade-offs between energy and packet loss ratio in Internet-of-Things devices [14]. General trade-offs between energy and performance are discussed by Banerjee *et al.* [3]. Recently, a framework based on a regression model connecting Java source code operations to energy, and code refactoring rules has been proposed [32]: our approach broadly aligns with this framework, although our model is more fine-grained, working at the level of bytecodes.

Transforming the static configuration parameters of an application into variables that can be tuned dynamically has also resulted in programs that consume substantially less energy with little reduction in functional performance [33]. Exhaustive exploration of alternative subtypes of container classes, with replacements substituted into bytecode of Java applications, was shown to reduce energy as measured in hardware [34]. A subsequent study proposing "Object-Oriented Genetic Improvement" using a GA [23] is extended as part of the present paper. Some interesting approaches optimize colour schemes used by applications with the goal of reducing the energy consumption due to the screen of mobile devices rather than the CPU [35], [36].

With the exceptions of [32], [34], none of the above work is concerned with the JVM. Furthermore, most current work applies SBSE to specialised algorithms for niche applications, limiting the potential benefits. In contrast, our work focuses on three very widely-used software components: Quicksort, the Multi-Layer Perceptron and collection classes (such as lists, sets, and maps).

III. OPACITOR

The fitness function used by the evolutionary algorithms in our experiments is the energy consumption as measured using the OPACITOR tool. This is designed to make measurements deterministically. This ensures that results are repeatable, but can also help to reduce experimental time as OPACITOR can distinguish between very similar execution traces without needing repeats for reliability. OPACITOR traces the execution of Java code, using a modified version of OpenJDK 8[2], generating a histogram of executed opcodes. This is combined with a model of the energy cost of each opcode, created and verified by Hao *et al.* [18], by multiplying the number of times an opcode was executed by its cost in Joules. These energy costs per opcode are then summed together, in order to obtain a total energy cost of the execution in Joules.

This opcode model-based approach allows distinctions to be made between very similar programs: this is vital to the effectiveness of the evolutionary process, which is predicated on the ability to create mutant programs very similar to their progenitors [37]. 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 for the final fitness measurements to ensure that the results remain valid under realistic conditions on an unmodified JVM.

[2]http://openjdk.java.net

Algorithm 1: Pseudocode for Quicksort With Variant Pivot Function.

```
double []
qsort (double [] input,
 Function < double [], double >
  pivotFn) {
 double pivot = pivotFn.apply(input);
 // pivotFn can be varied generatively
 return qsort( input.filter( < pivot ),
   pivotFn)
   ++ input.filter( == pivot )
   ++ qsort(input.filter( >
    pivot),pivotFn);
}
```

A significant benefit of OPACITOR, compared to approaches requiring 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. Previous work has successfully used OPACITOR to optimize and reduce the energy consumption of Google Guava's[3] IMMUTABLEMULTIMAP class [23], as well as OpenTripPlanner[4] [38]. In each of these cases, a comparison was made with an alternative timing-based estimation tool JALEN [15] in order to corroborate the assertion that the technique used by OPACITOR is effective and generates reliable measurements.

IV. QUICKSORT

The Quicksort algorithm for in-place sorting [39] has probably accounted for the greatest volume of executions of any sorting algorithm and remains popular [40]. For many years it was the default |qsort| implementation in most 'C' libraries [41]. Quicksort enjoys average case behaviour of $\theta(n \log n)$ but worst case behaviour of $\theta(n^2)$. Crucially, Quicksort's behaviour depends on the 'pivot function': a heuristic to choose the 'pivot' value to partition the input for subsequent divide-and-conquer recursion, as given in Algorithm 1.

A. Implementation

It is well-known that taking the median of the input array provides the Oracular choice of pivot and also yields worst case $\theta(n \log n)$ behaviour, but the associated $\mathcal{O}(n)$ algorithm for

[3]Version 18: https://github.com/google/guava/wiki/Release18
[4]http://www.opentripplanner.org

doing this leads to an unacceptably large overhead. In previous work [22], we used GP to provide an automatically-generated variant of this scheme. This allows the algorithm to be tuned for an input data set, for good performance on unseen data with a matching distribution. The approach uses the TEMPLAR library, a software framework for customising algorithms via the generative technique of the template method. This allows an algorithm to be customised with little development effort by specifying one or more 'variation points', each of which is permitted to express a family of behaviours, constrained merely by the types of its function signature. In this specific case there is a single variation point, the pivot function, which returns the median of r random samples from the input array, with $r = f(l, d)$, where l is input array length and d is the current recursion depth. Here, f is a bivariate rational function generated by GP, i.e. with function set $\{+, -, *\}$ and (protected) divide. As this approach is a hyper-heuristic (i.e. it 'searches the space of heuristics'), we named the algorithm *hyper-quicksort*.

B. Experiments

During evolution, the training set in our experiment consists of the 'pipeorgan' distribution, in which the values in the input array increase monotonically until some randomly-specified index, then decrease monotonically. This distribution can be seen as approximating the worst case behaviour of Quicksort (which arises on pathological distributions such as almost-sorted/reverse-sorted inputs [42]). Quicksort is known to behave poorly against data drawn from this distribution, so this case study could be also considered as an example of 'hardening' software against a denial-of-service attack.

In previous work [22] we used Java's reflection to generate the bytecode of each generated pivot function, so that JALEN could record the execution time and provide an estimate for the energy consumed. OPACITOR measures the energy consumed by running a program in an external JVM, and so this method is not suitable. Instead we generate the full source code of the new pivot function and allow OPACITOR to compile and execute each candidate solution externally.

The earlier results were also limited in that, although showing Quicksort with the evolved pivot function to be more energy efficient when applied to the 'pipeorgan' distribution, they did not test performance on randomly generated arrays. We now address this shortcoming, using OPACITOR to compare the evolved pivot function against commonly-used pivots on both 'pipeorgan' inputs and randomly generated arrays.

The experiments used GP to evolve a pivot function for a program implementing Quicksort, which is run on a target array of numbers. As explained in Section IV-A, the terminals (input variables) for the generated programs are 'array size' and 'recursion depth'. The template for the evolved GP function is configured as follows: the Oracular pivot value of an array of values is its median. GP is then used to generate the function:

$$\mathrm{GP} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathrm{GP} : (\mathrm{arraySize}, \mathrm{recursionDepth}) \mapsto \mathrm{numSamplePoints}$$

and take as pivot the median of $\mathrm{numSamplePoints}$ randomly-chosen array elements. Although Quicksort is defined on anything with a partial order, note that this particular method only works when the array values are numeric.

The experimental setup used GP configured with a population size of 100, 200 generations per run, an initial tree depth of 2 and a maximum tree depth of 4. These values were determined empirically as a reasonable trade-off between solution quality and execution time. All other GP parameters and operators were the EpochX 1.4 defaults [43]. The training set contained 70 cases, where each case consisted of 100 arrays to be sorted, each with size 100. The fitness function to be minimized was the energy used to perform the Quicksort on the training set, using OPACITOR to provide a deterministic, repeatable measurement. Evolution was repeated 30 times to obtain a range of possible pivot functions.

C. Results and Discussion

A number of different functions were generated across different runs of the GP, however commonly-observed among the fitter functions was one that simply returned $\mathrm{recursionDepth}$. This suggests that as the recursion depth increases (and array size decreases), performance is improved by increasing the number of samples. Near the beginning of a sort, with a large array, there is a low probability of randomly selecting suitable elements from which an ideal median may be estimated — as the array size decreases, this probability increases, with the overall effect that maximum recursion depth is reduced.

In Table I, the performance is compared with three well-known pivot functions: 'Middle index', 'Random index', and 'Sedgewick' (the latter returning the median of the first, middle and last elements). In this test, OPACITOR was used to calculate the energy required to sort 1000 'pipeorgan' arrays of varying lengths using each pivot function. This was repeated 100 times, the mean results being shown in Fig. 1, as well as the p-values and effect size measures calculated when comparing the GP result to each of the other pivot functions. Hyper-quicksort outperforms the Sedgewick pivot in all cases, with a significant energy saving. In several cases using the 'Sedgewick' and 'Middle index' pivots the sort resulted in a stack overflow due to the recursion depth. For the smallest arrays (up to 16 elements) hyper-quicksort needs more energy than the random and middle index pivot functions. This is due to the overhead of finding the median value for n randomly selected elements. As array size increases this overhead is mitigated by the associated reduction in recursion depth, allowing hyper-quicksort to outperform all the alternative pivot functions for array sizes of 32 elements or greater.

In order to verify that the results are significant, we used the ASTRAIEA statistical framework [44] to perform the non-parametric Mann-Whitney U-test and the Vargha-Delaney Effect size test. The use of the latter is motivated by a trend within software engineering to augment significance tests for stochastic algorithms with effect size measures, with Vargha-Delaney being specifically recommended for this [45]. For each array length, $2^3 - 2^{18}$, we ran the U-test and effect size test

TABLE I
ENERGY (J) REQUIRED TO SORT 1000 'PIPEORGAN' ARRAYS OF VARYING LENGTHS USING EACH OF THE PIVOT FUNCTIONS (MEAN OF 100 RUNS, AS WELL AS THE P-VALUES (P) AND EFFECT SIZE MEASURES (E) COMPARING THE GP RESULT TO EACH OF THE OTHER PIVOT FUNCTIONS)

Array size	GP		Random				Sedgewick				Middle			
	J	σ	J	σ	p	e	J	σ	p	e	J	σ	p	e
8	2.19	0.18	1.93	0.07	< .001	0.06	5.22	0.09	< .001	1.00	**1.80**	0.02	< .001	0.00
16	2.34	0.10	**2.18**	0.08	< .001	0.10	5.38	0.12	< .001	1.00	2.26	0.09	< .001	0.24
32	**2.34**	0.12	2.43	0.08	< .001	0.77	5.50	0.14	< .001	1.00	2.50	0.14	< .001	0.94
64	**2.51**	0.12	2.75	0.15	< .001	0.95	5.65	0.12	< .001	1.00	2.89	0.06	< .001	1.00
128	**2.60**	0.14	2.93	0.10	< .001	0.98	5.80	0.13	< .001	1.00	3.10	0.11	< .001	1.00
256	**2.69**	0.14	3.23	0.15	< .001	1.00	5.97	0.15	< .001	1.00	3.46	0.17	< .001	1.00
512	**2.93**	0.16	3.71	0.26	< .001	1.00	6.40	0.10	< .001	1.00	4.43	0.53	< .001	1.00
1024	**3.69**	0.57	4.26	0.70	0.01	0.71	6.68	0.13	< .001	1.00	5.53	0.64	< .001	0.96
2048	**3.60**	0.52	4.78	1.21	< .001	0.88	6.78	0.10	< .001	1.00	6.36	1.16	< .001	0.98
4096	**3.37**	0.27	6.01	1.72	< .001	1.00	7.31	1.39	< .001	1.00	6.96	1.88	< .001	1.00
8192	**3.72**	0.31	6.09	1.36	< .001	1.00	7.20	0.19	< .001	1.00	5.64	0.78	< .001	1.00
16384	**4.58**	0.39	8.18	1.71	< .001	0.99	8.07	0.20	< .001	1.00	9.61	1.39	< .001	1.00
32768	**6.11**	0.97	18.99	8.41	< .001	1.00	-	-	-	-	20.37	2.88	< .001	1.00
65536	**8.72**	2.56	24.22	10.46	< .001	0.95	-	-	-	-	12.88	6.62	< .001	0.82
131072	**14.67**	5.21	23.06	13.38	0.05	0.65	-	-	-	-	-	-	-	-
262144	**13.74**	4.60	24.21	12.55	< .001	0.73	-	-	-	-	-	-	-	-

In a number of cases the sort resulted in a stack overflow, these results are therefore excluded from the table. Bold values highlight the lowest energy use for each array size.

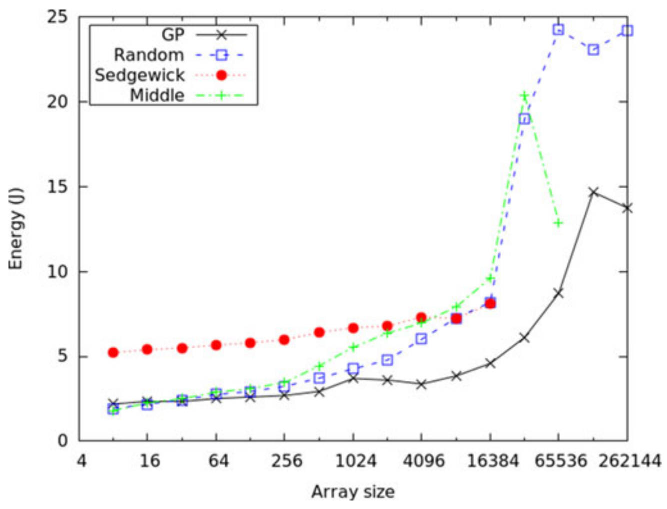


Fig. 1. Energy (J) required to sort 1000 'pipeorgan' arrays of varying lengths using each of the pivot functions. The lines with no points after a particular array size show where the sort was unsuccessful (recursion limit reached).

comparing the set of results obtained using hyper-quicksort to each of the alternative pivots in turn. In all cases the distributions of the results were significantly different ($p < 0.05$). According to Vargha and Delaney [46], an effect size of 0.56 indicates a small difference, 0.64 indicates a medium difference, and 0.71 indicates a large difference. Given this scale, in almost all cases the evolved pivot function has provided a large energy reduction.

Further than this, as the array size was increased to 32,768, the recursive algorithm caused a stack overflow in the JVM when using the Sedgewick pivot function. This also occurred with array sizes of 2^{17} when using the Middle Index pivot function. It is possible to increase the JVM's stack, however this serves to emphasise that hyper-quicksort has helped to harden the algorithm against a denial-of-service attack.

To determine how the same evolved pivot function performs on non-'pipeorgan' arrays, similar testing was performed—this time calculating the energy required to sort 1000 randomly generated arrays of varying lengths. This test was also run 100 times, with the mean results presented in Table II and Fig. 2, as well as the p-values and effect size measures when comparing the GP result to each of the other pivot functions. Notably, although hyper-quicksort was trained specifically to sort 'pipeorgan' input arrays, the evolved pivot function is also very successful at sorting randomly generated input arrays.

Except for the Sedgewick pivot function, which was outperformed by hyper-quicksort for all array sizes, the pivot functions are statistically inseparable with an array size of less than 4096 elements. At this point, the evolved pivot function is consistently more energy efficient than the alternatives, with $p < 0.05$ and in general a large effect size.

Although it might be expected that the energy required to sort an array will increase with array size, as the arrays in both experiments were randomly generated (either randomly selecting the point at which the value begins decreasing, or generating all the data at random) this is not guaranteed. All four pivot functions sorted the same arrays, but it is not possible to ensure the 'difficulty' of sorting each input array is the same across all array sizes. As such, some otherwise unexplained outliers exist, for example the middle index pivot function sorting arrays of size 65,536 displays a significant decrease in energy required compared to size 32,768 where the other pivot functions all show an expected increase.

These experiments also demonstrate that energy use is dependent on the dataset that the final algorithm is applied to. Of course, this is hardly surprising, but what is important here is that deploying SBSE to the task of tuning an algorithm for a particular distribution of data relieves the developer from having to consider these issues explicitly. As long as the developer selects a representative collection of data to tune the algorithm

TABLE II
ENERGY (J) REQUIRED TO SORT 1000 RANDOMLY GENERATED ARRAYS OF VARYING LENGTHS USING EACH OF THE PIVOT FUNCTIONS (MEAN OF 100 RUNS, AS WELL AS THE P-VALUES (P) AND EFFECT SIZE MEASURES (E) COMPARING THE GP RESULT TO EACH OF THE OTHER PIVOT FUNCTIONS)

Array size	GP		Random				Sedgewick				Middle			
	J	σ	J	σ	p	e	J	σ	p	e	J	σ	p	e
8	2.06	0.20	1.83	0.06	$<.001$	0.15	5.20	0.09	$<.001$	1.00	**1.81**	0.07	$<.001$	0.14
16	2.27	0.14	**1.99**	0.08	$<.001$	0.04	5.28	0.11	$<.001$	1.00	**1.99**	0.11	$<.001$	0.06
32	2.27	0.11	2.21	0.10	0.01	0.30	5.38	0.13	$<.001$	1.00	**2.19**	0.06	$<.001$	0.20
64	**2.33**	0.06	2.42	0.14	$<.001$	0.83	5.45	0.07	$<.001$	1.00	2.40	0.09	$<.001$	0.79
128	**2.44**	0.09	2.62	0.17	$<.001$	0.95	5.66	0.09	$<.001$	1.00	2.58	0.11	$<.001$	0.90
256	**2.58**	0.16	2.81	0.14	$<.001$	0.90	5.90	0.11	$<.001$	1.00	2.88	0.17	$<.001$	0.97
512	**2.98**	0.26	3.31	0.06	$<.001$	0.90	6.24	0.12	$<.001$	1.00	3.28	0.08	$<.001$	0.90
1024	4.30	1.41	3.82	0.53	0.09	0.37	6.41	0.12	$<.001$	0.96	**3.64**	0.07	0.01	0.31
2048	4.30	2.18	**3.94**	0.40	0.05	0.65	6.58	0.14	$<.001$	0.89	4.00	0.50	0.02	0.68
4096	**3.36**	0.27	5.24	1.18	$<.001$	0.98	6.71	0.13	$<.001$	1.00	5.15	0.86	$<.001$	0.98
8192	**3.72**	0.31	6.09	1.36	$<.001$	1.00	7.20	0.19	$<.001$	1.00	5.64	0.78	$<.001$	1.00
16384	**4.45**	0.59	7.05	1.04	$<.001$	0.98	7.90	0.31	$<.001$	1.00	6.63	0.92	$<.001$	0.96
32768	**5.88**	0.91	14.96	7.59	$<.001$	0.96	9.50	0.95	$<.001$	1.00	13.09	6.43	$<.001$	0.92
65536	**9.52**	3.18	14.81	7.62	$<.001$	0.75	14.55	3.36	$<.001$	0.84	14.78	6.61	$<.001$	0.76
131072	**13.12**	4.35	21.18	11.44	0.03	0.67	16.98	2.60	$<.001$	0.72	18.82	8.01	$<.001$	0.73
262144	**10.36**	3.41	20.75	10.37	$<.001$	0.90	16.14	3.74	$<.001$	0.89	18.50	7.61	$<.001$	0.89

Bold values highlight the lowest energy use for each array size.

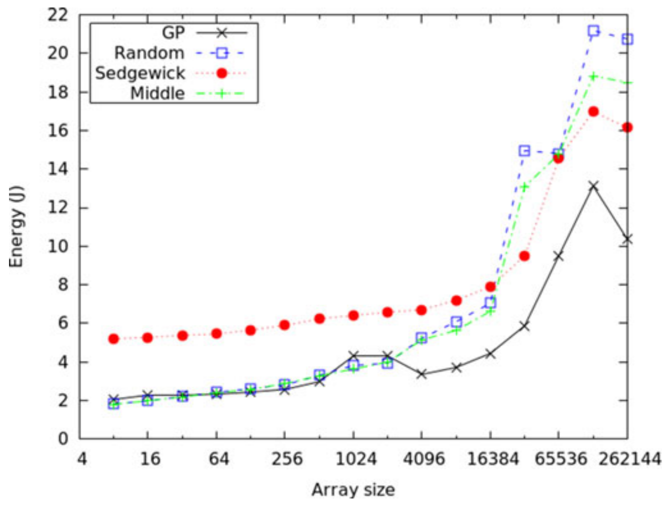


Fig. 2. Energy (J) required to sort 1000 randomly generated arrays of varying lengths using each of the pivot functions.

for, the search process will take care of the rest. The algorithm can then be easily retuned for new target datasets.

V. MULTILAYER PERCEPTRON

We now consider application to the training and validation phases of a very popular [47] neural net architecture — the Multi-Layer Perceptron (MLP). The MLP is a modification of the standard linear perceptron with 'hidden' layers of neurons with nonlinear activation functions. can distinguish data that are not linearly separable and are often applied to classification tasks like image recognition. They have seen renewed interest recently with the advent of deep learning.

We focus on two objectives for optimization. The first is the energy use ϵ, of the training and validation phases of the MLP, as determined by OPACITOR. The second is the mean square error E of the trained MLP on validation data. The goal is to realise Harman *et al.*'s Pareto Program Surface [48], comprising program variants spanning the trade-off between a functional property of the software, in this case error rate, against a non-functional property like energy. It can be envisaged that the user of a mobile device might trade-off some accuracy in an app using classification if it would extend battery life. Here we show how a multi-objective metaheuristic can easily explore this trade-off.

MLPs consist of input and output layers, with one or more 'hidden' layers. The multiple inputs to each neuron are subject to thresholding via an *activation* function — typically a sigmoidal expression in terms of $\tanh(x)$ or e^x. It is well-known that a MLP with only a single hidden layer is capable of universal function approximation [49] and this, together with relative ease of use for practitioners, has led to decades of prevalence in machine learning for the MLP, both in academia and industry.

The operation of back-propagation is dependent on the ability to take the derivative of the activation function (AF). Historically, the emphasis was on fixed AFs with well-known derivatives. Recently, interest has grown in alternative AFs [50], which has motivated the application of *Automatic Differentiation* (AD) to obtain their derivatives. One means of performing AD is via the use of *dual numbers* [51]. Analogous to the extension of the reals to the complex plane, in which every real number x is generalized as $x + \mathrm{i}\,y$, dual numbers are represented as $x + \varepsilon y$, where ε is an *infinitisimal* (which can be considered as a matrix expression, rather than a real number) with $\varepsilon^2 = 0$ [52]. By changing the implementation of any smooth real-valued function f to instead use dual numbers, it is possible to automatically obtain derivatives of f from the initial terms of its Taylor series expansion. AD has the same computational complexity as f and is not subject to the roundoff error of numerical approximations to the gradient via finite differences, nor does it suffer from

TABLE III
PARAMETERS OF MLP VARIED DURING OPTIMIZATION

Param.	Explanation	Type	Range (inclusive)
a	AF parameter 0	Continuous	$(0, 1)$
b	AF parameter 1	Continuous	$(0, 1)$
c	AF parameter 2	Continuous	$(0, 1)$
d	AF parameter 3	Continuous	$(0, 1)$
h	Neuron count in hidden layer	Integer	$(2, 30)$
l	Backpropogation learning rate	Continuous	$(0, 1)$
α	Backpropogation momentum	Continuous	$(0, 1)$

the expression bloat of symbolic methods. In this section, we describe the novel application of AD to the online generation of AFs in order to explore the tradeoff between accuracy and energy efficiency.

A. Implementation

A wide range of alternative AFs have been explored in the literature, including logistic, Gaussian (e^{-x^2}) trigonometric and radial basis functions [53]. Previous work on online adaptation of a piecewise linear AFs can be found in Agostinelli *et al.* [54]. The AF we use here is a parametric generalization of the popular logistic function $1/(1+\mathrm{e}^{-x})$, given by:

$$f(x) = \frac{a}{1 + b\,\mathrm{e}^{cx}} + d \tag{1}$$

The real-valued parameters $\{a, b, c, d\}$ define the family of AFs to be explored by search. AD has two operation modes, 'forward' and 'reverse'. The latter is more efficient for machine learning scenarios such as we are concerned with here, in which the derivative of the AF is required for multiple fitness cases at each fitness evaluation. Hence, the derivative of the AF was obtained via the application of reverse mode AD.

Three parameters controlling training and topology are also included, so a particular network is specified by the 7 parameter values in Table III. For each dataset, instances were divided at random into 2/3 training and 1/3 validation data.

As we have already seen, ϵ varies depending on a wide range of factors. E varies depending on the seed for the random number generator (used to set MLP starting weights) and the specific training/validation datasets chosen. This means that both objectives are noisy, so evaluations were repeated 5 times with the mean value used to guide the search (the 5 repeats counted as one *evaluation* in the experiments).

B. Experiments

The popular multi-objective evolutionary algorithm NSGA-II [55] as implemented in JMetal 4.5 [56] was used to conduct the search. For all data sets, the parameters were the same: population size 10; termination after 400 evaluations; SBX crossover with rate 0.9 and distribution index of 20.0; polynomial real mutation with rate 1/7 and distribution index of 20.0; and binary tournament selection.

The experiments covered 4 UCI datasets (Table IV) included with the WEKA machine learning library [57]. The

TABLE IV
UCI DATASETS USED

Dataset	Instances	Attributes
Pima Indians Diabetes	768	8
Glass Identification	214	10
Ionosphere	351	34
Iris	150	4

metaheuristic searched for solutions having specific values for each of the 7 parameters in Table III, seeking to minimise ϵ and E objectives for a given training dataset. 'Error' in the results is the mean square error for classification on the validation dataset. Two separate runs were performed for each dataset, measuring energy for:

1) t: training only.
2) tv: training and 1,000 repeats of the validation phase.

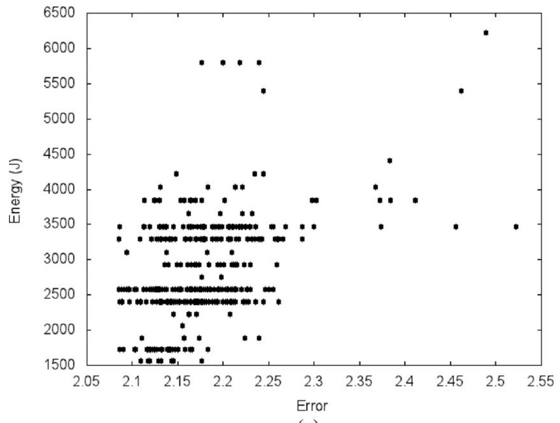
In the latter, the validation phase was repeated 1000 times so that it would impact on total energy, because its run time is much smaller than the training phase. In practice, training will be performed far less frequently than validation/classification, so energy gains for the latter will be of most benefit. These experiments are focused on the trade-off between objectives in terms of the application, rather than comparison of performances, so no repeat optimisation runs were performed. Statistics related to the repeats of the objective function are reported in the next section.

C. Results and Discussion

Fig. 3 shows all 400 solutions visited by the search on the Glass data set, plotted in the objectives (energy vs. error). Plots for the other three data sets have been omitted here to save space, but showed similar distributions of the solutions. For most data sets, there is little conflict between energy and error, so it is possible to minimize both. Energy is reduced to around 20% of the worse-case solutions for all data sets. For Glass, when measuring energy for training only, there is a Pareto front, albeit limited to two solutions. Energy use can be reduced from 1724 J to 1559 J (about 10%) by accepting an increase in mean square error from 2.08 to 2.11. All the results show essentially discrete steps in energy use, corresponding to the number of hidden layer neurons h. However, the search process is worthwhile because there is not always a simple linear relationship between h and energy. This was particularly true for the training phase on Diabetes, Glass and Ionosphere (Fig. 4 shows the relationship for Ionosphere).

Analysis of values taken by the variables in the optimal and near-optimal solutions found that a, d and α took similar values for both tv and t runs, but differed per dataset. In contrast, b, c and l varied depending on both dataset and t/tv.

The variation between repeat runs within evaluations was notably higher for the error objective than for energy. We considered the coefficient of variation (cov – the ratio of standard deviation to the mean) for the 5 repeats within each of the 400 evaluations in each run. Energy measurements were consistent: $cov(\epsilon)$ was never higher than 0.01% for all evaluations. In


(a)

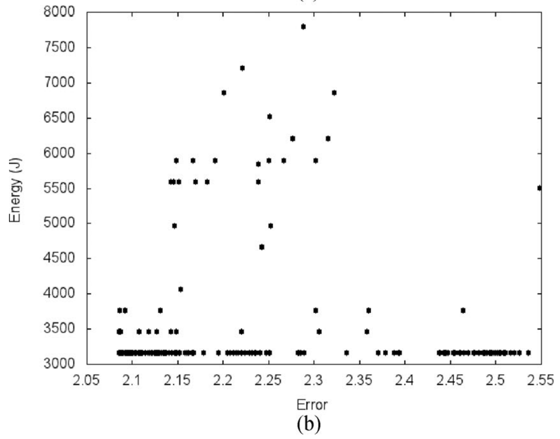

(b)

Fig. 3. All evaluations for the Glass dataset, plotted in objective space: energy (J) vs. mean square error. “t” are runs where the energy measurement included training only, “tv” runs measured energy for both training and validation. Diabetes, Ionosphere and Iris showed similar distributions. (a) Glass training. (b) Glass training & validation.

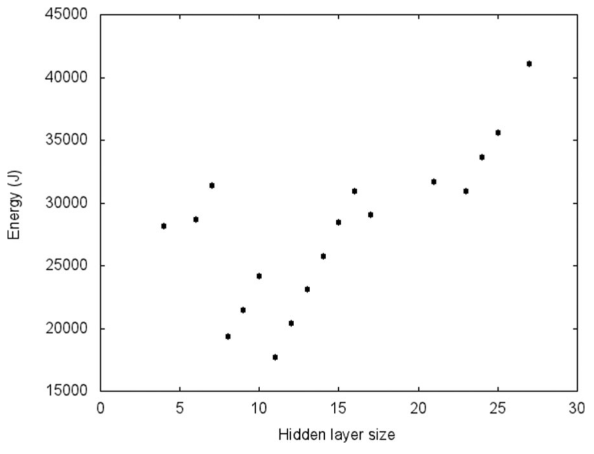


Fig. 4. Energy (J) vs. hidden layer size for Ionosphere, measuring energy for the training stage only.

contrast, $cov(E)$ for error varied from zero to 42%, with a mean of 2%. Applying t-tests to the objective values found $p \leqslant 0.005$ for the difference in means between the original and optimal configurations on all energy runs. For error, $p < 0.05$ with diabetes-v, glass-v, glass-nv, and iris-nv. Differences in error were non-significant for the other runs.

These experiments have demonstrated that SBSE allows the developer to easily explore the trade-off between functional and non-functional properties (in this case, accuracy and energy). Such relationships are not always clear: here, one might have expected there to be a conflict between the objectives, yet often there was none. Moving the burden of exploring such a relationship to an automated search means that it becomes much simpler to tune the balance between these objectives for a particular situation. In this case, the approach achieved the ideal result of minimising energy and error for the MLP on four well-known datasets. This could be fine-tuned per-dataset and with a bias towards improvement in the training or validation/classification phases, depending on end application. It would allow multiple configurations to be generated and stored that could be switched on-the-fly: for example, on a mobile device so that classification accuracy decreases as battery charge is exhausted.

VI. Guava and Apache Commons Collections

We now consider Object-Oriented Genetic Improvement (OO-GI), a technique for improvement specific to Object-Oriented programs, applied to optimization of container classes. These are implementations of well-known data structures like maps, sets, and lists. The choice of implementations for collections classes dramatically impact on energy consumption [58]. Our approach uses metaheuristic search to find semantically equivalent programs with reduced energy consumption. Semantics-preserving transformations are achieved by conformance to the behavioural equivalence that is central to object-orientation. OO-GI was first proposed in [23], and demonstrated using an earlier version of Opacitor, as applied to a single class in the Google Guava framework. This earlier work is here extended with a more thorough experimental study, covering a range of classes in Google Guava[5] and the Apache Commons Collections library.[6]

Although several existing approaches in GP and Grammatical Evolution (GE) claim to be ‘Object Oriented’ [59]–[63], we are not aware of any concerned with the central pillar of Object Orientation, the Liskov Substitution Principle (LSP) [64] as exemplified via subtype polymorphism. When designed according to the LSP, subtype polymorphism permits the implementation of frameworks according to the ‘Open Closed Principle’, i.e. both framework implementers and their client programmers can be assured that their code can interoperate without unexpected behaviour. It is this principle that allows OO software to scale more effectively than purely object-based approaches. ‘Design by contract’ (DbC) is the explicit codification of the LSP via preconditions, postconditions and invariants [65]. DbC is widely understood to be a vital adjunct to methods such as Test-Driven Development, since the above conditions serve to capture a wide range of required behaviours. It was first natively supported in Meyer’s Eiffel language in 1986 and is now natively supported by a number of popular programming languages, including ADA 2012, Clojure, D and Racket. The closest search-based work of

[5]Version 18: github.com/google/guava/wiki/Release18
[6]V4.0: commons.apache.org/proper/commons-collections/release_4_0.html

TABLE V
CLASSES CHOSEN FOR MODIFICATION BY OO-GI, WITH THE NUMBER OF VARIATION POINTS AND SIZE OF THE RESULTING SEARCH SPACE FOR EACH

Class for modification	Variation points	Search space
`com.google.common.collect.ArrayListMultimap`	3	2 738
`com.google.common.collect.ImmutableMultimap`	5	674 325
`com.google.common.collect.LinkedListMultimap`	6	7 513 072
`org.apache.commons.collections4.map.PassiveExpiringMap`	3	50 653
`org.apache.commons.collections4.set.ListOrderedSet`	4	38 416
`org.apache.commons.collections4.bidimap.DualHashBidiMap`	6	2 565 726 409

which we are aware is the SEEDS framework [34], in which alternative subtypes of container classes are substituted into bytecode in order to minimize power consumption. SEEDS uses separate exhaustive search at each object allocation location. In contrast, OO-GI uses a genetic approach, applicable to any code following DbC, to assign subclasses to all constructor invocations simultaneously. The benefit is to catch any situation where dependencies exist between variation points. It is difficult to determine when this might be the case by simple code inspection, meaning that simply combining results from separate searches at each variation point may be suboptimal.

A. Implementation

The improvement process is as follows:

1) Parse the source file to be improved into an Abstract Syntax Tree (AST). Identify *variation points*: i.e., source nodes in the AST corresponding to the creation of container classes from Java 8 `util`, Guava, or Apache collections. These comprise three types of syntax fragment: constructors (e.g. `new HashMap<>()`); factory classes (e.g. `Maps.newHashMap()`); and static creator methods (e.g. `ImmutableList.of()`).
2) Obtain the complete set of possible target substitutions T (i.e. all container classes in Guava, Apache and Java Collections. For each variation point S_i in the AST, determine the most specific abstract supertype of the created object, by checking whether it inherits from one of the following (always choosing the most specific type - the one appearing earliest in the list):
    a) `java.util.SortedMap`
    b) `java.util.Map`
    c) `java.util.SortedSet`
    d) `java.util.Set`
    e) `java.util.List`
    f) `com.google.common.collect.Multimap`
    g) `com.google.common.collect.Multiset`
    h) `java.util.Collection`
3) For each S_i, find the subset of possible target substitutions $t(S_i) \subseteq T$ which are valid (i.e., are concrete, and implement the same abstract supertype as the original class at S_i).
4) Given a sequence of the k variation points $[S_1, \ldots, S_k]$ from the AST, the search space is given by all combinations from $[t(S_1), \ldots, t(S_k)]$. The solution representation is 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 $0 \leq r_i < |t(S_i)|$.
5) Given such an assignment, the AST node for each S_i is replaced with its target substitution and the resulting mutated source file is written to disk.
6) The program containing the mutated source is compiled and its energy consumption evaluated by OPACITOR. By this means, combinatorial search is performed in the space of these representations by a GA to find the substitutions that minimize energy consumption.

Source file parsing and subsequent re-generation was completed automatically with an open-source library `com.github.javaparser.JavaParser`. A modified version of JavaParser's `SourcePrinter` made the required source code substitution at each variation point.

A careless programmer might depend on functionality not guaranteed by the LSP, such as assuming that a collection is sorted, despite this not being part of the superclass type specification actually visible at the point at which this assumption is made. This is a logical error by the programmer and, strictly speaking, requires rewriting of the code. We do not cater explicitly for such programming errors in our experiments. In practice, the best that can be done is to include, as we do, any existing unit tests as a constraint on the search.

B. Experiments

The open-source Java frameworks Google Guava and Apache Commons Collections implement 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 $\mathcal{O}(1)$.

An exhaustive search over all top-level concrete collections classes in Google Guava and Apache Commons Collections found 14 with 3 or more variation points. After ruling out those where some variation points were instantiations of the class itself, or used specialised classes where functionality would likely be broken by replacement with a generic container, 6 classes were chosen for closer consideration (Table V).

Energy was measured for each class over 10,000 repeat runs of a test program. This called all unit tests (from the Google Guava or Apache Commons source trees as appropriate) for the class. In addition, because (in some cases) the unit tests did not exercise all methods, a call was also made to every method in

TABLE VI
ENERGY (J) REQUIRED TO EXERCISE THE UNIT TESTS FOR EACH CLASS 10,000 TIMES (MEAN OF 100 RUNS, AND STANDARD DEVIATION σ), MEASURED BY OPACITOR WITHOUT (EXCL.) AND WITH (INCL.) JIT AND GC

Target class	Measure	GA		Original			Independent Exhaustive		
		J	evals	J	p	e	J	p	e
`ArrayListMultimap`	excl.	**260.70**	2455 (10.34%)	260.90	–	–	**260.70**	–	–
	incl.	**21.23** σ1.00		23.39 σ1.09	<.001	0.92	**21.23** σ1.00	–	–
`ImmutableMultimap`	excl.	**224.70**	10 035 (98.5%)	300.70	–	–	275.26	–	–
	incl.	**12.22** σ0.73		15.75 σ1.79	<.001	0.97	13.09 σ1.09	<.001	0.74
`LinkedListMultimap`	excl.	**370.28**	18 092 (99.76%)	370.29	–	–	370.29	–	–
	incl.	**19.82** σ1.25		32.95 σ2.79	<.001	1.00	26.11 σ0.82	<.001	1.00
`PassiveExpiringMap`	excl.	**108.34**	11 014 (78.26%)	193.60	–	–	158.94	–	–
	incl.	**11.54** σ1.52		17.06 σ0.65	<.001	1.00	14.74 σ1.37	<.001	0.94
`ListOrderedSet`	excl.	**70.83**	8470 (77.95%)	70.84	–	–	**70.83**	–	–
	incl.	**17.50** σ1.71		22.96 σ2.53	<.001	0.95	**17.50** σ1.73	–	–
`DualHashBidiMap` (best case)	excl.	**133.22**	23 646 (99.99%)	133.73	–	–	133.49	–	–
	incl.	**10.23** σ1.05		16.56 σ0.90	<.001	1.00	11.61 σ0.90	<.001	0.84
`DualHashBidiMap` (worst case)	excl.	**133.38**	23 646 (99.99%)	133.73	–	–	133.49	–	–
	incl.	**10.99** σ1.14		16.56 σ0.90	<.001	1.00	11.61 σ0.90	<.001	0.68

The p-values (p) and effect sizes (e) compare the GA result to the original or the result of IES (in two cases the GA and IES found the same set of substitutions). Also given are the number of evaluations (evals) required by the GA to reach the best fitness found, and the percentage reduction that this represents compared to an exhaustive search of the entire space. Bold values highlight the lowest energy use for each class.

the class using randomly-generated data. Programs failing the unit tests were rejected by the search as invalid.

The first experiment used a GA to search for a solution with reduced energy consumption. The Apache Commons Math library[7] GA implementation was used. The solution representation used was a vector of integers $r \in \mathbb{Z}^k$. The representation itself was constrained such that $0 \leq r_i < |t(S_i)|$. The GA parameters were chosen empirically to yield good results in reasonable run time (evaluations taking 5–30 s on a 3.25 GHz CPU). These were: population size 500; termination at 100 generations; single-point crossover with a rate of 75%; one-point mutation with a rate of 50%; binary tournament selection; 5% elitism. 30 repeat runs using different random number generator seeds were completed for each target class.

A second experiment ran an exhaustive search on the entire search space for each class (except `LinkedListMultimap` and `DualHashBidiMap` due to their large size) to determine how close the GA came to finding the optimal solution.

A final experiment ran an exhaustive search on each variation point, following the example of [34] but on source code instead of bytecode. Each variation point is examined in turn, finding the best substitution independently of the other variation points. The set of candidate results then constitutes one program for each of these separate substitutions. Added to this set of candidates is a program containing all of the independent substitutions combined together, and the original program. The program from this set requiring the least energy is the final result—this may not be the combined candidate, if the variation points are not in fact independent. We refer to this experiment as an *Independent Exhaustive Search* (IES).

C. Results and Discussion

Statistical testing was carried out using the ASTRAIEA [44] Wilcoxon/Mann Whitney U and Vargha-Delaney Effect size tests. Results were obtained with 100 samples in each dataset. In all cases except `DualHashBidiMap`, each GA run found an identical result, so this solution was used thereafter. For `DualHashBidiMap`, due to the large search space, 3 different results were found. The best and worst were each compared statistically to the original code and the result from IES.

For the four classes where an exhaustive search was performed, the best set of substitutions were the same as those found by the GA. The GA therefore reduced the search time by between 10% (`ArrayListMultimap`) and 98.5% (`ImmutableMultimap`). Using alternative parameters for the evolution would likely improve this further—particularly in the cases with the smallest search spaces.

The results are shown in Table VI. The measurements using OPACITOR without JIT and GC are deterministic and so no

[7]commons.apache.org/proper/commons-math/userguide/genetics.html

statistical tests are necessary. When using OPACITOR with JIT and GC enabled, the fitness was evaluated 100 times—the table shows the mean and standard deviation, as well as the p-values and Vargha-Delaney effect size measures comparing the results of the GA to the original and the results of IES.

Reductions in energy consumption were achieved for all six classes. An interesting result is that, without JIT and GC, the best result found by the GA for four of the classes was only marginally better than the original programs. Once JIT and GC were enabled for final testing the improvement becomes significant with a medium or large effect size. Deeper analysis revealed that, in these cases, the substitutions included a class from an alternative collection (Google Guava or Apache Commons) to the original class. This results in a large number of additional classes which must be imported by the JVM. When JIT is re-enabled the importing is optimized, resulting in a significant energy reduction compared to the original.

As the energy measurement during the final testing with JIT and GC is not deterministic, p-values and effect sizes vary accordingly. The p-values are all near-zero, with effect sizes all at least 0.74, which Vargha and Delaney suggest indicates a large difference. Although in some cases the improvement was marginal when measured with JIT and GC disabled, in the more realistic scenario of having these enabled, 2.16–13.13J were saved over the original implementations. These represent energy reductions of 9.23%–39.85% over the original.

During final testing, the GA also matched or further improved upon the lowest-energy solutions found by IES. This reflects the presence of some dependency between the variation points for some of the classes: if the points were independent then IES would find the optimal solution by combining the best value for each variation point. The possibility of such dependencies increases the search space to the product of all values at all variation points, necessitating a global search like the GA. For example, in `ImmutableMultimap` one variation point is within a nested subclass that is privately declared, which is instantiated at another variation point. IES requires a number of fitness evaluations approximately equal to the count of possible substitutions over each of the variation points, and so is significantly faster than the GA, however this speed is a trade-off with the quality of the final result. This highlights that there can be subtle interactions between sections of the code that influence energy consumption and may not be obvious to a developer. An SBSE approach is able to accommodate these interactions where they exist without the developer needing to specify where they are, achieving better performance without additional developer effort.

VII. THREATS TO VALIDITY

There are several potential threats to the validity of this work. The results are dependent on the accuracy of the model: while this cannot be completely eliminated, two factors mitigate this threat. The bytecode level estimates of energy use were taken from Hao *et al.* [18], where hardware measurements found the model to be within 10% of the ground truth for a set of mobile applications from the Google Play store. In our previous work [23], [38], we also validated the model by comparing against run-time based models of energy use. Additionally, in practice energy use is likely to follow a conditional distribution rather than remaining constant per-bytecode. A stochastic model is a major consideration for future study on this topic. In particular, (as per [66]) energy consumption is affected by contextual factors such as opcode prefetching and cache hits. There is a clear tradeoff between the generality of a static model such as used by OPACITOR and the device-specific labour required for dynamic measurement. It would however be possible to extend the static model with further contextual features. For example, the current model can be seen as a Markov chain of order 1, in which the energy cost of each opcode is independent. Another useful direction for future work is to use machine learning to build predictive models with more contextual features.

Since OPACITOR is implemented for the JVM, our major results are limited to that platform. Despite the popularity of Java, we cannot be sure that the results obtained would be reproducible in another context.

The general conclusions of our results are supported by the three examples to which we have applied SBSE. While the approaches may seem application-specific, they are simply specific examples of general techniques. The search-based generation of a custom Quicksort pivot function was rapidly prototyped (via the generic method of 'template method hyper-heuristics' [67]) using an existing publicly-available library TEMPLAR. The only application-specific aspect was selection of the variation point (the pivot function) and its signature definition, with GP doing the rest. The HyperMLP implementation is a specific instance of hyperparameter tuning, in this case determining the network topology and (via our application of automatic differentiation) the specific activation functions in the network. OO-GI applies to any well-designed OO system, i.e. those featuring multiple concrete subclasses. Further applications of the tools we have developed will doubtless help improve confidence in the general applicability for SBSE in tuning to data sets, trading off functional and non-function properties, and dealing with dependencies with little developer intervention.

In all data-driven applications, energy consumption might be expected to vary between runs, dependent on the specific execution paths followed. Thus, a threat lies in the choice of scenarios made for our experiments. We have tried to mitigate this where possible (e.g. Quicksort experiments explicitly include pathological inputs; we have many repeats of classification for the MLP). With OO-GI, the unit tests are targeted towards code coverage rather than typical use: in practice it would be important to target a series of tests closely aligned to envisaged usage patterns.

In a similar way, while we have used existing unit tests and public benchmark data, a potential threat lies in the selection of the particular data we considered. We could look at still larger array sizes for Quicksort, some of the larger UCI data sets for MLP, or different test scenarios for the collections classes. Practicality requires that we draw the line somewhere: with considerable additional computational cost, wider data sets of varying scales could be considered to reduce this threat.

VIII. CONCLUSION

Given the widespread use of Java, it is surprising that so little has been done to to apply the power of search-based software engineering (SBSE) to minimize the energy consumption programs written for the Java Virtual Machine (JVM). We presented OPACITOR, a new tool to measure the energy consumption of programs running on the JVM and used three different search based techniques to optimize the energy efficiency of Quicksort, Multi-Layer Perceptrons and popular suites of collection classes. OPACITOR has several advantages over time-based energy approximations or hardware measurements. It is deterministic: it will always produce the same energy measurement for a given program's compiled bytecode. This avoids the need for averaging across multiple runs that is required by other energy measurement approaches. It is unaffected by the rest of the computing environment: the energy measurements are not influenced by other applications that are running. This means that optimization runs making use of OPACITOR can be distributed across multiple threads or CPUs, in parallel on the same machine, without inadvertently affecting the results. The consistency in the energy measurements and the bytecode-level model used also mean that OPACITOR is able to detect small changes in execution profile, right down to the level of single opcodes. These characteristics make the energy measurements of OPACITOR highly amenable to metaheuristic search. We demonstrated how the energy measurements from OPACITOR can be used by three different SBSE approaches to achieve substantial energy savings for three widely-used software components:

1) GP was used to find better pivot functions for the Quicksort algorithm. Energy reductions of up to 70% were achieved against common alternative pivot functions.
2) Multi-objective hyperparameter search was applied to Multi-Layer Perceptrons, incorporating dynamically-generated activation functions via automatic differentiation. For four benchmark datasets, we found a MLP configuration that minimized both energy consumption and mean square error.
3) Object-Oriented Genetic Improvement was applied to six container classes in the Google Guava and Apache Commons libraries. An energy reduction of up to 39.85% was achieved against original code.

Existing SBSE approaches have mainly tackled only specialised applications. The first two approaches we describe, are applicable to any language, and are specific examples of generic methods (application of the template method, and hyperparameter tuning). The third is applicable to *any* code developed in accordance with 'Design by Contract', further widening applicability. This work demonstrates that SBSE is not only suitable for the specialist applications already tackled in the literature, but also much more frequently used code, where it will have a much greater real-world impact. We have shown that the metaheuristics we applied in each setting were able to match or better alternative methods: furthermore in general metaheuristics can be expected to scale to larger problems better than alternative search methods, with good results being reported for very large problems indeed [68].

The work has also highlighted the strength of SBSE in accommodating three aspects of energy optimisation without extensive developer effort. All that is required is the identification of variation points in the code and a suitable representation for possible replacements. Algorithms can be tuned to specific distributions of data; energy use can be traded off against other functionality; and the search can handle subtle interactions within the code that might otherwise be missed.

New directions for future research have been opened up. It would be interesting to explore alternative applications for OOGI, particularly where several public libraries implementing common interfaces exist. The model could also be extended to consider uncertainty in the energy attributed to each bytecode, and consider contextual features such as opcode prefetching and cache hits for improved accuracy.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their helpful feedback and suggestions.

Data Access Statement: This work made use of several existing libraries and data sets, freely available at the locations cited. Data generated in this work is available at http://hdl.handle.net/11667/88.

REFERENCES

[1] P. Marks, "Let's cut them some slack," *New Scientist*, vol. 230, no. 3067, pp. 36–39, 2016.

[2] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proc. Genetic Evol. Comput. Conf.* Madrid, Spain: ACM, 2015, pp. 1327–1334.

[3] A. Banerjee and A. Roychoudhury, "Future of mobile software for smartphones and drones: Energy and performance," in *Proc. IEEE/ACM Int. Conf. Mobile Softw. Eng. Syst.* Buenos Aires, Argentina, 2017, https://www.comp.nus.edu.sg/∼abhik/pdf/MOBILESoft17.pdf

[4] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. USENIX Tech. Conf.* Berkeley, CA, USA, 2010, p. 21.

[5] A. Carroll and G. Heiser, "The systems hacker's guide to the Galaxy: Energy usage in a modern smartphone," in *Proc. 4th Asia-Pacific Workshop Syst.* New York, NY, USA: ACM, 2013, pp. 5:1–5:7.

[6] J. G. Koomey, "Growth in data center electricity use 2005 to 2010, USA, CA, Oakland: Analytics Press, Jul. 2011.

[7] A. Hindle, "Green software engineering: the curse of methodology," *PeerJ PrePrints*, vol. 3, 2015, Art. no. e1832.

[8] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.* NY, USA: ACM, 2014, pp. 36:1–36:10.

[9] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," *Empirical Software Engineering and Verification*. Berlin, Germany: Springer-Verlag, 2012, pp. 1–59.

[10] D. R. White, "Genetic programming for low-resource systems," Ph.D. dissertation, Computer Science, University of York, York, U.K., 2009.

[11] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *Proc. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: ACM, 2014, pp. 639–652.

[12] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2009-042, 2009.

[13] V. Mrazek, Z. Vasicek, and L. Sekanina, "Evolutionary approximation of software for embedded systems," in *Proc. Companion Proc. Genetic Evol. Comput. Conf.* Madrid, Spain: ACM, 2015, pp. 795–801.

[14] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs, "Optimization of non-functional properties in internet of things applications," *J Netw. Comput. Appl.*, Mar. 2017, doi: 10.1016/j.jnca.2017.03.019.

[15] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "Runtime monitoring of software energy hotspots," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.* Essen, Germany, 2012, pp. 160–169.

[16] K. Aggarwal, A. Hindle, and E. Stroulia, "GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.* Bremen, Germany, 2015, pp. 311–320.

[17] S. A. Chowdhury *et al.*, "A system-call based model of software energy consumption without hardware instrumentation," in *Proc. Int. Green Sustain. Comput. Conf.* Las Vegas, NV, USA, 2015, pp. 1–6.

[18] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proc. Int. Conf. Softw. Eng*. San Francisco, CA, USA: IEEE, 2013, pp. 92–101.

[19] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng*. Hong Kong: ACM, 2014, pp. 588–598.

[20] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "GreenMiner: A hardware based mining software repositories software energy consumption framework," in *Proc. Mining Softw. Repositories*. Hyderabad, India: ACM, 2014, pp. 12–21.

[21] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[22] J. Swan and N. Burles, "Templar: A framework for template-method hyper-heuristics," in *Proc. Eur. Conf. Genetic Program.* Copenhagen, Denmark: Springer-Verlag, vol. 9025, Apr. 8–10, 2015, pp. 205–216.

[23] N. Burles, E. Bowles, A. E. I. Brownlee, Z. A. Kocsis, J. Swan, and N. Veerapen, "Object-oriented genetic improvement for improved energy consumption in Google Guava," in *Proc. Symp. Search-Based Softw. Eng.*, 2015, pp. 255–261.

[24] C. Zhang, A. Hindle, and D. M. German, "The impact of user choice on energy consumption," *IEEE Softw.*, vol. 31, no. 3, pp. 69–75, May./Jun. 2014.

[25] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, "Energypatch: Repairing resource leaks to improve energy-efficiency of android apps," *IEEE Trans. Softw. Eng.*, 2017, doi: 10.1109/TSE.2017.2689012.

[26] M. Rashid, L. Ardito, and M. Torchiano, "Energy consumption analysis of algorithms implementations," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.* Beijing, China, 2015, pp. 82–85.

[27] H. Field, G. Anderson, and K. Eder, "EACOF: A framework for providing energy transparency to enable energy-aware software development," in *Proc. 29th Annu. ACM Symp. Appl. Comput.* Gyeongju, South Korea, 2014, pp. 1194–1199.

[28] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An empirical study of the energy consumption of Android applications," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.* Victoria, BC, Canada, Sep. 2014, pp. 121–130.

[29] A. Gupta, T. Zimmermann, C. Bird, N. Nagappan, T. Bhat, and S. Emran, "Detecting energy patterns in software development," Microsoft Research, Tech. Rep. MSR-TR-2011-106, Redmond, WA 98052, Nov. 2011.

[30] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in Android apps: An empirical study," in *Proc. Work. Conf. Mining Softw. Repositories*. Hyderabad, India: ACM, 2014, pp. 2–11.

[31] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder, "Static analysis of energy consumption for LLVM IR programs," in *Proc. Int. Workshop Softw. Compilers Embedded Syst.* Sankt Goar, Germany: ACM, 2015, pp. 12–21.

[32] X. Li and J. P. Gallagher, "A source-level energy optimization framework for mobile applications," in *Proc. IEEE Int. Work. Conf. Source Code Anal. Manipulation.* Raleigh, NC, USA, Oct. 2016, pp. 31–40.

[33] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.* Newport Beach, CA, USA: ACM, 2011, pp. 199–212.

[34] I. Manotas, L. Pollock, and J. Clause, "SEEDS: A software engineer's energy-optimization decision support framework," in *Proc. Int. Conf. Softw. Eng*. Hyderabad, India: ACM, 2014, pp. 503–514.

[35] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk, "Optimizing energy consumption of GUIs in android apps: A multi-objective approach," in *Proc. 2015 10th Joint Meet. Found. Softw. Eng*, Bergamo, Italy: ACM, 2015, pp. 143–154.

[36] D. Li, A. H. Tran, and W. G. J. Halfond, "Optimizing display energy consumption for hybrid Android apps," in *Proc. Int. Workshop. Softw. Develop. Lifecycle Mobile*, Bergamo, Italy: ACM, 2015, pp. 35–36.

[37] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Raleigh, NC, USA: Lulu, 2013.

[38] N. Burles, E. Bowles, B. R. Bruce, and K. Srivisut, "Specialising Guava's cache to reduce energy consumption," in *Proc. Symp. Search-Based Softw. Eng.*, Bergamo, Italy: Springer-Verlag, 2015, pp. 276–281.

[39] C. A. R. Hoare, "Quicksort," *Comput. J.*, vol. 5, no. 1, pp. 10–16, 1962.

[40] M. Atallah and M. Blanton, *Algorithms and Theory of Computation Handbook, Second Edition, Volume 1: General Concepts and Techniques*. Boca Raton, FL, USA: CRC Press, 2009, pp. 3–10.

[41] P. J. Plauger, *The Standard C Library*. Upper Saddle River, NJ, USA: Prentice-Hall, 1992.

[42] M. D. McIlroy, "A killer adversary for quicksort," *Softw.: Pract. Experience*, vol. 29, no. 4, pp. 341–344, 1999.

[43] F. Otero, T. Castle, and C. Johnson, "EpochX: Genetic Programming in Java with statistics and event monitoring," in *Proc. Annu. Conf. Companion Genetic Evol. Comput.*, Philadelphia, PA, USA: ACM, 2012, pp. 93–100.

[44] G. Neumann, J. Swan, M. Harman, and J. A. Clark, "The executable experimental template pattern for the systematic comparison of metaheuristics," in *Proc. Companion Publ. Annu. Conf. Genetic Evol. Comput.*, Vancouver, BC, USA, 2014, pp. 1427–1430.

[45] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. Int. Conf. Softw. Eng.*, Honolulu, HI, USA: ACM, 2011, pp. 1–10.

[46] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.

[47] S. Haykin, *Neural Networks and Learning Machines* (Neural networks and learning machines), vol. 10. Englewood Cliffs, NJ, USA: Prentice-Hall, 2009.

[48] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng*. Essen, Germany: ACM, 2012, pp. 1–14.

[49] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Math. Control, Signals Syst.*, vol. 2, no. 4, pp. 303–314, 1989.

[50] S. K. Sharma and P. Chandra, "An adaptive slope sigmoidal function cascading neural networks algorithm," in *Proc. Int. Conf. Emerg. Trends Eng. Technol*. Berlin, Germany: Springer-Verlag, Nov. 2010, vol. 87, pp. 105–116.

[51] Clifford, "Preliminary sketch of biquaternions," *Proc. London Math. Soc.*, vol. s1-4, no. 1, pp. 381–395, 1871.

[52] J. Bell, *A Primer of Infinitesimal Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[53] B. DasGupta and G. Schnitger, "The power of approximating: A comparison of activation functions," *Adv. Neural Inf. Process. Syst.*, vol. 5, pp. 615–622, 1992.

[54] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, "Learning activation functions to improve deep neural networks," arXiv:1412.6830, 2014, http://adsabs.harvard.edu/abs/2014arXiv1412.6830A

[55] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[56] J. Durillo and A. Nebro, "jMetal: A Java framework for multi-objective optimization," *Adv. Eng. Softw.*, vol. 42, no. 10, pp. 760–771, 2011.

[57] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.

[58] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of Java collections classes," in *Proc. Int. Conf. Softw. Eng.* Austin, TX, USA: ACM, 2016, pp. 225–236.

[59] R. Abbott, "Object-oriented genetic programming, an initial implementation," in *Proc. Int. Conf. Comput. Intell. Natural Comput.* NC, USA, 2003, pp. 26–30.

[60] W. S. Bruce, "Automatic generation of object-oriented programs using genetic programming," in *Proc. 1st Annu. Conf. Genetic Program.* Cambridge, MA, USA: MIT Press, 1996, pp. 267–272.

[61] S. M. Lucas, "Exploiting reflection in object oriented genetic programming," in *Genetic Programming*, Berlin, Germany: Springer-Verlag, 2004, pp. 369–378.

[62] Y. Oppacher, F. Oppacher, and D. Deugo, "Evolving Java objects using a grammar-based approach," in *Proc. Genetic Evol. Comput. Conf.* NY, USA: ACM, 2009, pp. 1891–1892.

[63] T. White, J. Fan, and F. Oppacher, "Basic object oriented genetic programming," in *Modern Approaches in Applied Intelligence* (LNCS 6703), K. Mehrotra, C. Mohan, J. Oh, P. Varshney, and M. Ali, Eds. New York, NY, USA: Springer-Verlag, 2011, pp. 59–68.

[64] B. Liskov, "'Data abstraction and hierarchy' (keynote address)," *ACM SIGPLAN Notices*, vol. 23, no. 5, pp. 17–34, Jan. 1987.
[65] B. Meyer, *Object-Oriented Software Construction*, 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1988.
[66] N. Burles, J. Swan, E. Bowles, A. E. I. Brownlee, Z. A. Kocsis, and N. Veerapen, "Embedded dynamic improvement," in *Proc. Companion Publ. Annu. Conf. Genetic Evol. Comput. Conf.*, 2015, pp. 831–832.
[67] J. R. Woodward and J. Swan, "Template method hyper-heuristics," in *Proc. Companion Publ. 2014 Annu. Comput.* Vancouver, BC, Canada: ACM, 2014, pp. 1437–1438.
[68] D. M. Cabrera, "Evolutionary algorithms for large-scale global optimisation: A snapshot, trends and challenges," *Progr. Artif. Intell.*, vol. 5, no. 2, pp. 85–89, 2016.

Alexander Edward Ian Brownlee is currently a Senior Research Assistant in the University of Stirling, Stirling, U.K., previously worked as a Software Engineer in the industry. He has published more than 40 papers in international journals and conferences. His research interest focuses on value-added optimization: Combining metaheuristics and related methods with machine learning to both find optimal solutions and reveal insights into the problem, helping people make better decisions.

Nathan Burles is currently a Software Engineer at IBM York, York, U.K., previously worked as a Research Associate at the University of York, York, U.K. He has published a number of papers in international journals and conferences, as well as refereeing for international conferences and workshops. His research interests include neural networks, metaheuristic optimization, and machine learning.

Jerry Swan spent nearly 20 years, before entering academia, as a Systems Architect and software company owner. He has published more than 70 papers in international journals and conferences, has lectured and presented his research worldwide, and has run international workshops and tutorials on the automated design of algorithms. His research interests include metaheuristic optimization, symbolic computation, and machine learning.