Approximate Oracles and Synergy in Software Energy Search Spaces

Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr

Abstract—Reducing the energy consumption of software systems through optimisation techniques such as genetic improvement is gaining interest. However, efficient and effective improvement of software systems requires a better understanding of the code-change search space. One important choice practitioners have is whether to preserve the system's original output or permit approximation, with each scenario having its own search space characteristics. When output preservation is a hard constraint, we report that the maximum energy reduction achievable by the modification operators is 2.69 percent (0.76 percent on average). By contrast, this figure increases dramatically to 95.60 percent (33.90 percent on average) when approximation is permitted, indicating the critical importance of approximate output quality assessment for code optimisation. We investigate synergy, a phenomenon that occurs when simultaneously applied source code modifications produce an effect greater than their individual sum. Our results reveal that 12.0 percent of all joint code modifications produced such a synergistic effect, though 38.5 percent produce an antagonistic interaction in which simultaneously applied modifications are less effective than when applied individually. This highlights the need for more advanced search-based techniques.

Index Terms—Search-based software engineering, search space, energy consumption, genetic improvement, synergy, antagonism, oracle, approximation

1 Introduction

Reducing energy consumption is an increasingly important software engineering concern. In 2010, large server clusters consumed 1.12–1.50 percent of global energy consumption [30]: an amount equivalent to that consumed by the United Kingdom in 2015 [12]. Environmentally unfriendly sources generate much of this energy: in 2013, 67 percent of global energy consumption derived from burning fossil fuels, with 41 percent generated from the most highly-polluting of all sources, coal [6]. Using a variety of search techniques [23], recent studies have shown how to reduce the energy consumption of software given reasonable assumptions about the end-use of the improved software system such as the likely input data [15], network usage information [39], and tolerance to less desirable user-interfaces [42].

Reducing energy consumption via the search-based modification of software systems is an instance of 'Genetic Improvement' (GI) [52]. To genetically improve a program, search techniques modify software with the goal of constructing related versions that retain some important properties while improving others. GI research, hitherto, has been dominated by three operators: *delete*, *copy*, and *replace* applied to source code lines[1] [34], [35], [53]. The *delete* operator deletes a line of code; *copy* copies a line of code to another location; and *replace* replaces a line of code with another. The challenge in GI research is designing search techniques to select a subset of all possible modifications that may then be applied to the target software to produce an optimal (or near optimal) solution. Until now, there has been little effort put to analysing the search space these operators produce, and that must be subsequently traversed, when optimising software's energy consumption. This is unfortunate as GI practitioners have much to gain from understanding the characteristics of these search spaces.

The *delete*, *copy*, and *replace* operators generate an infinite search space, bounded by the number of *copy* operator applications. Even when restricted to a single operation, the search space remains large. For a program with N lines of code, every line can be deleted (N), copied into the program before existing lines (N^2), or replaced with any other line but itself ($N^2 - N$). In this study, the smallest application we investigate, Bodytrack, has 1,030 modifiable lines of code and, thus, over 2 million possible variants generated by a single application of an operator. GI techniques typically restrict the search by selecting a subset of the software system for modification. This subset is usually chosen by an expert with intimate knowledge of the system or via profiling; selecting lines/files/components/etc. based on their likelihood of impacting the target non-functional property. In practice, even this restricted search space remains vast. The necessity for well-designed search techniques is clear, though the

• *The authors are with the University College London, London WC1E 6BT, United Kingdom.*
E-mail: r.bruce@cs.ucl.ac.uk, {j.petke, mark.harman, e.barr}@ucl.ac.uk.

Manuscript received 10 Mar. 2017; revised 23 Feb. 2018; accepted 20 Mar. 2018. Date of publication 16 Apr. 2018; date of current version 12 Nov. 2019. (Corresponding author: Bobby R. Bruce.)
Recommended for acceptance by S. Apel.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2018.2827066

1. Other GI work has also modified software at the binary and assembly levels [32], [55].

information required to effectively design one is not presently available. The aim of this investigation is to gain greater understanding of the search space and considerations researchers should take when optimising software's energy consumption using GI.

When software is optimised using GI, an oracle must be provided. An oracle verifies a modified software's correctness [8]. In genetic improvement it is often a well-designed test-suite, though it can be anything which conforms to this definition. In this work, we focus on opportunities for energy improvement under two different test oracles—exact and approximate. An exact test oracle requires the original and improved programs to produce identical output, while an approximate test oracle uses a more relaxed notion of whether the output from the improved program is acceptable. In this investigation, we use test suites to determine the correctness of an application and, therefore, we wish to emphasise that, when we say 'exact' or 'approximate', it is exact or approximate *modulo a test suite*. Each of these test oracles produce their own search space, both applicable to GI research and both worthy of study. Approximate test oracles permit trading quality attributes against energy consumption, which previous work on energy improvement (using GI and other techniques) has shown effective. For example, a mobile application can trade the aesthetics of a user-interface [42] while a graphics-based application can trade image quality [57]. In such cases, deviation from the precise output of the original may be tolerable if a decrease in energy consumption is observed. In a survey of software engineers responsible for systems in which energy consumption is a concern, the majority (80 percent) were willing to sacrifice certain requirements for reduced energy consumption [43].

For this study, we analyse the search space of four systems—7zip, Bodytrack, Ferret, and OMXPlayer. For each, we define, justify, and investigate approximate oracles that make domain-specific trade-offs between energy consumption and solution quality. We are interested in knowing at what frequency effective modifications exist in this search space, what impact they are capable of producing, and how this varies between exact and approximate test oracles.

Most previous energy optimisation work in software engineering has used *indirect* measures of energy consumption. Examples are tools which estimate energy by logging processor states [15], monitoring bytecode execution [16], or via simulation of hardware [61]. They interpolate energy from correlated measurements. Indirect measurements are typically close to actual energy consumption, but their error is often unknown. Given that improvements reported hitherto are relatively modest (in the range of a few percent to a few tens of percent), it is important to quantify measurement error. To this end, we conduct our experiments on a suite of 6 MAGEEC energy measurement boards [2], connected to a cluster of 25 Raspberry Pi devices [4]. The use of MAGEEC boards allows us to take *direct* energy measurements. That is, energy is measured directly rather than through a proxy. We chose to monitor the energy of software running on Raspberry Pi devices as they are a simple, cheap, widely available, and easily configurable platform.

This Raspberry Pi cluster enables us to distribute software variants across different physical devices. We are not the first to use direct energy measurements; the GreenMiner project used direct energy measurements to determine the energy consumption of Android mobile applications [27], for example. However, we are, to the best of our knowledge, the first to evaluate energy consumption in this distributed format. As such, we can take many more measurements than we would otherwise be able to and can thereby quantify statistical error, like 'background noise', by reporting the averages found over many runs. A key finding of ours is that individual devices exhibit systematic error [58]. We find energy changes reported in Joules can vary considerably across different devices even when the statistical error within a single device is small. In future work, it is paramount that such systematic error is properly addressed. Within our investigation, we find the proportional change in energy measurements is stable across all devices and therefore report results as proportional increases or decreases.

This setup enables us to understand the properties of the energy search space by measuring the energy consumed when running software modified by the *delete*, *copy*, and *replace* operators. We can analyse both the local neighbourhood (a single modification) and beyond (multiple modifications), allowing us to give insight to GI practitioners.

If we were to find that the local search space is flat (i.e., a single modification is incapable of, or rarely produces, a significant proportional change in energy consumption), then we could conclude that either the *delete*, *copy*, and *replace* operators are relatively ineffective or a highly explorative search technique is required to optimise software. Alternatively, if we find the local search space to be on a steady gradient, then the search-based algorithm should be based on exploitation (such as a hill-climbing algorithm) and, depending on the incline, may suggest that GI researchers intuitions are correct—the *delete*, *copy*, and *replace* are effective.

The nature of the wider search space can be determined by combining modifications and noting their interaction. In our investigation, we observe instances when adding or removing a set of modifications produces a good solution but adding or removing a subset produces a much less effective solution. We refer to this as *synergy*, a specific form of interaction where the improvement of simultaneously applied modifications exceeds the sum of applying each in isolation [9]. We also observe *antagonism*, another form of interaction that is the opposite of synergy. Antagonism occurs when the effectiveness of a solution worsens as modifications are combined in comparison to when they are applied individually. If antagonism is infrequent, then a greedy approach would be sufficient in combining modifications; simply sample modifications uniformly, evaluate them and, if they are found to be effective, add them to a list of good mutations to then be applied en-masse at the end of the process. In our investigation we find that antagonism occurs in 38.5 percent of all modification pairings—a frequency high enough to justify more advanced search techniques.

We investigate the search space and provide considerations researchers should take when optimising software's energy efficiency using GI. This paper makes three main contributions:

(1) The investigation shows how real-world energy measures can be made while taking into account the effects of per-device statistical error and systematic error across devices.

```
1  Property getProperty (List<Summary> summaries){
2    List<int> values;
3    for(Summary s in summaries){
4      values.add(s.getValue());
5    }
6
7    return new Property(
8      //Mod_1: aggregate → sample
9      new aggregate(values)
10   );
11 }
12
13 int sample (List<int> input){
14   input=sort(input); //Mod_2: Delete this line
15   return input.get(random(0, input.size()))
16           * input.size();
17 }
```

Fig. 1. Two software modifications: `Aggregate` consumes more energy than `sample`; Mod_1 replaces `aggregate` with `sample`, which does not need sorted inputs, so Mod_ 2 combines with Mod_1 to further reduce energy consumption.

(2) Software testing traditionally relies on exact oracles that do not tolerate output deviations; we show that approximate oracles, which tolerate output deviations, open the door to greater energy savings via genetic improvement.
(3) Software changes that alter an application's energy consumption may interact: sometimes synergistically and sometimes antagonistically. We show that this phenomenon is ubiquitous, implying sophisticated search must be used when optimising software's energy efficiency.

2 Motivating Example

The key to understanding the search space of energy-efficient software optimisations is to know at what frequency effective modifications occur, what impact they are capable of producing, and whether synergy and antagonism are common. We find these using both approximate and exact test oracles. This section provides a motivating example to explain these concepts.

In Fig. 1, 'Mod_1' swaps a method that aggregates a list (at line 9) with one that samples. This increases the approximation of `getPropery`'s output but may achieve considerable energy savings because of sampling's relative efficiency. This is the type of modification that an approximate test oracle allows.

If we further assume the input to the method `sample` is sorted, then line 14, `input = sort (input);`, is not required. The software engineer responsible for this line may have included it to ensure robustness or due to a lack of knowledge about the contract that the `sample` method obeys. Regardless, guided by a sufficiently adequate test suite, GI can remove such redundancies when using an exact test oracle. In previous GI work by Petke et al. [53] and later Bruce et al. [15], such optimisations were found when deleting complex assertions in MiniSAT's `Solver.c` class. The 'Mod_2' example is similar; an exact test oracle can find the modification, since it does not affect the software's output, only its target non-functional property.

It is tempting to pursue the modifications found by the exact oracle exclusively, as they produce benefits without cost. However, if we permit the quality of output to degrade (i.e., permit approximate output), then this should increase the set of valid solutions in the search space and facilitate the search for even more energy-efficient solutions. We are the first to quantify the frequency of these modifications and measure their interactions. Fig. 1 demonstrates synergistic software modifications. 'Mod_1' decreases the number of times in which the more energy inefficient method `aggregate` executes by replacing it with `sample` while 'Mod_2' increases the efficiency of `sample`.

The equations below explain the basic mathematics of this synergistic interaction. The energy consumed by the program m_p equals the sum of m_g, the energy consumed by `getProperty`, multiplied by N_g, the number of runs, and m_s, the energy consumed by `sample`, multiplied by N_s, the number of samples. In our example, $N_g \geq 1$ and $N_s \geq 1$. Activity outside these methods is assumed to be constant and is represented by m_o and thus we have

$$m_p = m_s N_s + m_g N_g + m_o.$$

'Mod_1' changes `getProperty` to call `sample` instead of `aggregate`. The energy consumption of `getProperty` thereby includes the energy of a single iteration of `sample` plus the remainder of `getProperty` minus the call to `aggregate`, m_a. So when Mod_1 is applied, $m_g \rightarrow m_g - m_a + m_s$ and we have

$$m_p = m_s N_s + (m_g - m_a + m_s) N_g + m_o,$$

'Mod_2' decreases the energy consumption of `sample`, m_s. With 'Mod_1' present, energy is reduced in both $m_s N_s$, and in `getProperty`, formally $m_g N_g$ is now $(m_g - m_a + m_s) N_g$. Without 'Mod_1', 'Mod_2' only affects the energy consumed in `sample`, however, with 'Mod_1', 'Mod_2' may reduce energy consumption in both functions. In this investigation, we wish to understand how frequent these synergistic (or, the opposite—antagonistic) interactions occur within the search space.

3 Methodology

In this section, we explain the design and implementation of our measurement framework. We then discuss our source code representation and how we modify it before explaining how we compare the effectiveness and energy efficiency of a program and one of its variants, under both exact and approximate test oracles. Finally, we introduce our system for classifying interactions between modifications.

3.1 Measurement Framework

Given a set of modifications, we seek to measure the effect on the energy consumption when each is applied to the target application. In theory, the setup is simple: take a program (modified or otherwise) along with an input and measure its energy consumption during execution. In practice, however, it is not so simple: one must choose between direct and indirect measurement and contend with the cost of taking a measurement, since program execution can be expensive.

Most previous search-based approaches to optimising the energy efficiency of software have estimated energy consumption [15], [16], [61]. These estimates can miss important high or low energy events thereby directing the

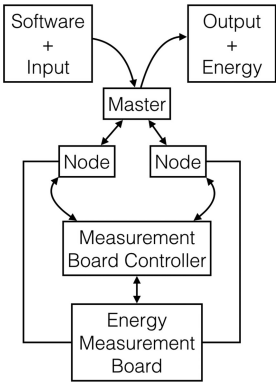

Fig. 2. Diagram of the energy measurement cluster, showing two nodes measured by a single energy measurement board.

search away from an optimal solution. The only guaranteed way to capture these events is to ensure the framework makes direct energy measurements rather than relying on either estimates or simulation.

Programs are one component of a larger system: the computer that executes them. At present, one cannot directly measure the energy consumption of a program, because existing devices do not expose the coupling points between hardware components or operating system's processes. One can, however, directly measure the energy consumption of the entire computing system; this is what we chose to do in this investigation. Measuring the whole system carries with it the challenge of contending with statistical measurement error due to events external to the program, like OS background processes. We mitigate these effects by taking multiple measurements and averaging the results.

Directly measuring the energy consumption of a program entails running it, and, as we have just argued, the system that hosts it. Thus, taking multiple direct measurements exacerbates measurement cost. It thereby follows that our framework must be efficient and scalable. For instance, the experiments outlined in Section 6 require the evaluation of 28,000 modifications (some of which can take up to five minutes to evaluate), across 4 applications, with each modification run multiple times against a test suite. Fortunately, this task is easily parallelisable, and our framework exploits this: it is a cluster of individual computer systems, each of which can measure its own energy consumption. Jobs (programs, modified or otherwise, along with input data) are sent from a client to the cluster's master node. The master node then distributes jobs to nodes (a maximum of one job running on any given node at any one time). These nodes then measure their energy consumption while running these jobs. The energy measurements, along with the outputs from each job, is then returned to the client via the master node. Fig. 2 shows our framework's layout with 2 nodes.

The nodes in this cluster are Raspberry Pi 2 Model B devices [4], each running Raspbian OS [5], a GNU/Linux OS based on Debian. The Raspberry Pis were chosen as they provide a cheap computer system representative of a real-world system in terms of architecture and their running of a Unix-based operating system. Each Raspberry Pi node can measure its own energy consumption via a MAGEEC Energy Measurement Board [2].

The MAGEEC Energy Measurement boards are simple, inexpensive devices which sample the voltage drop across a resistor inline to the target's power supply (i.e., the Raspberry Pis' power cable in our case) at a sustained rate of 2MHz. A micro-controller on the MAGEEC Energy Measurement board listens for start and stop commands over a USB connection. When a start command is received the micro-controller begins sampling measurements and sends readings across the USB connection until a stop command is received. The MAGEEC board is controlled by a separate Raspberry Pi device (we refer to this as the 'measurement board controller') that is responsible for issuing the start and stop requests and reading energy data from the USB connection. As all the Raspberry Pi devices (nodes and measurement board controllers) are on the same network, nodes send requests to their respective measurement board controller to start, stop, and receive energy measurements. Each MAGEEC board can measure up to three targets at once and, therefore, the ratio is three Raspberry Pi nodes to every MAGEEC board plus an additional Raspberry Pi (the measurement board controller) to manage the readings, and the start and stop commands.

For clarity, here is an abstracted view of how a job is run from the point of view of a node:

(1) Receive a job from the master node.
(2) Setup the job environment (typically decompressing files and moving them to the correct directories).
(3) Send a message to the measurement board controller to begin energy readings.
(4) Run the job.
(5) Send a message to the measurement board controller to stop energy readings.
(6) Request the energy reading from the measurement board control.
(7) Send the output of the job and the energy reading back to the master node.

This setup mitigates the costly process of evaluating many thousands of modifications and can easily be expanded if needed. The relatively inexpensive components are an advantage compared to alternative approaches, allowing for more nodes than would otherwise be possible. As we discuss in Section 6.1, the MAGEEC energy measurement framework is incapable of producing results with the level of accuracy that we would deem acceptable for most investigations and, as such, have had to report proportional measurement increases/decreases which we find are reliable. As in most endeavours, there is a clear trade-off between quantity and quality of hardware.

3.2 Producing Variants

As previously noted, we use three genetic improvement operators: *copy*, *delete*, and *replace*, each of which is applied at the source code line level. We apply these to a simple tagged representation of source code; a representation specially created for GI research, first introduced by Langdon and Harman in 2010 [33], and later utilised in a variety of other GI work [34], [35], [53]. We refer to this format as the *Langdon format*.

To translate code to the Langdon format each line is labelled with a unique identifier. These identifiers indicate

```
<LzFind_259>    ::=     " if" <IF_LzFind_259> " \n"
<IF_LzFind_259> ::=     "(p->keepSizeAfter >= 0)"
<LzFind_261>    ::=     "{\n"
<LzFind_262>    ::=     "" <_LzFind_262> "\n"
<_LzFind_262>   ::=     "MatchFinder_ReadBlock(p);"
<LzFind_265>    ::=     "}\n"
```

Fig. 3. A snippet from LzFind.c, a 7zip file, in the Langdon format. Lines starting with <LzFind are unmodifiable.

whether a line is modifiable or not. Unmodifiable identifiers begin with <{FILE}. Opening and closing curly brackets, variable initialisations, and function declarations, are unmodifiable. In the case of IF, WHILE, and FOR, only the conditions, and the pre- and post-statements in the case of FOR, can be modified. Fig. 3 shows a snippet of source code in the Langdon format.

At present, tools to transform code into the Langdon format exist only for C/C++ code so we target only C/C++ applications in this investigation. As the operators to be applied operate on source code lines, the targeted code is formatted so that each statement is on its own separate line to avoid modifications being applied to multiple statements. Opening and closing curly brackets are moved to their own line so any modifications to lines containing a statement do not interfere with program scopes. In order to reduce errors, we also ensure the bodies of bracketless one-statement FOR/WHILE/IF constructs are refactored to be enclosed within curly brackets.

Once converted to the Langdon format the source code can be modified by simply *deleting*, *replacing*, or *copying* a tag, taking into account the aforementioned restrictions. It can then be expanded back to the original source code by taking the unmodifiable lines then expanding them. For example, in Fig. 3 <LzFind_262>, an unmodifiable line, references <_LzFind_262>. When converted back to source code <LzFind_262> is expanded to produce MatchFinder_ReadBlock(p);\n.

Fig. 4 shows an example of how modifications generated by the *copy*, *delete*, and *replace* operators are represented and combined. A modifiable identifier alone, <LINE_ID>, is a *delete*; a modifiable identifier followed, without a space, by another, <LINE1_ID><LINE2_ID> is a *replace* that replaces the former with the latter; and two identifiers separated by +, <LINE1_ID>+<LINE2_ID>, is a copy operation that copies one line (the latter) to another area of the source code (above the former). A space separates multiple operations.

Outside of this representation, we apply some restrictions. First we restrict *replace* so that the condition of an IF, FOR, or WHILE statement can only be replaced with the condition of a matching statement. For example, an IF's conditional can only replace another IF's conditional. A FOR's pre-statement can only replace another FOR's pre-statement, and the post-statement with another FOR's post-statement. When the *delete* operator is applied to a conditional clause, it replaces the conditional with false; this is equivalent to the deletion of the IF, FOR, or WHILE body. For example, the *delete* operator transforms if(i < 10) to if(false).

In line with previous uses of the Langdon format, we limit the search space by restricting the *copy* and *replace* operations to a single file, e.g., a line from file X can only be copied to another location in file X. This restriction significantly decreases the number of compilation errors related to out-of-scope variables and methods.

```
#DELETE line 262
<_LzFind_262>

#REPLACE if condition in line 259 with if
#condition in line 307
<IF_LzFind_259><IF_LzFind_307>

#COPY line 299 and insert above line 325
<_Solver_325>+<_Solver_299>

#REMOVE line 262 and REPLACE if condition
#at line 259 with if condition in line 307
<_LzFind_262>  <IF_LzFind_259><IF_LzFind_307>
```

Fig. 4. Four examples of modifications that may be applied to LzFind.c.

3.3 Assessing Individual Modifications

As we noted in our introduction, the search space of possible modifications is vast, too vast to analyse exhaustively. We therefore choose to uniformly sample it.

Formally, we define a modification as follows:

Definition 1 (Program Modification). *A program modification is a pair $\delta = (e, \vec{l})$ where $e \in \{copy, delete, replace\}$ and $\vec{l}$ is a pair of program locations.*

Remark. We require locations $\vec{l}$ to be a pair, since *copy* and *replace* operations require two program locations.

We apply modifications chosen uniformly at random to lines tagged as modifiable in the Langdon format (let this number be n). We then add individual modifications that compile to what we refer to as the *Modification Set* until the cardinality of this set is $2n$.

Algorithm 1. The Filtering Step

Input: E, a set of modifications (Def. 1)
P, the target software
T, the set of testcases
N, the number of energy measurements

1: $t \leftarrow$ uniformSelection(T)
2: $M_P \leftarrow \{\}$ # A set of energy measurements
3: **for** 1..N **do**
4: $M_P \leftarrow M_P \cup \{m(P(t))\}$
5: **end for**
6: $[b_l, b_h] =$ CI (M_P) # We discard b_h; CI defined in text
7: $E' \leftarrow \{\}$
8: **for** $\delta \in E$ **do**
9: $P' \leftarrow$ applyMod(P, δ)
10: $J \leftarrow m(O'p \leftarrow P'(t))$
11: **if** testOracle(O'_P, t) $\wedge$ $J < b_l$ **then**
12: $E' \leftarrow E' \cup \{\delta\}$
13: **end if**
14: **end for**
15: **return** E'

Even when sampling, we cannot evaluate every variant against all available testcases. Variants that are inert, produce software that breaks hard-constraints or increase

energy consumption are uninteresting. In previous work, we observed that these variants make up the majority of any given local search space [15]. Therefore, we filter them out.

Algorithm 1 presents our filtering algorithm. The algorithm evaluates members of the modification set, E. First, it uniformly selects a testcase t from a the set of testcases T. Then, at lines 3 to 5 the algorithm runs the original program P using test t with its energy measured via function m for N times (100 in this case). The set of energy measurements M_P is then used to determine the 95 percent confidence interval lower bound, b_l. In lines 8 to 14, for each modification δ in the modification set E, we apply the modification to the program, thereby creating a program variant P'. We then record the output of the program O'_p, and measure the energy consumption, J (Joules), of this program variant. If the variant passes the testcase (validated using an oracle) and its energy consumption is less than the 95 percent confidence interval (CI) of the mean lower bound, we add the modification to the *Candidate modification set*, E'. After this filtering step, the Candidate modification set contains those modifications for which we can say, with statistical confidence, an improvement in energy efficiency has been observed while still passing the testcase.

It should be noted that 'passing' a testcase in this instance does not necessarily mean producing the same output as the original, it may be approximated. For example, in the case of 7zip to pass a testcase, the application must compress the testcase in a manner that it may be decompressed to its original state though the compressed file generated by a program variant is permitted to differ from that produced by the original application. Section 5 precisely describes the criteria used for the exact and approximate test oracles. These criteria determine whether the software variants pass or fail for a given input.

Algorithm 2. The Evaluation Step

Input: E', the set of modifications (Def. 1)
P, the target software
T, the set of testcases
N, the number of energy measurements
1: $D \leftarrow \{\}$ #Collection of modification data
2: **for** $t \in T$ **do**
3: $O_P \leftarrow P(t)$
4: **for** 1..N **do**
5: $J \leftarrow m(P(t))$
6: D.addRecord($\perp$, t, J, 0, true)
7: **end for**
8: **for** $\delta \in E'$ **do**
9: $P' \leftarrow$ applyMod(P, δ)
10: **for** 1..N **do**
11: $J \leftarrow m(O'p \leftarrow P'(t))$
12: $p \leftarrow$ testOracle(t, O'_P)
13: $a \leftarrow$ getApproxVal(O_P, O'_P)
14: D.addRecord(δ, t, J, a, p)
15: **end for**
16: **end for**
17: **end for**
18: **return** D

Algorithm 2 presents the pseudocode that explains how we gather data to evaluate the candidate modification set. For each testcase, t, the unmodified software is run N times (N is 30 in our investigation) with its energy, J, measured on each iteration via function m (line 5). Then, again for each testcase, at line 9 each candidate modification δ is applied to the unmodified software to produce the modified variant, P'. The modified variant then processes the testcase with its energy J measured and its output O'_P recorded for N iterations. We subsequently use this data to determine whether a modification produces a statistically significant reduction in energy consumption, using the Mann-Whitney U test (for the α level 0.05).

At line 12 of the algorithm, we record whether the software variant has passed the testcase, and, at line 13, we determine the *Approximation Value*, a, our unified approach to recording values from both exact and approximate oracles. The formula for both the approximation value and what passing a testcase means for each application is defined in Section 5.

In all cases, an approximation value of zero denotes satisfaction of the exact oracle—that is, the output of the modified program corresponds to the output of the original program (modulo the testcases). However, the results of the approximation value can be non-zero, with higher approximation values corresponding to greater degrees of approximation. The calculation of the approximation value is unique to the application domain and therefore approximation values from different applications cannot be directly compared. If a new application were to be introduced, the calculation of that application's approximation value would have to be created by an expert with domain knowledge. Our common terminology serves to combine very different measures of approximation. We define the four domain-specific approximation criteria we use in this investigation in Section 5.

3.4 Classifying Interactions of Multiple Modifications

Within this investigation, we wish to study how modifications interact. In obtaining the data to do so, we take two effective modifications (those known to pass all tests and reduce energy consumption) and measure their energy when applied to a piece of software both individually then when combined. The difficulty lies in interpreting the results. To do so, we label an interaction in accordance to its place within an *Interaction Spectrum* outlined in the following definitions.

Definition 2 (Patch). *A patch Δ is a non-empty sequence of modifications δ (Definition 1).*

Remark. In the case where a location within the software is modified, and that location is subsequently used by another modification later in the sequence, the new value of that location is used (i.e., the value of that location after the preceding modifications have been applied). For example, if line X is deleted via the *delete* operation, and then line Y is replaced with the value of line X via the *replace* operation, it is a deleted (i.e., blank) line that line Y is replaced with.

Definition 3 (Interaction Spectrum). *Let $r(P_\Delta)$ denote the reduction in energy measurement when patch Δ is applied to program P. Formally, $r(P_\Delta) = m(P) - m(P_\Delta)$, where m is our energy measurement function. Two patches Δ_1 and Δ_2 interact*

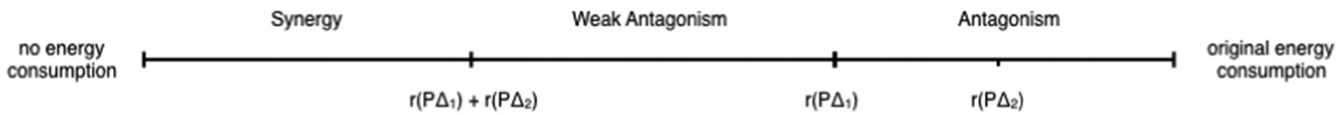

Fig. 5. The interaction spectrum (Definition 3) used in this investigation when studying the effects of two energy-saving software modifications in respect to the energy consumed by the original software.

when the measure of their joint and independent application differs. Formally, $r(P_{\Delta_1\Delta_2}) \neq r(P_{\Delta_1}) + r(P_{\Delta_2})$. *Fig. 5 shows this* interaction spectrum *partitioned into three categories:*

$$\begin{aligned} Synergy &: r(P_{\Delta_1\Delta_2}) > r(P_{\Delta_1}) + r(P_{\Delta_2}) \\ Weak\ Antagonism &: r(P_{\Delta_1\Delta_2}) < r(P_{\Delta_1}) + r(P_{\Delta_2}) \\ &\wedge\ r(P_{\Delta_1\Delta_2}) > \max(r(P_{\Delta_1}), r(P_{\Delta_2})) \\ Antagonism &: r(P_{\Delta_1\Delta_2}) < r(P_{\Delta_1}) + r(P_{\Delta_2}) \\ &\wedge\ r(P_{\Delta_1\Delta_2}) < \max(r(P_{\Delta_1}), r(P_{\Delta_2})) \end{aligned}$$

Remark. We classify an interaction as synergistic when the joint application of two patches has a greater effect than assuming no interaction. When weak antagonism occurs, we accept that the patches interact in a way as to dilute their effects but not in a manner that precludes their joint application. When interactions exhibit synergy or weak antagonism, both patches should be applied because the reduction in energy from their joint application exceeds either applied alone; if the patches exhibit strong antagonism, the most effective patch should be chosen and the other discarded.

4 Research Questions

Any attempt to improve energy consumption, search-based or otherwise, relies on the ability to reliably measure energy. Our first question therefore investigates the degree to which our energy measurements are sufficiently reliable to assess energy improvement:

RQ1, Measurement. What variance occurs when measuring energy consumption?

As with all forms of real-world measurement, energy measurements are vulnerable to a number of different sources of variation. With this in mind, we wish to establish the degree of variance to expect, both for a single energy measurement device and across multiple devices. Even on a single device, the amount of energy consumed may vary when, on different occasions, exactly the same software system is executed with exactly the same test suite; we wish to understand the magnitude of this variance. If the variance is high, then we have no foundation upon which to make reliable measurements. We argue that *any* experimental work on energy assessment or improvement should, as a preliminary step, report results for such variance, in order to exclude a serious potential threat to validity of the scientific findings. This motivates our first research question:

RQ1a: What is the variance when measuring using a single energy measurement device?

To answer this question, we choose a node within our cluster as a test target. Then for each application, we uniformly select a testcase and execute the application 30 times on the target node, recording the energy consumed during each iteration. We use this data to measure within device variance. This variance informs us of the statistical error in the measurements we obtain.

Even if the variance is small when executing within a device, there may be variance between different devices. Several previous studies of energy assessment and improvement have reported results based on only a single device [7], [41], [44]. This leaves open another potential threat to the validity of the findings, which would occur if different instances of the same device type give highly different readings for the same software system and test suite. While RQ1a informs us of the statistical error, we may miss detecting a form of systematic error where different devices give different measurements for the same process. *This motivates RQ1b*:

RQ1b: What is the variance in direct energy measurements across multiple devices?

To answer RQ1b, we uniformly select an application and a corresponding testcase. We then run that application and testcase pair on *all* the devices in the cluster, 100 times, measuring the energy consumption each time. We use box-plots (one box-plot per device) to determine if there is variance in energy measurements across the devices.

Our goal in answering RQ1b is to find whether there is systematic error across different devices. This error *can* be tolerated if it is consistent as we are only interested in the *proportional* differences in energy consumption when assessing energy improvement, not the *absolute* measure of energy consumed. *This motivates RQ1c*:

RQ1c: What is the variance in proportional energy changes across multiple devices?

To answer RQ1c, we uniformly select an application and all of its testcases. We then run the application and with all of its testcases on every device in the cluster. For each device, starting with the testcase which consumed the smallest amount of energy, we record the proportional increase in energy consumption between it and the next smallest testcase. We then create a box-plot for each of these proportional increases across all devices to show whether these proportional figures are reliable across our cluster. Once we have determined the suitability of our energy measurement cluster, we may begin evaluating our applications and the variants produced by applying modifications to them.

When assessing whether an improved program is acceptable or not, we need a test oracle that determines whether the behaviour of the improved program is acceptable with respect to the behaviour of the original. In software testing, more generally, this is an instance of the oracle problem which, though significant, is largely unsolved [8]. However, one of the advantages of genetic improvement is that the original version of the program can act as the test oracle, against which improved versions are compared [24], [60]. For a given candidate program variant, we compare the

behaviour of the original program with that of the candidates to check whether it has deviated from the behaviour of the original, and therefore should be discarded. This raises the fundamental question of how much deviation from the behaviour of the original can be tolerated.

For some application scenarios, no deviation can be tolerated, but, in many other scenarios, exact replication of the behaviour of the original is unnecessary. Previous work on genetic improvement has shown that genetically modified programs may improve not only targeted non-functional properties of interest, but also the functionality of the original program [34]. In such situations, the original program's behaviour only acts as a guide to the desired behaviour of the genetically improved program.

Furthermore, even when preservation (or improvement) of functional properties is not possible, the genetically modified program may need only *approximate* the behaviour of the original, sacrificing some degree of output quality for improvements in non-functional characteristics. For example, programming graphics shaders inherently involves a precision-speed trade off that genetic improvement techniques can exploit to produce renders of lower quality in a more limited time budget [57]. Often minor quality degradation is imperceptible or acceptable to the end user, making such trade-offs highly desirable. Much of the work on energy improvement falls into this category [28], [42], [47].

This motivates RQ2, which investigates using an approximate test oracle that allows us to trade solution quality against energy improvement:

RQ2, Improvement. What additional energy improvement can be achieved when using approximate test oracles in place of exact test oracles?

In answering RQ2, we investigate the degree to which energy efficiency can be improved by sacrificing solution quality, guided by a domain-specific approximate test oracle in each case. We also investigate the effect of approximate test oracles on the frequency and impact with which the different genetic operators affect energy consumed and the trade-off between energy consumption and solution quality.

Finally, we consider the way in which different genetic improvement modifications to the original program combine to improve energy efficiency. The motivation for this research question derives from the way in which search-techniques typically combine lower-level building blocks of partially fit solutions in order to arrive at fitter combined solutions [19], [25], [46]. In RQ3, we therefore study the interactions of combinations of individual modifications, reporting the frequency of different kinds of synergistic effects:

RQ3, Synergy. How frequently do synergistic and antagonistic effects occur when combining known effective modifications?

To answer RQ3, we perform a pairwise investigation of the modifications found to reduce energy. We take 15 percent of all possible pairings from the set of effective modifications found in answering RQ2 (those found when using the approximate test oracle). We evaluate each and report the frequency of synergy and antagonism observed in accordance to the interaction spectrum outlined in Section 3.4.

5 Test Subjects and Their Oracles

In order to answer these Research Questions, we chose four test subjects using the following selection criteria.

5.1 Selection Criteria

The tool used to generate the Langdon format required all software to be C/C++ with license permitting its use for experimental purposes. As evaluation takes places on a Raspberry Pi device running the Raspbian OS, the software had to be compilable within this environment.

Due to inevitable overheads associated with sending energy measurement start and stop commands over a network [36], [37], we chose applications that have a non-trivial execution time, which we have defined to be greater than 5 seconds. The larger the execution time, the smaller the overheads are as a percentage of total energy consumption.

We limited the selection further to applications that can be run via command line, have testcases (or applications in which they can easily be generated), provide a deterministic output for any given input and, once execution has started, do not require further user interaction. We imposed these requirements to aid in the automation of experiments.

In choosing applications, we consulted relevant literature on energy optimisation and found the PARSEC benchmark suite has been utilised frequently [28], [55]. We therefore decided that applications from these suites should make up part of our selection. To avoid selecting from a single source, we limited selection from the PARSEC benchmark suite to two applications then searched open-source repositories, such as GitHub and SourceForge, for the remainder. In addition to the criteria outlined above, we diversified the application domains in our corpus, searching until we found applications in each of the following application domains: file compression, video processing, database processing and image processing.

We selected from these domains as they are both important and popular, thus mitigating any concerns that our findings are not representative of real-world software. Furthermore, we believed these domains were likely to satisfy our aforementioned requirements, particularly in that they are all domains known to have non-trivial execution times for inputs that can be easily obtained or generated.

Table 1 shows the applications' domain, their lines of code (LOC), the number of lines declared as modifiable in the Langdon format, and the number of modifications we generate and study. In the following sections, we summarise each application, focusing on the technical or implementation details relevant to our investigation.

5.2 7zip

7zip Version 9.38.1 (Unix/Linux port) [1] is an open source file archiver with its own `7z` archive format. It consists of 136,828 lines of C/C++ code spread over 400 files. For the experiments outlined in this paper, we concerned ourselves only with the core Lzma compression and decompression algorithms for optimisation. Excluding files associated with user I/O behaviour, we identified 6 files (`7zCrcOpt.c`, `LzFind.c`, `Lzma2Dec.c`, `Lzma2Enc.c`, `LzmaDec.c` and `LzmaEnc.c`) that accounted for over 99 percent of execution time when compressing and decompressing a 50 MB

TABLE 1
Number of Modifications Investigated for Each Application Studied

App	Domain	LOC	Modifiable LOC	No. of Modifications
7zip	Compression/decompression	136,828	2,524	5,000
Ferret	Image Search-Engine	13,260	5,032	11,000
Bodytrack	Body tracking	3,020	1,030	2,000
OMXPlayer	Media Player	14,164	5,184	10,000
Total				28,000

text file. We chose to optimise these files exclusively due to their dominant role in the application. These files contain 6,258 lines of C code, 2,524 of which are modifiable in the Langdon format. For our experiments we generate a modification set consisting of 5,000 modifications.

7zip is evaluated by measuring the total energy required to compress and then decompress a testcase. For a testcase to pass, the testcase must be compressed and then decompressed to its original state. The approximation value is calculated using the compression ratio. Equation (1) shows how this approximation value is calculated. The original program P compresses a testcase, t, with the size of the compressed testcase recorded (we overload $|\cdot|$ to denote file size). We do the same with the modified program P'. The approximation is the size of the compressed file produced by the modified program divided by the compressed file produced by the original. This ratio has one subtracted so that a value of zero is returned when there is no change in compression rates. A higher approximation value indicates worse compression while a lower approximation value indicates better compression in the modified software.

We use 40 testcases to evaluate 7zip: 10 audio files, 10 text files, 10 image files, and 10 large files. The latter includes files and directories which range from 22.2 MB to 64.4 MB while the other three categories contain files with sizes ranging from 546 KB to 12 MB. These testcases covered 42 percent of all modifiable statements. We ran 7zip using `./7za a test.7za {test}` to compress and `./7za x test.7za -o ./output/` to decompress.

$$\frac{|P'(t)|}{|P(t)|} - 1. \tag{1}$$

5.3 Ferret

Ferret is an image search engine. The program takes an image database and an image query as inputs. It then searches the databases for images similar to the input image and returns the top candidates ranked by relevance (the number dependent on configuration). Ferret is part of Princeton's PARSEC Benchmark Suite [10] and has previously been used as a candidate for genetic improvement at the machine-code level by Schutle et al. [55]. We are using the most up-to-date version of Ferret at the time of writing; that contained within Parsec 3.0. Ferret is made up of 52 C/C++ files (excluding libraries) which contain 13,260 lines of code. When the Langdon format is used, 5,032 lines of code are deemed as modifiable. Due to Ferret's relatively small size, we have chosen to optimise the entire application. We generate a modification set consisting of 11,000 modifications.

We use the 'simlarge', 'simmedium', and 'simsmall' testcases provided as part of the PARSEC Benchmark Suite. 'simlarge' runs 256 image queries on a database of 34,973; 'simmedium' runs 64 images queries on a database of 13,787 images; and 'simsmall' runs 16 image queries on a database of 3,544 images. Without alteration, these return the top 10 from the ranking. In our work, we increase this so that the top 50 are returned to achieve greater granularity in the approximation value. For a testcase to pass, a non-null ranking must be returned by the application. We found our testcases covered 41 percent of all modifiable statements.

The calculation of the approximation value is shown in Equations (3) and (4). The output rankings produced by the original software P is compared against that produced by the modified software P'.

For each query ($q \in Q$), the ranks are compared using Kendall's τ ranking statistic. For equal rankings, Kendall's τ returns one and tends to negative one for more unequal rankings produced. Our approximation value rules require zero to be returned when no approximation has taken place and tend higher for more approximate solutions. Equation (3) manipulates the Kendall's τ statistic to conform to this by incorporating a stretch factor s which we set to 5,000. This results in K, our modified ranking statistic, ranging from 0 for equal rankings to 10,000 for completely unequal rankings. The interval arithmetic [26] for K is shown in Equation (2). The approximation value is the averaged over all queries. If an image was ranked in the top 50 for the original output but not the modified output then it is added to the end of the modified ranking. We run Ferret using `./parsecmgmt -a run -p ferret -i {test}`.

$$\begin{aligned} \tau(L_1, L_2) &= [-1, 1] && \#Range\ of\ \tau \\ [0, 2] &= [-1, 1] + [1, 1] && \#Shift\ interval \\ [0, 2s] &= [0, 2] * [s, s] && \#Stretch\ by\ s \in \mathbb{R}^+ \end{aligned} \tag{2}$$

$$\begin{aligned} &2s - s(\tau(L_1, L_2) + 1) \\ &= [2s - 2s] - [0, 2s] \qquad \#Complement\ by\ 2s \end{aligned}$$

$$K = 2s - s(\tau(L_1, L_2) + 1) \tag{3}$$

$$\frac{1}{|Q|} \sum_{q}^{Q} K(P(q), P'(q)) \tag{4}$$

5.4 Bodytrack

Bodytrack is a computer vision application that tracks a human body through an image sequence. The application is capable, without markers or human involvement, to recognise body position from an array of cameras over a series of frames. It then adds boxes to these frames to mark the body to produce a human-readable output. It is part of Princeton's PARSEC Benchmark Suite [10], version 3.0. Excluding

libraries, Bodytrack consists of 23 C++ files that, in total, contain 3,020 lines of code. When the Langdon format is applied 1,030 lines of code are modifiable. A modification set of 2,000 was created to investigate Bodytrack.

Bodytrack comes with three test sets: 'simsmall', 'simmedium', and 'simlarge'. The 'simsmall' test set consists of 4 cameras, each of which take 1 frame of footage. 'simmedium' has 4 cameras and takes in 2 frames of footage. 'simlarge' has 4 cameras and takes in 4 frames of footage. The output for each is a series of points which can be plotted on the input frames to highlight the location of a body within it. For a testcase to be passed, Bodytrack must return the same number of points (of non-null value) as the original, unmodified application. These testcases covered 66 percent of all modifiable statements.

Algorithm 3 calculates Bodytrack's approximation. It averages the differences between the points produced by the original software, l_P, and the points produced by the modified software l'_P (both contain $1 \ldots N$ points) for any given input. The difference between two points is the sum of the difference in the x component plus the difference in the y component. An approximation value of zero means the output is identical to the original and gets higher as the results become more approximate. We run Bodytrack using `./parsecmgmt -a run -p bodytrack -i {test}`.

Algorithm 3. Bodytrack's Approximation Calculation

1: $l_P \leftarrow P(t)$
2: $l'_P \leftarrow P'(t)$
3: assert $(l_P.\text{ size }() == l'_P.\text{ size }())$
4: $N \leftarrow l_P.\text{ size }()$
5: **while** $l_P \neq \{\}$ **do**
6: $\quad (x, y) = l_P.\text{ dequeue }()$
7: $\quad (x', y') = l'_P.\text{ dequeue }()$
8: $\quad s \leftarrow s + |x - x'| + |y - y'|$
9: **end while**
10: **return** $\frac{s}{N}$

5.5 OMXPlayer

OMXPlayer [3] is a Video Player operated via command-line interface. It takes in a video file and outputs the necessary data to the HDMI port. Of particular interest for this investigation is that OMXPlayer has been specifically designed with the Raspberry Pi hardware in mind, taking advantage of the Raspberry Pi's GPU. It thereby differs from the other candidates that exclusively interact with the traditional computer architecture.

OMXPlayer consists of 14,164 lines of code spread over 24 C++ files. This excludes the `FFmpeg` package which, though included in the source code and necessary for execution, functions as a third-party library to the application. The number of lines tagged as modifiable using the Langdon format is 5,184 and a modification set of 10,000 modifications was generated.

The tests for OMXPlayer consist of MP4 video clips gathered from https://archive.org. The videos' average length is 14.7 seconds with a minimum of 13.0 seconds and maximum of 15.0 seconds. These testcases covered 38 percent of all modifiable statements. In order to evaluate modified versions of OMXPlayer, the application is modified to copy the data that would be sent through the HDMI interface to a text file; one HDMI packet per line. As this writing to file may have some impact on energy consumption, all OMXPlayer variances are run twice. Once with the HDMI-to-text-file functionality and again without. The latter is when the energy measurement is taken; the former is used to evaluate the approximation value. For a testcase to pass a non-null output must be written to the text file.

Equation (5) shows how the approximation value is calculated. As can be seen, the approximation value for any given test is calculated by taking the number of lines returned by a POSIX `diff` on the output generated by the original software, L_P, in comparison with the output of the modified software, L'_P, for a given testcase. This is then divided by the total number of lines the original output. Therefore, zero is returned when the outputs are identical and tends higher the more approximate the output becomes. We use this as a proxy for video quality. The more HDMI packets that differ from the original, the more lines will be returned by POSIX `diff` and the higher the approximation value will be. We run OMXPlayer using `./omxplayer -p -o hdmi {test}`.

$$\frac{|\texttt{diff}(L_P, L'_P)|}{|L_P|}. \tag{5}$$

6 Results

We measure energy consumption for both the original and all the 28,000 software variants (see Table 1) of the four test subjects presented in Section 5, using the methodology outlined in Section 3. Having thereby identified a set of effective (i.e., energy reducing) modifications, we uniformly sample 15 percent of all possible pairwise combinations. We apply these, in turn, to the four applications under test and measure the energy consumption to check for synergistic or antagonistic effects. We summarise our results and answer the research questions posed in Section 4 as follows.

6.1 RQ1: Measurement

The first research question is concerned with the reliability of energy measurements within our Raspberry Pi cluster. In particular, we ask: *what variance occurs when measuring energy consumption?*

RQ1a asks "*What is the variance when measuring using a single energy measurement device?*". To answer this, we plot the variance in energy measurement, within a single device, across all applications studied. This results in the box-plots found in Fig. 6. The mean for 7zip is 48.45J with a standard deviation of 0.31, for Bodytrack, 100.19J with a standard deviation of 1.17, for Ferret, 363.25J with a standard deviation of 1.26, and for OMXPlayer, 57.17J with a standard deviation of 0.18. We therefore conclude that the precision of the energy measurement setup is sufficiently high for the needs of our investigation.

RQ1b asks "*What is the variance in direct energy measurements across multiple devices?*". To answer this we measure the energy consumption of each node running 7zip (chosen uniformly at random from the four applications studied) on a single, uniformly chosen testcase (in this case, 'The Complete Works of William Shakespeare', a 5.6 MB file). The box-

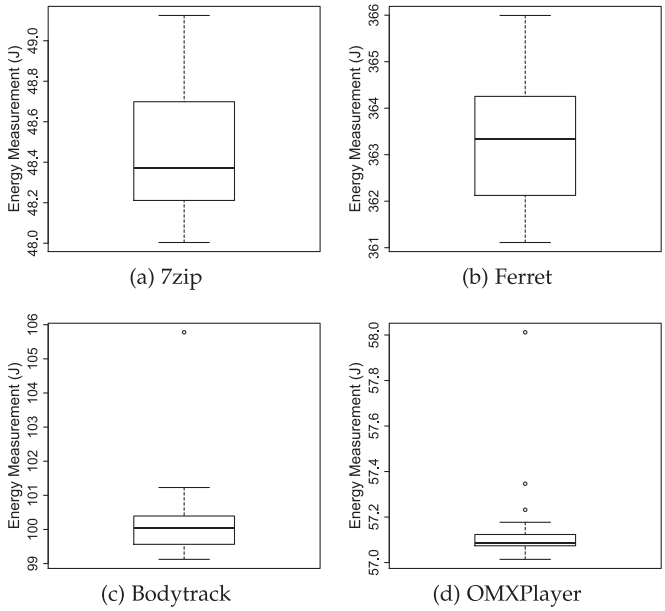

Fig. 6. The variance in measurements that occurred when running each application (unmodified) 30 times on the same device on a uniformly chosen testcase.

plots in Fig. 7 show the distribution of energy readings for each Raspberry Pi device. As can be seen, the measurements vary noticeably between devices but are consistent within each device. In answering this research question, we also observed that restarting a node in the cluster can result in different readings compared to those given before its restart. This did not interfere with our experiments as readings between restarts were consistent. A survey of the relevant literature unearthed an analysis by Kalibera et al. [29] which describes this phenomenon as a little known but none-the-less near-universal problem when taking measurements of modern computer systems. The significant differences that can occur when rebooting is primarily due to non-deterministic properties in modern operating systems, particularly that of memory

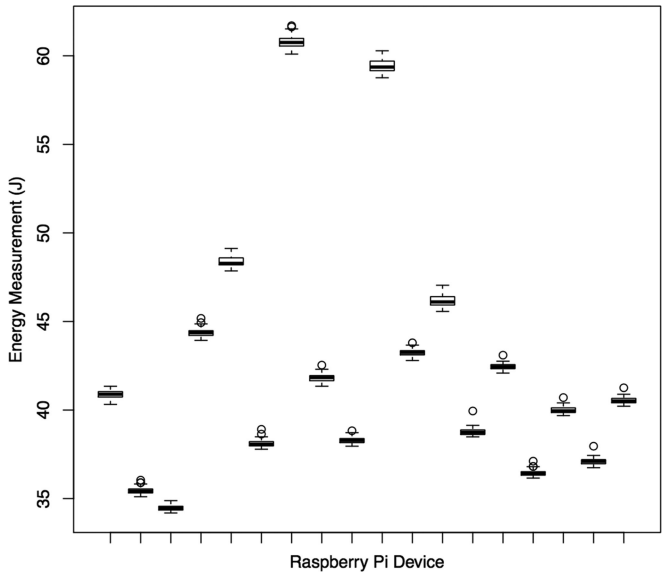

Fig. 7. The variance in measuring the same program with the same input across different devices.

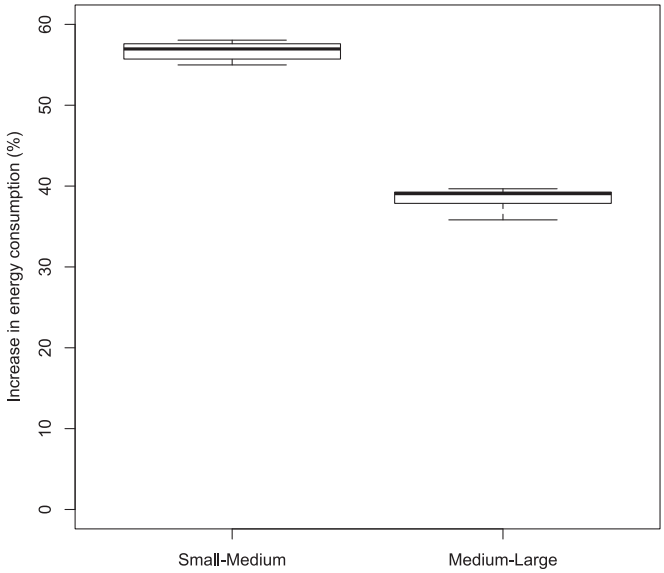

Fig. 8. The variance in proportional energy change across different devices for Bodytrack's simsmall, simmedium, and simlarge testcases.

management which can have a knock-on effect on cache hit-rates. In line with our findings, Kalibera et al. note that measurements are consistent between restarts.

Therefore, we conclude that this approach to measuring energy consumption produced results that *lack accuracy but have good precision* [58]. As Fig. 7 shows, it is impossible for each reading to be accurate, but within each device, the readings are precise. This inaccuracy, however, is only relevant if we want to obtain energy reductions or increases in Joules which, in this investigation, we do not. In our investigations, we wish to obtain the proportional change between software variants and, therefore, it is important to show that the proportional change observed in one device is consistent across all devices.

It is for this reason RQ1c asks "*What is the variance in proportional energy changes across multiple devices?*". To answer this, we ran Bodytrack (uniformly chosen from all applications studied) with each of its three testcases ('simsmall', 'simmedium', 'simlarge') on all devices in the cluster and recorded the proportional increases in energy consumption between simsmall and simmedium, and simmedium and simlarge for each device within the cluster.

The box-plots in Fig. 8 show these proportion increases between the three Bodytrack testcases. The difference between the simsmall and simmedium averages 56.64 percent with a standard deviation of 1.03, and the difference between simmedium and simlarge averages to 38.46 percent with a standard deviation of 1.01. Given this small standard deviation, we believe that proportional change in energy consumption between two measurements within the same device is consistent across all devices.

6.2 RQ2: Improvement

We are concerned with the trade-off between energy consumption and solution quality produced by modified software, which we obtain using the *delete*, *copy* and *replace* search operators. Therefore, we ask: *what additional energy improvement can be achieved when using approximate test oracles in place of exact test oracles?*

TABLE 2
Each Application with the Number and Percentage of Modifications That Reduced Energy Consumption According to an Exact Test Oracle

Application	+ve Mods	%age Mods	Max	Average
7zip	0	0.00%	N/A	N/A
Ferret	0	0.00%	N/A	N/A
Bodytrack	6	0.30%	2.69%	1.54%
OMXPlayer	4	0.04%	2.32%	1.51%
Average	2.5	0.09%	1.25%	0.76%

The average and maximum magnitude of these modifications is also included.

In order to obtain a baseline measurement, we first investigate the question: *what is the frequency and impact of energy-efficient modifications in the local neighbourhood when using exact test oracles?* To answer this, we extract the data generated from the experimental procedure outlined in Section 3. We only consider a modification to be successful when it, on average, reduces energy consumption across 30 runs with this effect observed to be statistically significant ($p < 0.05$ according to the Mann-Whitney U test) for each testcase. As we use exact test oracles in answering this research question, we exclude any modifications that have an approximation value not equal to zero.

Table 2 shows the results obtained to determine the frequency and magnitude of effective modifications in the local search space (i.e., defined by the *delete*, *copy* and *replace* operators), assessed using the exact test oracles. The most striking finding is the frequency of modifications that reduce energy consumption (i.e., '+ve mods'); averaging only 0.09 percent across all cases. The impact of these is an average decrease of 1.25 percent across all applications with a maximum of 2.69 percent. This is a striking finding as it indicates only small improvements can be found in the one-step local neighbourhood when using an exact test oracle.

Table 3 reports results obtained when approximate outputs are permitted. We analyse the same dataset but allow modifications with a non-zero approximation value. As discussed in Section 5, a higher approximation value for 7zip means less compression; for Ferret, a greater inaccuracy in the search engine result rankings (using the original ranking as the baseline); for Bodytrack, a larger error in the plotting of the body's location within a series of images; and for OMXPlayer, a greater proportion of incorrect, or misplaced, HDMI packets. As our approximate oracles permit a significant degradation in output quality, it should be noted that the solutions which make up the averages in Table 3 may not be applicable in many real-world environments. As an example, a valid approximation for 7zip could be an alteration which results in it producing compressed files of equal or greater size than what was input. Evidently such a result would not be of use in any conceivable domain. What we wish to demonstrate here is that approximation is an avenue for more optimisations; that the more a GI practitioner permits approximation, the more energy saving solutions can be found.

TABLE 3
Each Application with the Number and Percentage of Modifications That Reduced Energy Consumption According to an Approximate Test Oracle

Application	+ve Mods	%age Mods	Max	Average
7zip	8	0.16%	48.24%	13.16%
Ferret	157	1.43%	79.88%	51.13%
Bodytrack	72	3.60%	33.69%	8.17%
OMXPlayer	24	0.24%	95.60%	63.15%
Average	65.25	1.36%	64.35%	33.90%

The average and maximum magnitude of these modifications is also included.

We found that approximation increased the frequency of effective modifications in the local search space to 1.36 percent; a 15-fold increase compared to those found when using the exact test oracle. This increase in frequency was mirrored in the increase in impact the average modification was capable of producing. While the average effective modification energy consumption reduction when using the exact test oracles was 1.25 percent, an average reduction of 33.90 percent was achieved when using approximate test oracles.

As previously mentioned, these statistics assume any level of approximation is acceptable. Therefore, Table 4 provides Pareto fronts for each of the applications investigated which show the energy reduction versus output quality trade-off. The approximation value in each case is calculated using the formulæ presented in Section 5. For each, we describe what each Pareto optimal solution produces when run. We also

TABLE 4
Pareto Fronts for the Four Subjects Investigated Showing Trade-Off Between Energy Consumption and Solution Quality

(a) 7Zip

Energy Reduction	Approximation Value
5.08%	3.93×10^{-4}
12.29%	0.072
13.17%	0.102
48.30%	0.741

(b) Ferret

Energy Reduction	Approximation Value
43.19%	0.154
60.79%	39.873
75.53%	78.800
75.55%	1550.200
76.21%	2669.710
79.88%	6221.220

(c) Bodytrack

Energy Reduction	Approximation Value
2.69%	0.000
19.26%	0.131
27.97%	0.170
29.13%	0.192
33.69%	0.452

(d) OMXPlayer

Energy Reduction	Approximation Value
2.32%	0.000
78.45%	0.003
92.70%	0.637
95.53%	1.002
95.60%	1.043

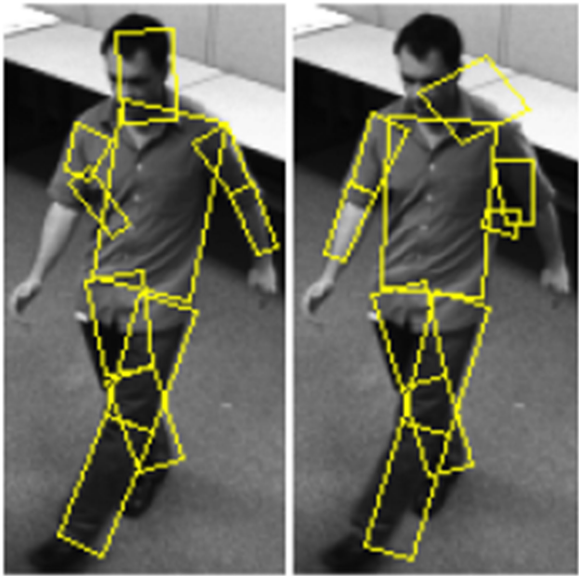

Fig. 9. The output from two versions of the Bodytrack application. The image on the left is generated from the original application and the image on the right is generated from a modified variant. The 'body tracking' on the right is approximated but takes 33.69 percent less energy to compute.

sample a single approximate solution for each application and attempt to explain why the modification reduced energy consumption and reduced the output quality.

In the case of 7zip, the Pareto front contains 4 solutions, ranging from a 5.08 percent energy reduction with an approximation value of 3.93×10^{-4} to a 48.30 percent reduction with an approximation value 0.741. The latter translates to the compressed file generated by the modified software being, on average, 74 percent larger than if compressed using the unmodified version of 7zip. This variant of 7zip is still performing non-lossy compression but less effectively. We analysed this solution in greater detail and found the modification deleted a line in `LzmaEnc.c`, which initialised the a variable declaring dictionary size (an unsigned 32 bit integer) to zero. There is then a method that follows which sets the dictionary size variable to 8 MB *if* the dictionary size variable was initialised to zero. Otherwise, the dictionary size variable is kept at its specified non-zero value. As uninitialised integers produce undefined behaviour in C, the value of the dictionary size varies between runs though, from our observations, this was always significantly below the 8MB figure. The highest we observed was 4 KB. A lower dictionary size inevitably leads to less compression, a shorter execution and, therefore, less energy consumed overall.

For Ferret, there are 6 Pareto optimal solutions. This ranges from a 43.19 percent reduction in energy, for an approximation value of 0.154, to a 79.88 percent reduction in energy, for an approximation value of 6221.220. In the former case, the approximation value translates to a Kendall's τ of 0.73. The next solution on the Pareto front achieves a 60.79 percent energy reduction with an approximation value of 39.873. This approximation value translates to a Kendall's τ of -0.95, a value close to the Kendall's τ 'worst case' of -1. Therefore, five of Ferret's six Pareto optimal solutions produce solutions close to random (i.e., with very high inaccuracy). We sampled the solution with a 43.19 percent energy reduction and an approximation value of 0.154. This modification applied the *delete* operator to the condition of a `FOR` loop within `emd.c` (i.e., turned it to `false`). This turned off an energy intensive branch within Ferret's EMD (Earth Mover's Distance) algorithm. The EMD algorithm computes the distances of colour histograms, which Ferret uses as a metric to show how similar two images are. With this modification, Fthis distance is more approximate and therefore leads to a different ranking of images output for a given input.

Bodytrack has 5 Pareto optimal solutions. These range from a 2.69 percent reduction where there is no approximation to a reduction of 33.69 percent with an approximation value of 0.452. Fig. 9 shows the most energy-efficient solution's output compared to that produced by the original, unmodified software. We sampled the instance with a 33.69 percent energy reduction and a 0.452 approximation value. Bodytrack functions by iteratively generating models (configurations of the wire-frame body, like that shown in Fig. 9), assigning them weights proportional to their likeness of the body within the image. A subset of models are selected proportional to these weights and these models are 'mutated' by incrementing their parameters by random amounts as determined by the Gaussian distribution. This process is repeated until the maximum number of iterations is met or until the search converges on a model that has a sufficient likeness to the body within the image. Bodytrack determines the likeness of a model to the input image via an error function. The modification responsible for the 33.69 percent reduction in energy consumption deletes a line in Bodytrack's `ImageMeasurements.cpp` file which interferes with how this error is calculated, effectively lowering it. This allows Bodytrack to converge on a more approximate model earlier, reducing execution time and, by extension, energy consumption.

Finally, the OMXPlayer Pareto front contains 5 Pareto optimal solutions. We visually inspected the video output when these modifications were applied. We found the three most approximate solutions did not produce video output which was viewable (a black screen with no audio). The next most approximate solution achieved a 78.45 percent reduction with an approximation value of 0.003. This approximation value means, on average, 0.15 percent HDMI packets differed from the output of the original application. We found that this solution was viewable but played video files at increased speed and with distorted audio. The solution with no approximation and a 2.32 percent reduction in energy consumption, when visually inspected, was identical to the original as expected. We sampled the variant which achieved a 78.45 percent reduction in energy consumption with an approximation value of 0.003. We found that this was due to a *replace* operation in OMXPlayer's `OMXPlayerAudio` class; modifying an `IF` statement's condition, turning its condition to true. This resulted in declaring audio to be 'passed through'. Pass-through is an option available in media players to turn off audio decoding and, instead, output the encoded audio stream unprocessed. This is desirable when the user wishes to offload decoding to more advanced hardware, such as home stereo setups. This explains why audio was so distorted in our inspection of this modification. It also partway explains the energy reduction as this modification removes

TABLE 5
Number of Effective Modifications Using the *Delete*, *Copy* and *Replace* Search Operators Across All Applications

	delete	*copy*	*replace*
7zip	5	0	3
Ferret	81	7	69
Bodytrack	44	1	27
OMXPlayer	8	3	13
Percentage	52.9%	4.2%	42.9%

costly audio decoding. We also found the speed of the video increases due to this modification. The reason for this is less clear but appears to be due to how OMXPlayer processes video along side audio. Each are allocated their own thread and, to ensure these threads remain in-sync, the video is sped up or sped down to keep pace with the audio. Normally, these corrections would be unnoticeable to the user, speeding up or slowing down video stream by a small amount for a small period of time. As this modification results in the audio being passed through instead of processed, the audio thread runs considerably faster and thus the video speed is increased in a futile attempt to remain in-sync. If the user specifies audio pass-through when running the program via the command line interface, checks are done to ensure the audio is passed-through to external decoders. If these checks fail, the user's decision is overridden. This modification bypasses these checks. We believe this increased video speed explains most of the energy reduction as it simply results in the total execution time of the video player decreasing.

Using the data gathered in this investigation, we were able to determine how frequently each of the search operators occur in energy-saving modifications. Forest et al. [20] and Le Goues et al. [38] evaluated the *delete*, *copy*, and *replace* operators in the context of automated software repair. They found that *delete* is the most effective at 'fixing' bugs (Qi et al. have since shown that many of these 'fixes' reduced symptoms rather than repaired bugs [54], however, this form of pseudo-repair may be sufficient in some circumstances), followed by *replace* with *copy* being considerably less successful. We find this trend largely holds when applied to genetic improvement for energy consumption. Table 5 shows the frequency of effective modifications, for each application studied, broken down by operator type. *delete*, followed closely by *replace*, are most likely to produce an effective solution. Table 6 shows the average and median energy reduction per operator type. This table shows that

TABLE 6
The Mean and Median (Bracketed) Impact of Effective *Delete*, *Copy*, and *Replace* Modifications, in Terms of Percent of Energy Reduction, Across All Applications

	delete	*copy*	*replace*
7zip	16.60% (12.29%)	0.00% (0.00%)	7.42% (5.08%)
Ferret	56.45% (73.64%)	64.74% (75.17%)	43.50% (37.23%)
Bodytrack	8.27% (4.70%)	0.16% (0.16%)	8.31% (7.09%)
OMXPlayer	57.01% (71.09%)	71.86% (78.40%)	64.92% (78.44%)
Average	34.58% (40.43%)	46.59% (38.43%)	31.04% (31.96%)

TABLE 7
A 20 Percent Sample of All Effective *Replace* Operations Manually Classified as Either Effective *Delete* Operations, Legitimate *Replace* Operations or Unknown Effect

	Delete-by-proxy	Genuine Replace	Unknown
7zip	0	0	1
Ferret	7	4	1
Bodytrack	4	2	0
OMXPlayer	0	2	1
Total	13 (50%)	10 (38%)	3 (12%)

when copy is effective (albeit rarely as shown in Table 5) it can have the biggest impact, followed by *delete*, then *replace*.

In viewing these results, we were motivated to discover how many of the *replace* operations were, in effect, *delete* operations; *replace* operations in which the same effect could be obtained via a single *delete* operation. We uniformly sampled 20 percent of the effective *replace* modifications (those that reduced energy consumption and passed the tests, with approximation permitted) from each application and manually inspected them.[2] Our manual inspection process involved two software engineering researchers who examined each *replace* operation independently and classified them as either 'delete-by-proxy' when it was decided the same effect was possible via the application of a single *delete* operation, or as 'genuine replace' when this was not possible. We reserved classification 'unknown' if there was insufficient evidence from manual inspection of the source-code to make a decision. Once both manual inspectors completed their classifications, the inspectors met and compared their classifications. Where there were conflicts in a modification's classification, the inspectors discussed their reasoning for their respective classification with the goal of coming to an agreement. In this case, the two inspectors reached agreement on all modifications.

We record the findings of this study in Table 7. We found half of all *replace* operations could have occurred through a single *deletion*, though a substantial minority, 38 percent, were legitimate *replace* operations that could not be recreated through a single use of the *delete* operator. Referring back to Table 5, which shows 52.9 percent of all effective modifications were *delete* and 42.9 percent were *replace* operations, we can add more weight to the argument that *delete* is most effective as half of all *replace* are effectively *delete* operations.

6.3 RQ3: Synergy and Antagonism

This research question asks *How frequently do synergistic and antagonistic effects occur when combining known effective modifications?* We answered this question by uniformly selecting 15 percent of all available pairs of effective modifications (approximation permitted), evaluating them, then classifying them according to our interaction spectrum, as defined in Section 3.4.

Table 8 shows the distribution of the interaction classifications. As can be seen, at 49.5 percent of all pairings, weak antagonism is the most common classification. 12.0 percent of all pairings were found to exhibit synergy though this

2. We sampled a minimum of one, for applications with less than 5 effective *replace* modifications.

TABLE 8
The Percentage of Effective Modification Pairings within the Interaction Spectrum (Definition 3)

App	synergy	weak antagonism	antagonism
7zip	0.9%	60.4%	38.7%
Ferret	9.2%	48.8%	42.0%
Bodytrack	35.3%	40.1%	24.6%
OMXPlayer	2.6%	48.7%	48.7%
Average	12.0%	49.5%	38.5%

figure is skewed by Bodytrack where 35.3 percent of all pairings were classified as having a synergistic interaction.

In total, 38.5 percent of modification pairings produced antagonistic behaviour. To obtain the most optimal solutions, it is evident that effective modifications must be selected carefully and therefore greedy approaches will rarely produce superior solutions. Though 61.5 percent of modification pairs are worth combining, the rate of antagonism is high. For this reason, we advocate the usage of evolutionary search techniques such as GAs. They are capable of building complex solutions by testing the interaction of components. Bad interactions incur a fitness penalty thereby disincentivising their combination in the population even when individually they are of benefit.

We chose to investigate an instance of synergy and an instance of antagonism to better understand the phenomenon. 7zip has one instance of 'synergy' and was the first found in this investigation. We therefore dedicated some time to investigating the synergistic interaction.

The two modifications responsible for synergy in the case of 7zip altered lines in two separate classes, `LzmaEnc.c` (shown in Fig. 10) and `LzFind.c` (shown in Fig. 11). We monitored the control flow of the `LzmaEnc.c` class when running without any modification, modification just in `LzmaEnc.c`, modification just in `LzFind.c`, and, finally, when both classes were modified. We observed that the control flow in `LzmaEnc.c` is the same whether the `LzFind.c` modification is applied exclusively or no modification is applied. When the `LzmaEnc.c` modification is solely applied it results in the skipping of a large portion of a frequently iterated `FOR`-loop. This is shown Fig. 10 where line 40 is deleted thereby leaving `numAvailFull` uninitialised. In our analysis this resulted in the `IF` condition at line 46 being true at a much higher frequency than it would when `numAvailFull` was initialised. When both modifications are present, the `LzmaEnc.c` maintains this behaviour but, in addition, the likelihood of returning at Line 35 also increases.

The modification in `LzFind.c`, in Fig. 11, achieves this additional, synergistic behaviour. The modification, the application of a *delete* operation at line 40, turns `while (++len != lenLimit)` to `while (false)` in the `Hc_GetMatchesSpec` method. We found this results in the value returned by `Hc_GetMatchesSpec` having a higher likelihood of being a lower value. This is passed, via `GetMatches`, to the `ReadMatchDistance` method in `LzmaEnc.c` (Fig. 10). This value is then used to calculate a variable, `lenRes`, the value of which `ReadMatchDistance` eventually returns and assigns to `lenEnd` in `getOptimum` at line 26. Though other factors feed into this

```
1  static UInt32 ReadMatchDistances(CLzmaEnc *p,
2    UInt32 *numDistancePairsRes)
3  {
4    UInt32 lenRes = 0, numPairs;
5
6    ...
7
8    numPairs = p->matchFinder.GetMatches(
9      p->matchFinderObj, p->matches);
10
11     /*
12      * Calculation determining
13      * the value of lenRes
14      */
15
16   ...
17
18   return lenRes;
19 }
20
21 static UInt32 GetOptimum(CLzmaEnc *p,
22   UInt32 position, UInt32 *backRes)
23 {
24   ...
25
26   lenEnd = ReadMatchDistances(p, &numPairs);
27
28   ...
29
30   for(;;){
31     UInt32 numAvailFul;
32
33     cur++;
34     if (cur == lenEnd){
35       return Backward(p, backRes, cur);
36     }
37
38   ...
39
40     numAvailFull = p->numAvail; //Mod: Delete
41     UInt32 temp = kNumOpts - 1 - cur;
42     if (temp < numAvailFull){
43       numAvailFull = temp;
44     }
45
46     if (numAvailFull < 2)
47       continue;
48
49   ...
50
51   }
52 }
```

Fig. 10. Synergy Modification (`LzmaEnc.c`).

calculation, a lower value returned by `GetMatches` results in a lower value returned by `ReadMatchDistances`. Ultimately, the smaller the value of `lenEnd` the quicker the return statement is encountered at line 35.

We find this reduces the execution frequency of the more expensive false branch of the `IF` statement, at line 34, by 15.7 percent. Individually, the `LzmaEnc.c` modification achieves a 24.7 percent reduction in energy consumption. Likewise, in the case of the `LzFind.c` modification a 17.5 percent reduction in energy reduction is found. When both are combined a 43.4 percent reduction is achieved. The synergistic effect results in an 'additional' saving of 1.2 percent.

In a similar vein, we investigated an antagonistic reaction. In Bodytrack, we found two modifications that both deleted parameter declarations in Bodytrack's `CameraModel.c` class. One deleted `mc_ext(1,1) = Rc_ext[0,1]`

```
1  UInt32 GetMatches(CMatchFinder *p,
2    UInt32 *distances)
3  {
4    UInt32 offset;
5    curMatch = p->hash[hashValue];
6    offset = (UInt32)
7      (Hc_GetMatchesSpec(lenLimit,
8      p->hash[hashValue],
9      MF_PARAMS(p),
10     distances, 2) - (distances));
11   return offset;
12 }
13
14 static UInt32 * Hc_GetMatchesSpec(
15   UInt32 lenLimit,
16   UInt32 curMatch, UInt32 pos,
17   const Byte *cur, CLzRef *son,
18   UInt32 _cyclicBufferPos,
19   UInt32 _cyclicBufferSize,
20   UInt32 cutValue,
21   UInt32 *distances, UInt32 maxLen)
22 {
23
24   ...
25
26   for (;;)
27   {
28     UInt32 delta = pos - curMatch;
29     if (cutValue-- == 0
30         || delta >= _cyclicBufferSize)
31       return distances;
32     const Byte *pb = cur - delta;
33     curMatch = son[_cyclicBufferPos
34          - delta
35          + ((delta > _cyclicBufferPos)
36          ? _cyclicBufferSize : 0)];
37     if (pb[maxLen] == cur[maxLen] && *pb == *cur)
38     {
39       UInt32 len = 0;
40       while (++len != lenLimit) //Mod: Delete
41         if (pb[len] != cur[len])
42           break;
43       if (maxLen < len)
44       {
45         *distances++ = maxLen = len;
46         *distances++ = delta - 1;
47         if (len == lenLimit)
48           return distances;
49       }
50     }
51   }
52
53   ...
54
55 }
```

Fig. 11. Synergy modification (`LzFind.c`).

and another deleted `mc_ext(1,2) = Rc_ext[1,2]`. Known as the 3D Displacement Matrix, `mc_ext` is utilised frequently in Bodytrack. In both cases, when these matrix values are left undefined, energy consumption reduces. When applied together, a smaller energy consumption is measured than when either is applied individually. We cannot fully explain this effect as both modifications work by leaving these matrix values as uninitialised; undefined behaviour in C. However, the effect on energy consumption differs depending on whether either or both values are uninitialised. We found that application of the first *delete* operation alone results in a 17.6 percent reduction in energy consumption. The second *delete* operation, when applied alone, reduces energy consumption by 13.8 percent. However, when applied simultaneously, we found energy reduces by only 11.6 percent, lower than when either are applied individually.

7 Discussion

In RQ1, we showed that the measurement framework we used in this investigation is sufficient to understand the energy optimisation search space in GI. However, we showed that our energy measurement framework did not produce reliable results between devices. We also observed that, in line with observations from other researchers [29], node restarts can affect energy readings. Despite this, we have shown the *proportional* difference in energy readings between devices are consistent, and believe it is important that those working in energy optimisation research are aware of these issues. The seemingly simple task of being able to measure what is being optimised remains a significant hurdle in energy optimisation research [11] and, thus, care and consideration is needed when planning experiments. The Raspberry Pi/MAGEEC board setup is an abstract representation of 'real-world' systems, which researchers can use to gain insight into energy consumption in a manner which is controlled, low cost, and easily expandable. We acknowledge that more accurate results may be obtained with more expensive devices, though this would come at the cost of having fewer devices running in parallel, limiting the amount of data that may be gathered in a given timescale. We believe the setup we use can be used in a variety of other energy optimisation activities, such as training models which could be translated, via transfer learning techniques [50], to optimise more advanced software systems.

With this energy measurement setup, we analysed what was possible with the application of a single modification in RQ2. We conclude from this data that the *delete*, *copy*, and *replace* operators are largely ineffective at optimising energy consumption with only 0.09 percent capable of producing a statistically significant reduction in energy consumption while preserving output quality in GI. Evaluations of code modifications are typically costly, as checks must be done to ensure functionality has been preserved. On top of this, testing must determine the modifications' effect on the target property (in our case, energy consumption). We have shown that less than one in every thousand modifications is effective. Though evaluations of ineffective modifications is expected in GI, this rate is extremely high. Some practitioners of GI who target energy optimisation have had success with more bespoke operators, such as swapping of Java Collection implementations [44], or alteration of colours in GUI interfaces to reduce the energy consumption of OLED Smartphone screens [42]. We argue this is a good avenue for research as our findings suggests the 'standard set' of operators discussed in this paper are not very effective.

We have shown that permitting a degradation in output quality can aid in the optimisation of energy efficiency. We observe that the number of modifications that can reduce energy consumption increase from 0.09 to 1.36 percent when approximation is permitted; a 15-fold improvement in the number of effective modifications. We appreciate that a lot of these approximate solutions are undesirable and that, in most cases, there are limits on how approximate an output can be. However, we have found no evidence that permitting approximation has any negative effect on reducing energy consumption. Many multi-objective optimisation

methods are available [45] and have already seen adoption in genetic improvement research [13], [14], [42], [62]. We believe integrating 'output quality' as an objective can significantly benefit future projects.

In answering RQ3, we explored the wider search-space by analysing the effectiveness of combining modifications. Our analysis shows that both synergy and antagonism are present in the search space. If there was low antagonism, we could advocate a greedy approach as the combination of any effective two modifications rarely produced a non-linear negative effect. However, we did not observe this. In fact, 38.5 percent of modifications produced some form of antagonism. For this reason, we advise more advanced search techniques such as genetic algorithms, though any search that effectively tests combinations of modifications would be sufficient. Simply combining any and all effective modifications will not produce the sums of their parts in all cases.

In answering RQ1, we showed that, with careful analysis and understanding, we can report reliable results. We could have reduced variance more by running applications on a bare machine, thereby removing interference that may emerge from the operating system. This may produce results of greater interest as running on a bare machine is more common in embedded systems which, unlike the Raspberry Pi devices studied, may be battery powered, making the goal of reducing energy consumption more important. The 'Internet of Things' is likely to constitute of many small embedded systems powered by batteries that are expected to run for a certain time before depletion. It would therefore be useful to explore this area in future; however, in this work, we focused on a more typical architecture — that with an application working on top of an operating system. The primary motivation for this is that there is considerably more open-source software available for optimisation in such an environment.

Our measurement cluster can expand indefinitely, thus allowing more modifications to be evaluated in parallel. We have explored a very small area of each respective search-space. In future work, it may be of value to explore the wider search space (i.e., interaction between many modifications).

8 Threats to Validity

The work presented here uses direct energy measurements. Though this results in more reliable evaluations compared to the ones based on simulation, these direct energy measurements inevitably also contain variance. We have quantified this in answering RQ1 but it means that modifications which produce very small but positive changes are undetectable. While we detected energy decreases as small as 0.009 percent, there may be modifications that produce even smaller changes that are simply undetectable with our framework. Our investigations, however, show that modifications which produce detectable, non-trivial, energy reductions are rare.

In this investigation, we have been careful to ensure that any modifications reported as being effective truly are. To achieve this aim, our requirements for what constitutes an 'effective modification' have been strict. For a modification to be classified as effective, it must produce a solution in which we observe a statistically significant decrease for all testcases. While we believe this to be the most honest approach to presenting the data, it may not be representative of real-world genetic improvement where modifications can be seen as effective if they cause improvements in only a proportion of testcases. Determining at what point we may classify a modification as effective is subjective and thereby left to the GI practitioner's discretion. We have chosen to be strict rather than risk being too lenient, thereby avoiding publication of results that may not be applicable to all those in the GI research community.

All the applications we have chosen to study have user-level parameters which may be modified. Some of these parameters may be used to further approximate output quality while reducing energy consumption. We have not experimented with traditional application parameter tuning and have therefore not made comparisons between the results presented here and what may be achievable through other techniques. Such a study is outside of this investigation's scope but we acknowledge there are other established methods to trading application output quality for reductions in energy consumption.

As we only target Raspberry Pi devices running the Raspbian OS, we cannot ensure that the conclusions drawn from this investigation are universal across all software systems. We acknowledge that results are likely to be different when investigating hardware that utilises complex I/O components such as wireless network interfaces which are known to consume large amount of energy in mobile devices [40]. More research will be needed to investigate such platform and system-specific variations.

We chose to investigate the *copy*, *delete* and *replace* search operators because of their frequent use in state-of-the-art genetic improvement work [34], [35], [53]. Other search operators may function better in the context of energy consumption; however, our aim was to investigate the nature of the search space produced in modern GI research.

9 Related Work

While improved hardware performance can ameliorate software systems' energy consumption, recent work on search-based approaches to software improvement has demonstrated that software engineers also have an important role to play. White et al. [61] were among the first to automatically search for modified versions of existing programs to reduce energy consumption, trading functionality for energy reduction. More recently, we have witnessed an explosion of activity in this area.

In this investigation, we have focused on the *delete*, *copy*, and *replace* operators. Though popular [34], [35], [53], there are other methods to modify software to improve energy efficiency. Schulte et al. [55] introduced the Genetic Optimisation Algorithm (GOA) that was shown to reduce the energy consumption of existing software systems by an average of 20 percent. Their investigation modified software systems at the machine code level. A a tool using a similar algorithm was also used to fix bugs at this level [56]. There is an argument that working at lower levels may produce more fruitful results [49], but this introduces changes that can be hard for humans to understand. This is part of the reason most research in GI has focused on human-readable source-code.

Li et al. [42] demonstrated that by sacrificing some degree of usability, energy savings of up to 40 percent could be achieved for (battery-restricted) smartphones. Their approach searched for contrast-preserving changes in screen colours to reduce energy consumed by a smartphone display. In this case, the genetic improvement algorithm toggled the colour settings of an Android application's interface.

Manotas et al. [44] used a constrained exhaustive GIto modify existing open source Java systems, reporting energy improvements ranging between 2 and 17 percent. Their approach used an operator that changed Java Collection API implementations. The *delete*, *copy*, and *replace* operators studied in this investigation work at a higher granularity than Manotas et al.'s course-grained operator, yet Manotas et al.'s approach has been shown capable of reducing energy with considerably less effort. This idea of swapping subclass implementations to find those that are most optimal was later used by Burles et al. to reduce the energy consumption of Google Guava's `ImmutableMultimap` class by 24 percent [16].

Hoffman et al. dynamically tuned parameters to limit power spikes in server clusters [28]. They traded the quality of their applications' outputs in response to the power budget; when power was plentiful, the applications would produce high quality solutions and when power was scarce, the applications would produce lower quality solutions. This work differs from ours in that they tuned parameters directly exposed by software developers that were already known to trade off execution time for output quality. It would be possible to expose optimisations found in source-code to a level in which they may be tuned as if they were 'traditional' parameters. This is part of a growing area within genetic improvement known as 'deep parameter tuning', which has been used to optimise the execution time and memory consumption of C standard library's `malloc` function [62], optimise a face-detection algorithm's execution time while permitting a degradation in its accuracy [13], [14], reduce the energy consumption of Google Guava's `CacheBuilder` class [17].

In GI, it is increasingly common to optimise software for specific hardware targets. Typically, this takes the form of tuning parameters. CLBlast [48] is an example of a library which incorporates an auto-tuning component to optimise its OpenCL BLAS library to the target hardware. Due to this tuning, CLBlast typically outperforms its direct competitor clBLAS, up to a factor of two in some cases. Paone et al. introduced a technique to auto-tune OpenCL kernels for target devices with up to 60 percent performance improvements [51]. Their technique tackles the problem of a large search space by identifying parameter constraints and then developing a feasible subset of parameters. They find this reduced the search space to 0.1 percent of its original size.

Not all changes have to occur in software; hardware itself may be optimised. Zhang et al. [63] observed that up to 50 percent of an embedded system's energy was consumed by cache memory. They noted that a direct mapped cache is more efficient per access than one that is is set-associative but only if the cache hit-rate is high—something which is dependent on the software being run. Similarly, there are decisions to be made regarding the cache's size. Smaller caches are more energy efficient but exhibit poorer hit rates. Zhang et al. resolved this problem by creating a special cache that can have its size set and be toggled as either directly mapped, two-way, or four-way set associative. In evaluating this configurable cache they found it was capable of reducing cache energy consumption of by 40 percent when correctly tuned for specific work-loads.

While optimisations may be made at the software or hardware level, we cannot ignore the benefits of compiler optimisations. GCC has approximately 100 flags exposed for tuning and the optimal combination of these flags differ depending on the target hardware. Typically, compiler optimisation techniques take an iterative approach by setting some compiler flags, evaluating the quality of the compiled product/products, then using this feedback to produce a better set of flags [18], [21], [31]. This approach echoes the iterative approach typical in GI research and, as in GI, this approach is costly as the iterative process is required for every new hardware. In 2011, Fursin et al. introduced 'Milepost GCC' [22], which uses machine learning to build a model that identifies GCC parameters for a hardware target. They find that using Milepost GCC can improve execution time by up to a factor of two in some cases (11 percent on average), without the need for costly iterative processing. Though typically targeting execution time, there has been work into tuning parameters to reduce energy consumption [59].

10 Conclusions

We investigated software system's energy optimisation search space, focussing on three widely used modification operators: *copy*, *delete*, and *replace*. We show that when using exact test oracles, modifications to source code that produce more energy-efficient solutions occur 0.09 percent of the time on average; a flat search space that would be difficult to traverse without a highly explorative search technique, though when approximation of output is permitted this figure grows to 1.25 percent, a 15-fold increase.

In terms of impact, when using the exact test oracle an average decrease of 0.76 percent is observed but when using the approximate test oracle the average impact increases to 33.90 percent. This finding points to the critical importance of approximation for evolutionary energy optimisation. Fortunately, many energy optimisation applications support exactly this kind of optimisation as studied in the work reported here.

We used direct energy measurements to obtain these results and show that the energy measurement framework used in this investigation was precise but lacked accuracy; highlighting an important systemic error in energy measurements. However, we found any proportional energy measurements were reliable and, as such, advocate their use when carrying out such research.

We produced findings which, in line with previous research from non-energy-based optimisation problems, demonstrate that the *delete* and *replace* mutation operators are the most likely to be effective with *copy* modifications rarely producing energy reductions. However, as finding effective modifications was rare for any operator, we advise future researchers to focus efforts into developing more specialised genetic improvement operators for energy optimisation.

We also investigated the effects that energy-efficient modifications produce when combined. We found that

61.5 percent of pairings were worthwhile. That is, the pairing's impact was greater than that of its most effective member with 12.0 percent exhibiting synergy. The remaining 38.5 percent of modification pairs were antagonistic and thereby conclude there is no guarantee that two good modifications will always produce an energy-efficient software variant. It is evident that more advanced search techniques are required.

Acknowledgments

Dr. Justyna Petke is supported by an EPSRC fellowship: EP/P023991/1.

References

[1] 7zip, [Online]. Available: http://www.7-zip.org, Accessed on: Jun. 28, 2017.
[2] MAGEEC Energy Measurement Board, [Online]. Available: http://mageec.org/wiki/Power_Measurement_Board, Accessed on: Jun. 28, 2017.
[3] Omxplayer. [Online]. Available: https://github.com/popcornmix/omxplayer, Accessed on: Jun. 28, 2017.
[4] Raspberry Pi, [Online]. Available: https://www.raspberrypi.org, Accessed on: Jun. 28, 2017.
[5] Raspbian, [Online]. Available: http://www.raspbian.org, Accessed on: Jun, 28, 2017.
[6] International Energy Agency, *Key world energy statistics* OECD Publishing, 2009, doi: 10.1787/9789264039537-en
[7] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proc. Int. Symp. Foundations Softw. Eng.*, 2014, pp. 588–598.
[8] E. T. Barr, M. Harman, P. McMinn, Mu. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.
[9] M. C. Berenbaum, "What is synergy?" *ASPET Pharmacological Rev.*, vol. 41, no. 2, pp. 93–141, 1989.
[10] C. Bienia, "Benchmarking Modern Multiprocessors," PhD thesis, Dept of Computer Science, Princeton Univ., Princeton, NJ, 2011.
[11] M. A. Bokhari, B. R. Bruce, B. Alexander, and M. Wagner, "Deep parameter optimisation on Android smartphones for energy minimisation — A tale of woe and proof-of-concept," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2017, pp. 1501–1508.
[12] BP, "BP statistical review of world energy June 2014," *BP World Energy Review*, 2014, https://www.bp.com/content/dam/bp-country/de_de/PDFs/brochures/BP-statistical-review-of-world-energy-2014-full-report.pdf
[13] B. R. Bruce, "Deep parameter optimisation for face detection using the Viola-Jones algorithm in OpenCV: A correction," Dept. Comput. Scie., Univ. College London, London, U.K., Rep. no. RN/17/07, 2017.
[14] B. R. Bruce, J. M. Aitken, and J. Petke, "Deep parameter optimisation for face detection using the Viola-Jones algorithm in OpenCV," in *Proc. Symp. Search-Based Softw. Eng.*, 2016, pp. 238–243.
[15] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 1327–1334.
[16] 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.
[17] 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.*, 2015, pp. 276–281.
[18] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *J. Supercomputing*, vol. 23, no. 1, pp. 7–22, 2001.
[19] K. A. De Jong, "On using genetic algorithms to search program spaces," in *Proc. Int. Conf. Genetic Algorithms*, 1987, pp. 210–216.
[20] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proc. Genetic Evol. Comput. Conf.*, 2009, pp. 947–954.
[21] G. Fursin, A. Cohen, M. OâĂŹBoyle, and O. Temam, "A practical method for quickly evaluating program optimizations," in *Proc. Int. Conf. High-Perform. Embedded Architectures Compilers*, 2005, pp. 29–46.
[22] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, et al., "Milepost GCC: Machine learning enabled self-tuning compiler," *Int. J. Parallel Program.*, vol. 39, no. 3, pp. 296–327, 2011.
[23] M. Harman and B. F. Jones, "Search-based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.
[24] 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. Int. Conf. Automated Softw. Eng.*, 2012, pp. 1–14.
[25] M. Harman and P. McMinn, "A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 73–83.
[26] T. Hickey, Q. Ju, and M. H. Van Emden, "Interval arithmetic: From principles to implementation," *J. ACM*, vol. 48, no. 5, pp. 1038–1068, 2001.
[27] 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. Conf. Mining Softw. Repositories*, 2014, pp. 12–21.
[28] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *SIGPLAN Notices*, vol. 46, pp. 199–212, 2011.
[29] T. Kalibera, L. Bulej, and P. Tuma, "Benchmark precision and random initial state," in *Proc. Int. Symp. Perform. Eval. Comput. Telecommun. Syst.*, 2005, pp. 484–490.
[30] J. G. Koomey, "Worldwide electricity used in data centers," *Environ. Res. Lett.*, vol. 3, no. 3, 2008, doi: 10.1088/1748-9326/3/3/034008
[31] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast searches for effective optimization phase sequences," *SIGPLAN Notices*, vol. 39, pp. 171–182, 2004.
[32] J. Landsborough, S. Harding, and S. Fugate, "Removing the kitchen sink from software," in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 833–838.
[33] W. B. Langdon and M. Harman, "Evolving a CUDA kernel from an nVidia template," in *Proc. IEEE World Congr. Evol. Comput.*, 2010, pp. 1–8.
[34] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE Trans. Evol. Comput.*, vol. 19, no. 1, pp. 118–135, Feb. 2015.
[35] W. B. Langdon, M. Modat, J. Petke, and M. Harman, "Improving 3D medical image registration CUDA software with genetic programming," in *Proc. Genetic Evol. Comput. Conf.*, 2014, pp. 951–958.
[36] W. B. Langdon, J. Petke, and B. R. Bruce, "Optimising quantisation noise in energy measurement," Dept. Comput. Sci., Univ. College London, London, U.K., Rep. no. RN/16/01, 2016.
[37] W. B. Langdon, J. Petke, and B. R. Bruce, "Optimising quantisation noise in energy measurement," in *Proc. Int. Conf. Parallel Problem Solving Nature*, 2016, pp. 249–259.
[38] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proc. Genetic Evol. Comput. Conf.* 2012, pp. 959–966.
[39] D. Li and W. G. J. Halfond, "Optimizing energy of HTTP requests in android applications," in *Proc. Int. Workshop Softw. Develop. Lifecycle Mobile*, 2015, pp. 25–28.
[40] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An empirical study of the energy consumption of android applications," in *Proc. Int. Conf. Softw. Maintenance Evolution*, 2014, pp. 121–130.
[41] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating source line level energy information for Android applications," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 78–89.
[42] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for OLED smartphones," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 527–538.
[43] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in *Proc. IEEE Int. Conf. Softw. Eng.*, 2016, pp. 237–248.
[44] I. Manotas, L. Pollock, and J. Clause, "SEEDS: A software engineer's energy-optimization decision support framework," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 503–514.
[45] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," *Structural Multidisciplinary Optimization*, vol. 26, no. 6, pp. 369–395, 2004.
[46] M. Mitchell, S. Forrest, and J. H. Holland, "The royal road for genetic algorithms: Fitness landscapes and GA performance," in *Proc. Eur. Conf. Artif. Life*, 1992, pp. 245–254.

[47] V. Mrazek, Z. Vasicek, and L. Sekanina, "Evolutionary approximation of software for embedded systems: Median function," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2015, pp. 795–801.
[48] C. Nugteren, "Clblast: A tuned OpenCL BLAS library," *arXiv preprint arXiv:1705.05249*, https://arxiv.org/pdf/1705.05249.pdf, 2017.
[49] M. Orlov and M. Sipper, "Flight of the FINCH through the Java wilderness," *IEEE Trans. Evol. Comput.*, vol. 15, no. 2, pp. 166–182, Apr. 2011.
[50] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.
[51] E. Paone, F. Robino, G. Palermo, V. Zaccaria, I. Sander, and C. Silvano, "Customization of opencl applications for efficient task mapping under heterogeneous platform constraints," in *Proc. IEEE Des. Autom. Test Eur. Conf. Exhibit*, 2015, pp. 736–741.
[52] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic Improvement of software: A comprehensive survey," *IEEE Trans. Evol. Comput.*, 2017, doi: 10.1109/TEVC.2017.2693219.
[53] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement & code transplants to specialise a C++ program to a problem class," in *Proc. Eur. Conf. Genetic Program.*, 2014, pp. 137–149.
[54] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.
[55] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 639–652, 2014.
[56] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proc. Int. Conf. Automated Softw. Eng.*, 2010, pp. 313–316.
[57] P. Sitthi-Amorn, N. Modly, W. Weimer, and J. Lawrence, "Genetic programming for shader simplification," *ACM Trans. Graph.*, vol. 30, no. 6, 2011, Art. no. 152.
[58] J. Taylor, *Introduction to Error Analysis: the Study of Uncertainties in Physical Measurements*. Herndon, VA, USA: Univ. Science Books, 1997.
[59] A. Tiwari, M. Laurenzano, L. Carrington, and A. Snavely, "Auto-tuning for energy usage in scientific applications," in *Proc. Euro-Par Workshops*, 2011, pp. 178–187.
[60] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Trans. Evol. Comput.*, vol. 15, no. 4, pp. 515–538, Aug. 2011.
[61] D. R. White, J. Clark, J. Jacob, and S. M. Poulding, "Searching for resource-efficient programs," in *Proc. Genetic Evol. Comput. Conf.*, 2008, pp. 1775–1782.
[62] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 1375–1382.
[63] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," in *Proc. Int. Symp. Comput. Archit.*, 2003, pp. 136–146.

Bobby R. Bruce received the BEng degree with 1st class honours in software engineering from Edinburgh Napier University. He is working toward the PhD degree at the University College London's Centre for Research on Evolution, Search, and Testing (CREST). He has worked for both Microsoft and Synopsys as a software developer. During his time at UCL, Bobby has worked in the emerging area of Genetic Improvement, with particular emphasis on the optimisation of software's energy efficiency and automatic parallelisation.

Justyna Petke is a principal research fellow and a proleptic lecturer with the University College London. She has published articles on the applications of genetic improvement. Her work on GI won an ACM SIGSOFT distinguished paper award at ISSTA and two Humie's at GECCO 2014 and 2016 (awarded for human-competitive results). She also has expertise in combinatorial interaction testing and has a doctorate in Computer Science from the University of Oxford in the area of constraint solving. She is supported by an Early Career EPSRC Fellowship.

Mark Harman is currently an engineering manager with Facebook London, where he manages a team, working on Search Based Software Engineering (SBSE) with Facebook Scale. He is also a part time professor of software engineering with the Department of Computer Science, University College London, where he directed the CREST centre for ten years (2006-2017) and was Head of Software Systems Engineering (2012-2017). This work was done while Mark was at UCL full time. He is known for work on source code analysis, software testing, app store analysis and empirical software engineering. He was the co-founder of the field SBSE, which has grown rapidly with more than 1,700 scientific publications from authors spread over more than 40 countries. SBSE research and practice is now the primary focus of his current work in both the industrial and scientific communities. In addition to Facebook itself, Mark's SBSE scientific work is also supported by the ERC and EPSRC funding councils.

Earl T. Barr received the PhD degree from the University of California, Davis. He is a senior lecturer (associate professor) with the University College London. Before joining UCL, he was an Institute for Information Infrastructure Protection (I3P) fellow. Earl has published more than 50 peer-reviewed papers on testing and program analysis, software engineering, and computer security. His recent work focuses on automated software transplantation (Gold medal at GECCO's Humies), the application of empirical game theory to software processes, and the application of NLP and ML to software. His work on time-travel debugging is shipping in Microsoft's Chakra JavaScript engine. Earl has won three ACM distinguished paper awards; his paper entitled "The Naturalness of Software" was a research highlight in the May 2016 Communications of the ACM. Earl dodges vans and taxis on his bike commute in London.