UCL DEPARTMENT OF
COMPUTER SCIENCE

Research Note

RN/17/01

Approximate Oracles and Synergy in Software Energy Search Spaces

25 January 2017

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

Abstract

There is a growing interest in using evolutionary computation to reduce software systems' energy consumption by utilising techniques such as genetic improvement. However, efficient and effective evolutionary optimisation of software systems requires a better understanding of the energy search landscape. One important choice practitioners have is whether to preserve the system's original output or permit approximation; each of which has its own search space characteristics. When output preservation is a hard constraint, we report that the maximum energy reduction achievable by evolutionary mutation is 2.69% (0.76% on average). By contrast, this figure increases dramatically to 95.60% (33.90% on average) when approximation is permitted, indicating the critical importance of approximate output quality assessment for effective evolutionary optimisation. We investigate synergy, a phenomenon that occurs when simultaneously applied evolutionary mutations produce a effect greater than their individual sum. Our results reveal that 12.0% of all joint code modifications produced such a synergistic effect though 38.5% produce an antagonistic interaction in which simultaneously applied mutations are less effective than when applied individually. This highlights the need for an evolutionary approach over more greedy alternatives.

1 Introduction

Reducing energy consumption is an increasingly important software engineering concern. In 2010, large server clusters consumed 1.12%–1.50% of global energy consumption [21]: an amount equivalent to the entire energy consumption of the United Kingdom in 2015 [10]. Environmentally unfriendly sources generate much of this energy: in 2013, 67% of global energy consumption derived from burning fossil fuels, with 41% generated from the most highly-polluting of all sources, coal [6]. Using a variety of search techniques [16], including evolutionary computation, recent studies have shown that the energy consumption of software can be reduced when given reasonable assumptions about the end-use of the improved software system such as the likely input data [11], network usage information [27], and tolerance to less desirable user-interfaces [29].

Modifying a software system using evolutionary computation to improve its energy consumption is an instance of 'Genetic Improvement' (GI) [23, 24, 36, 39]. To genetically improve a program, search techniques modify its generic material to construct related versions that retain some important properties while improving others. The name 'genetic improvement' reveals GI's origins as a form of genetic programming, albeit applied to existing code rather than *ab initio*. GI research, hitherto, has been dominated by three operators: *delete*, *copy*, and *replace* which are applied at the source-code line level [23, 24, 34]. However, until now there has been little effort put to analysing the search space they produce when optimising for energy consumption.

The *delete*, *copy*, and *replace* operators generate a vast search space, bounded soley by the number of *copy* operator applications. Even when restricted to a single operation, the search space remains large. For a program of size N, every line can be deleted (N), copied into the program before existing lines (N^2), or replaced ($N^2 - N$). 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 the application of a single operator. GI techniques typically restrict the search to a designated subset of modifiable target lines. This subset is is typically determined by an expert with intimate knowledge of the system, or via profiling; selecting lines based on a candidate line's likelihood of impacting the target non-functional property. In practice, even this restricted search space remains so vas. The necessity for well-designed search-based techniques is clear though the 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.

We measure energy consumption, focusing 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. Each 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 [29] while a graphics-based application can trade image quality [37]. In such cases, deviation from the precise output of the original may be tolerable when it decreases energy consumption. In a survey of software engineers responsible for systems in which energy consumption is a concern, the majority (80%) were willing to sacrifice certain requirements for reduced energy consumption [30].

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. Tools which estimate energy by logging processor states [11], monitoring bytecode execution [12], or via simulation of hardware [39] are examples of indirect approaches to obtaining energy information. They interpolate energy from correlated measurements. Indirect measurements are typically close actual energy consumption, but the 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, they do not estimate energy through a proxy, they measure it directly. We chose to study the Raspberry Pi, because it is a simple, cheap, widely-available, and easily configurable platform representative of many kinds of hardware platforms.

Our cluster parallelises the measurement of energy consumption across software variants running on different physical devices. This allows us to take many more measurements than we would be able to do otherwise, thereby allowing us to account for statistical error, like 'background noise', by reporting on the averages found over many runs. A key

finding of ours is that individual devices exhibit systematic error. The presence of significant systematic error means that previous results based on the measuring of a single device should be revisited. 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*, *replace* operators. We can analyse both the local neighbourhood (a single modification) and beyond (multiple modifications). This allows us to give guidance on the design search-based algorithms for the optimisation of software using GI.

If we were to find that the local search space is flat (i.e. a single modification is incapable or rarely produces a significant proportional increase 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 (such as a genetic algorithm with a high mutation rate). 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/removing a subset of those changes produces a worse or much less effective solution. We refer to this as *synergy*, a specific instance of epistasis when the improvement of simultaneously applying two modifications exceeds the sum of applying each in isolation. We also observe *antagonism*, another instance of epistasis that may be seen as the opposite of synergy. This occurs when the effectiveness of a solution worsens as modifications are combined in comparison to when they are utilised individually.

Since many search-based approaches to program improvement seek to synthesise large sets of modifications from smaller sets, we want to identify individual improvements that exhibit these forms of epistatic effects. If antagonism is infrequent, then we may advocate a greedy approach to GI; randomly sample modifications, evaluate them, and, if they are found to be effective, then add them to a list of good mutations to then be applied en-masse at the end of the process. However, if antagonism is frequent, we require more advanced approaches. Population-based techniques, such as genetic algorithms (GAs), may be appropriate as they allow alternative schemata [20, Chapter 5] to develop separately in the population then merge, via crossover, without destroying the original individuals. GAs are common in GI as modifications to the source-code can be easily represented as a series of genes with each gene being a distinct modification to the software. Crossover in GAs are a type of antagonism detection mechanism since a child produced by crossover will eventually be evaluated and discarded if antagonism cripples its fitness while its parents, and their independent genetic material, will continue to exist. In our investigation, we find that antagonism occurs in 38.5% of all modification parents thereby justifying advanced search-techniques, such as a genetic algorithm with crossover.

This paper makes three main contributions:

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

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 finding savings in the search space of energy-saving software modifications.

3. Software changes that alter an application's energy consumption exhibit epistasis: some synergistic, others antagonistic. We show that this phenomenon is ubiquitous and implies that 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 understand 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 highlight these concepts.

```
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 (input){
14   input=sort(input); //Mod_2: Delete
15   return input.get(random(0, input.size()));
16 }
```

Figure 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.

In Figure 1, 'Mod_1' swaps a method which aggregates a list with one that samples. This increases the approximation of `getPropery`'s output but may also achieve considerable savings in energy because of sampling's relative efficiency. It is this 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. [34], and later Bruce et al. [11], 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.

It is tempting to pursue the modifications found by the exact oracle exclusively, as they produce benefits without any cost. However, if we permit the quality of output to degrade (i.e. permit approximate output), then this increases the set of valid solutions in the search space and facilitates the search for even more energy-efficient solutions. We are the first to quantify the frequency of these modifications and measure their interactions. Figure 1 demonstrates synergistic software modifications. 'Mod_1' decreases the number of times in which the more energy inefficient method `aggregate` is executed by replacing it with `sample` while 'Mod_2' increases the efficiency of `sample`. Equation 1 explains the basic mathematics of this synergistic interaction. The energy consumed by the program, P, is equal to the sum of the energy consumed by `getProperty`, Y, and `sample`, X, each multiplied by the number of executions, N_2 and N_1 respectively. Activity outside these methods is assumed to be constant and is represented by C. 'Mod_1' changes the control flow of `getProperty` to include a call to `sample`. The energy consumption of the program thereby includes the energy of a single iteration of `sample` plus the remainder of `getProperty` minus the call to `aggregate` (Z).

$$\begin{aligned} P &= XN_1 + YN_2 + C \\ Y &= X + Z \\ P &= X(N_1 + N_2) + ZN_2 + C \end{aligned} \tag{1}$$

When 'Mod_1' is present, 'Mod_2' decreases the energy consumption of `sample` but, by extension, the energy consumption of `getProperty`. Without 'Mod_1', 'Mod_2' is only capable of effecting the energy consumed in `sample`. 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 first 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 an original program and one of its variants, under both exact and approximate test oracles.

3.1 Measurement Framework

Given a set of genetic improvement operators, we seek to measure the effect on the energy consumption of a program their application produces. 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 running a program can be very expensive.

Most previous search-based approaches to optimising the energy efficiency of software have estimated energy consumption [11, 12, 39]. These estimates can miss important high or low energy events thereby directing the search away from an optimal solution. To capture these events, a framework must make 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 systems processes. One can, however, directly measure the energy consumption of the entire computing system; this is what we choose 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. To mitigate against these effects, we take multiple measurements and average 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, across 4 applications, running each variant multiple times against a test suite. Indeed some modification evaluations take up to five minutes to complete. Fortunately, this task is easily parallelisable. Our framework exploits this fact: 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, which then distributes a job to a node. This node then measures its energy consumption. The energy measurement, along with the output of the job, is then returned to the client. Figure 2 shows our framework's layout with 2 nodes.

The nodes in this cluster are Raspberry Pi 2 Model B devices [4], each running the 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. The cluster comprises 18 Raspberry Pi nodes, an additional Raspberry Pi master node to distribute jobs to them, and six MAGEEC Energy Measurement Boards, each with its own Raspberry Pi to manage energy measurement start and stop commands.

Each Raspberry Pi node can measure itself using a MAGEEC Energy Measurement Board [2]. The MAGEEC Energy Measurement Board intercepts the power supply of the Raspberry Pi, measuring the voltage drop across a resistor inline to the power supply at a sustained sampling rate of 2 MHz. This allows us to make *direct* energy measurements, avoiding the unknown uncertainties of indirect measurements. Each MAGEEC Energy Measurement board can measure up to three targets. The MAGEEC Energy Measurement board is itself controlled by a separate Raspberry Pi Device which accepts requests to start and stop energy measurement from the three measurement targets over a network. Thus, each MAGEEC Energy Measurement board measures three raspberry Pis with a forth controlling the measurement board and interaction between the board and the other three devices.

We find this setup is capable of mitigating 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 we would otherwise be possible. As we discuss in Subsection 6.1 these MAGEEC energy measurement devices do not achieve a level of accuracy that we would deem acceptable for most investigations and, as such, have had to report proportional measurement increases/decreases (which are accurate). 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 is applied at the source-code line level. We apply these to a tagged representation of source-code; a representation specially created for GI research, first introduced by Langdon and Harman in 2010 [22] and later utilised in a variety of other GI work [23, 24, 34]. We refer to this format as the *Langdon format*.

At present tools only exist to transform C/C++ code into Langdon format. As the operators to be applied function at the source-code line level the code is formatted so that each statement is on its own separate line to avoid modifica-

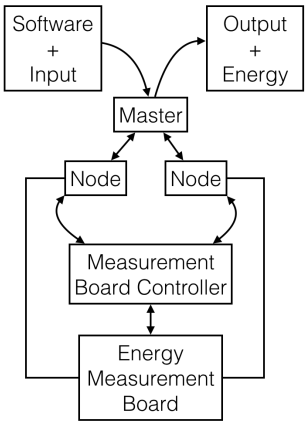


Figure 2: Diagram of the energy measurement cluster, showing two nodes measured by a single energy measurement board.

```
<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"
```

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

tions being applied to multiple statements. Opening and closing curly brackets are moved to their own line so that modifications line's containing a statement does not interfere with program scopes. In order to reduce errors we also ensure bracketless one-statement `FOR/WHILE/IF` bodies are refactored to be enclosed by curly brackets.

To translate this formatted C/C++ code to the Langdon format, each line is labelled with a unique identifier. These identifiers indicate whether a line is modifiable or not. Unmodifiable identifiers begin with `<{FILE}`. Opening and closing curly brackets, variable initialisations, and function declarations, are unmodifiable. Only the conditions, and the pre- and post-statements of a `FOR`, and the headers of `IF`, `WHILE`, and `FOR` statements can be modified. Figure 3 shows a snippet of source-code in the Langdon format.

Once converted to the Langdon format the source-code can be modified taking into account the aforementioned restrictions. It can then be expanded back to the original source-code by including the unmodifiable lines then expanding. For example, in Figure 3 `<LzFind_262>`, an unmodifiable line, references `<_LzFind_262>`. When converted back to source-code `<LzFind_262>` is added and expanded to produce `MatchFinder_ReadBlock(p);\n`.

Figure 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* operation; a modifiable identifier followed, without a space, by another, `<LINE1_ID><LINE2_ID>` is a *replace* operation and replaces the former with the latter; and two identifiers separated by `+`, `<LINE1_ID>+<LINE2_ID>`, is a `copy` operation which copies one line (the latter) to another area of the source code (above the former). A space separates multiple operations.

We replace the condition of a `IF`, `FOR`, and `WHILE` statement only with the condition of a matching statement, i.e., an `IF`'s conditional can only be replaced by another `IF`'s conditional, etc. A `FOR`'s pre-statement can only replace another `FOR`'s pre-statement, the post-statement another `FOR`'s post-statement. When the *delete* operator is applied to a conditional clause, it replaces the conditional with `false`. 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. For example, 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.

3.3 Assessing Program Behaviour

As we observed in the introduction, our search space of possible modifications is vast, too vast to analyse exhaustively. We choose, therefore, to uniformly sample it. We apply *delete*, *copy*, and *replace* operations to points, chosen

```
#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>
```

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

uniformly at random, in the source-code tagged as modifiable (let this number be n), then add individual modifications that compile to a *Modification Set* until the cardinality of this set is $2n$. This sampling method requires only $2.25n$ evaluations since our edit operators on the Langdon format achieve compilation rates of 79-81% [34].

Algorithm 1 The Filtering Step

Input: E, A set of edit-location pairs (e, l)
S, The target software
T, The set of testcases

$t \leftarrow$ uniformSelection(T)
$M_o \leftarrow \{\}$
for $0 \ldots N$ **do**
  startEnergyMeasure()
  $R_o \leftarrow$ run(S, t)
  $m \leftarrow$ endEnergyMeasure()
  $M_o \leftarrow M_o \cup \{m\}$
end for
$b \leftarrow$ lowerBound(M_o)
$C \leftarrow \{\}$
for $(e, l) \in E$ **do**
  $S_e \leftarrow$ applyMod(S, (e, l))
  startEnergyMeasure()
  $R_e \leftarrow$ run(S_e, t)
  $m \leftarrow$ endEnergyMeasure()

  if hasPassed(R_o, R_e, t) and $m <$ b **then**
    $C \leftarrow C \cup \{(e, l)\}$
  end if
end for
return C

Even when sampling, we cannot evaluate every variant against all the available testcases. Variants that are inert, produce software that breaks hard-constraints, or increase energy consumption are not of interest to us. In previous work, we observed that these variants make up the majority of any given local search space [11]. Therefore, we filter them out. Algorithm 1 presents our filtering algorithm. The algorithm evaluates members of the Modification Set, E which we may conceptualise as a set of edit (i.e. *delete*, *copy*, and *replace*), e, and location, l pairs. These are evaluated against a single, randomly chosen, testcase, t from the original program's test suite T. If the variant resulting from the modification S_e, passes the testcase and its energy consumption is less than the 95% confidence interval of the mean lower bound, b, for the original software running a predetermined number of times (N, 100 in this case), we add to the *Candidate Modification Set* C. After this filtering step C contains those modifications for which we can say, with statistical confidence, that 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. The output 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 then decompressed to its original state though the compressed file may be approximated. For 7zip, this means a less compressed `7z` file. Section 5 gives a precise description of the criteria used for the exact and approximate test oracles which determines whether the software variants pass or fail for a given input.

Algorithm 2 The Evaluation Step

Input: C, The set of candidate edit-location pairs (e, l)
S, The target software
T, The set of testcases

```
1:  D ← { }
2:  for t ∈ T do
3:    for 1 ... N do
4:      startEnergyMeasure()
5:      R_o ← run(S, t)
6:      m ← endEnergyMeasure()
7:      D.add(N/A,t,m, 0, 1)
8:    end for
9:    for (e, l) ∈ C do
10:     S_e ← applyMod(S, (e, l))
11:     for 1 → N do
12:       startEnergyMeasure()
13:       R_e ← run(S_e, t)
14:       m ← endEnergyMeasure()
15:       p ← hasPassed(t, R_e)
16:       a ← getAproxVal(R_o, R_e)
17:       D.addRecord(m, t, m, a, p )
18:     end for
19:   end for
20: end for
21: 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 ($N = 30$ in our investigation) with its energy, m, measured on each iteration. Then (again for each testcase) each candidate modification is applied to the unmodified software to produce the modified variant, S_e. The modified variant then processes the testcase with its energy, measured and its output, R_e recorded. We subsequently uses 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).

From this output data we determine the approximation value, a, and whether the software variant has passed the testcase, p. The formula for both the approximation value and what passing a testcase means for each application is defined in Section 5.

In the evaluation stage, we measure the '*Approximation Value*', which we obtain from method $getAproxVal$), our unified approach to recording values for both exact and approximate test oracles. In all cases, an approximation value of zero denotes satisfaction of the exact oracle; the output of the modified program corresponds exactly to the output of the original program. 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. This common terminology serves to combine very different measures of approximation. We define four domain-specific approximation criteria set out in Section 5.

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 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:

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 running executions 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, 28, 31]. 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. However, if such an error is present, in the specific case of work on energy improvement, this variance *can* be tolerated if it is consistent. We are only interested in the *relative* differences in energy consumption when assessing energy improvement, not the *absolute* measure of energy consumed.

This motivates RQ1b:

RQ1b: What is the variance when measuring across multiple devices?

To answer RQ1b, we carry out a process similar to that used to answer RQ1a, differing in its focus on one application, against which it runs one testcase across *all* the devices in the cluster. We use box plots to determine if there is variance across the devices.

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 acts as the test oracle, against which improved versions are compared [17, 38]. For a given candidate improved program, 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 software scenarios, exact replication of the behaviour of the original is either unnecessary or even undesirable. 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 [23]. 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 functional improvement is not possible, the genetically modified program may need only to *approximate* the behaviour of the original, sacrificing some degree of result quality for improvements in non-functional characteristics. For example, work on graphics shaders inherently involves a precision-speed trade off [37]. This would allow genetic improvement to reduce shader quality for improvements in execution time (or other non-functional properties). Often minor quality degradation is imperceptible or accepted to the end user, making such trade-offs highly desirable. Much of the work on energy improvement falls into this category [19, 29, 33].

This motivates RQ2, which investigates the trade-off achievable when 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

App	Description	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

Table 1: Number of modifications investigated for each application studied.

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 evolutionary algorithms typically combine lower-level building blocks of partially fit solutions in order to arrive at fitter combined solutions [14, 18, 32]. In RQ3, we therefore study the synergistic effects of combinations of individual modifications, reporting the frequency of different kinds of synergistic effects:

RQ3, Synergy: How frequently do synergistic 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% 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.

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

Limitations in the tool used to generate the Langdon format requires all software to be written in C/C++ with this code being open source and having a licence permitting it to be used for experimental purposes. As evaluation takes place on a Raspberry Pi device running the Raspbian OS, the software has to be compilable in this environment.

Due to inevitable overheads associated with sending energy measurement start and stop commands over a network [25], we choose 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.

Finally, 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.

Given these criteria, we settled on the applications in Table 1, which shows their lines of code (LOC), the number of lines modifiable in the Langdon format, and the number of modifications we generate and study for each application. Below, we summarize each application, focusing on the technical or implementation details relevant to our investigation.

5.2 7zip

7zip Version 9.38.1 [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, Lzma- 2Enc.c, LzmaDec.c and LzmaEnc.c) that accounted for over 99% of execution time when compressing and decompressing a 50MB 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 when converted to the Langdon format. For our experiments, we generated a Modification Set of size 5,000.

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 2 shows how this approximation value is calculated, where M is the compressed file produced by the modified software for a given testcase and O is the compressed file generated by the original software for the same test. Function S returns the size of the compressed file. The equation determines the ratio of the compression between the file produced by the modified software compared to that 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.

There are 40 testcases used 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 22.2MB to 64.4MB while the other three categories contain files with sizes ranging from 546KB to 12MB.

$$f(O, M) = \frac{S(M)}{S(O)} - 1 \tag{2}$$

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 [9] and has previously been used as a candidate for genetic improvement at the machine-code level by Schutle et al. [36]. 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. The 'simlarge' runs 256 image queries on a database of 34,973, the '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 will 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 be passed a non-null ranking must be returned by the application.

The calculation of the approximation value is shown in Equations 3 and 4. The output rankings for the original software O is compared against the modified software's output rankings M. Both O and M are a set of rankings (one for each query).

For each query,($q \in Q$) the ranks are compared using Kendall's τ Ranking statistic, R. For equal rankings Kendall's τ returns 1 and tends to -1 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. As can be seen from the Equation 3, when the Kendall's τ calculation tends to -1 the approximation value tends to infinity. To avoid this we modify the Kendall's τ calculation to have a minimum of -0.9999. This means, in practise, the approximation value ranges from 0 to 10,000. As both O and M contains many queries the approximation value is averaged by dividing the sum over the number of queries, N. If an image was ranked in the top 50 for the original output but not the modified output then is added to the end of the modified ranking.

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

$$f(O, M) = \frac{1}{N} \sum_{q}^{Q} K(O(q), M(q)) \tag{4}$$

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 and adds boxes to mark it for a human-readable output. It is part of Princeton's PARSEC Benchmark Suite

[9], version 3.0. Excluding libraries, Bodytrack consists of 23 C++ files that, in total, contain 3,020 lines of code. All this code has been used for modification. 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. 'simlarge' 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.

The approximation value is the average of the differences between the points produced by the modified software, M, and the points produced by the original software O (both contain $0..N$ points) for any given input (shown in Equation 5). The difference between two points, $diff$, 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.

$$f(O, M) = \frac{1}{N} \sum_{n=1}^{N} |diff(O(n), M(n))| \tag{5}$$

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 is 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. 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. 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-textfile functionality and again without. The latter is when the energy measurement is taken, the other run is used to evaluate the approximation value. For a testcase to pass a non-null output must be written to the text file.

As equation 6 shows, the approximation value for any given test is calculated by taking the number of lines returned by a POSIX `diff` on the output generated the modified software, M, in comparison with the output of the original software, O, for a given testcase. This is then divided by the total number of lines the original output. Therefore 0 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.

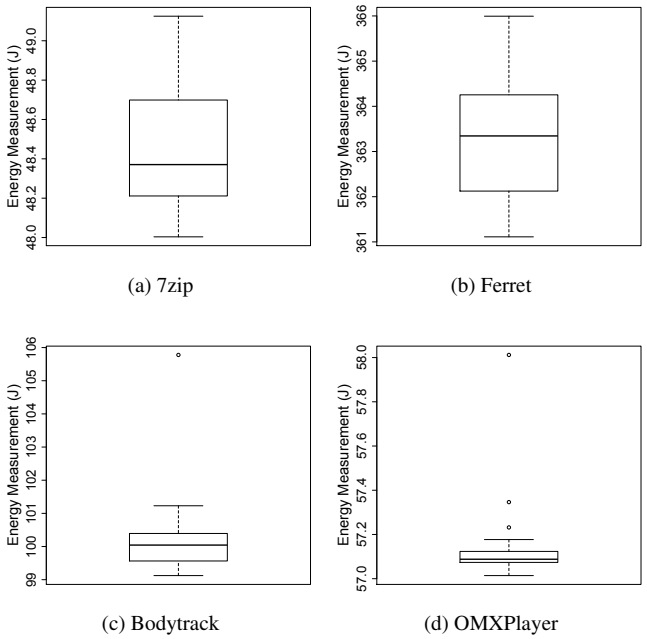
$$f(O, M) = \frac{N(\text{diff}(M, O))}{N(O)} \tag{6}$$

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 randomly sample 15% 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?*



(a) 7zip (b) Ferret

(c) Bodytrack (d) OMXPlayer

Figure 5: The variance in measurements that occurred when running each application (unmodified) 30 times on the same device on a randomly chosen testcase.

RQ1a is "*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 Figure 5. Within devices the coefficient of variances are 0.65%, 0.35%, 1.17%, and 0.31% for 7zip, Ferret, Bodytrack and OMXPlayer, respectively. Therefore, we can conclude that the precision of energy measurement in our framework is high.

RQ1b is "*What is the variance when measuring across multiple devices?*". To answer this we measure the energy consumption on each node running 7zip (chosen uniformly at random from the four applications studied) on a single, randomly chosen testcase (in this case, 'The Complete Works of William Shakespeare', a 5.6MB file). The box-plots in Figure 6 show the distribution of energy readings for each Raspberry Pi device. As can be seen, the measurements vary noticeably between devices. In answering this research question we observed that restarting a node in the cluster can result in different readings compared to those given before its restart. However, assuming no system reboots take place the results remain consistent (as was determined in answering RQ1a).

Therefore, we conclude that the MAGEEC energy measurement boards *lack accuracy but have good precision*. This inaccuracy, however, is only relevant if one wants to obtain energy reductions or increases in Joules. The proportional increase or decrease for one measurement compared to another is consistent across devices and thus any energy saving observed for one device will result in a constant observed saving on other 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?*

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 exact test oracles. The most striking finding is the frequency of modifications that reduce energy consumption (i.e., '+ve mods'); averaging only

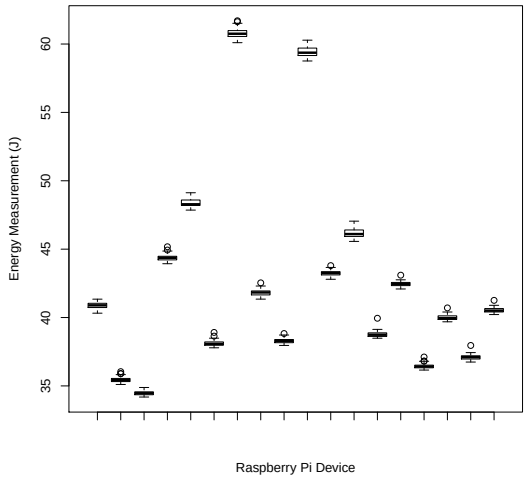


Figure 6: The variance in measuring the same program with the same input across different devices.

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%

Table 2: Each application with the number and percentage of modifications that reduced energy consumption according to an exact test oracle. The average and maximum magnitude of these modifications is also included.

0.09% across all cases. The impact of these is an average decrease of 1.25% across all applications with a maximum of 2.69%. 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 on 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 as the baseline); 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.

We found that approximation increased the frequency of effective modifications in the local search space to 1.36%; a 15-fold increase compared to those found when using the exact test oracle. This increase in frequency is 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% , an average reduction of 33.90% was achieved when using approximate test oracles.

Thus far we have assumed that any level of approximation is acceptable. In practice, however, this is not the case. A software variant that is more energy efficient than the original, but does not preserve *any* functionality of the program

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%

Table 3: Each application with the number and percentage of modifications that reduced energy consumption according to an approximate test oracle. The average and maximum magnitude of these modifications is also included.

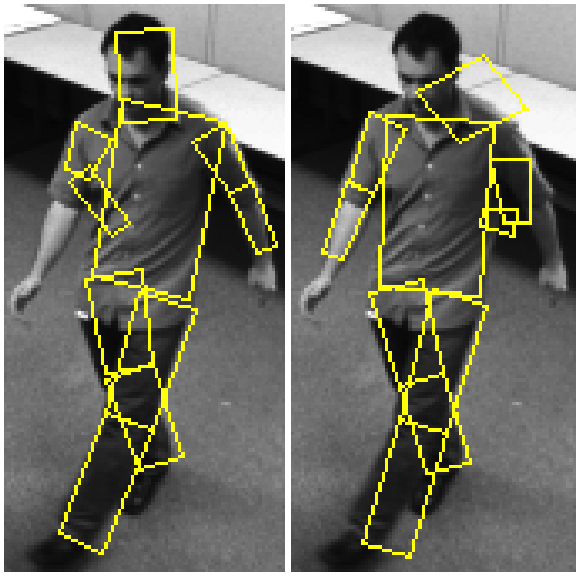


Figure 7: The output from two versions of the Bodytrack application. The image on the right is approximated but consumes 33.69% less energy to compute.

it was derived from, is unlikely to be considered as an improved version of the original software. Therefore, in Table 4 we provide Pareto fronts for each of the applications investigated which show the energy reduction vs. output quality trade-off. The approximation value in each case is calculated using the formulæ presented in Section 5.

In the case of 7zip the user has a choice of 4 Pareto Optimal solutions, ranging from a 5.08% energy reduction with an approximation value of 3.93×10^{-4} to a 48.30% reduction with an approximation value 0.741. The latter translates to the compressed file generated by the modified software being, on average, 74% larger than if compressed using the unmodified version of 7zip. This variant of 7zip is still performing non-lossy compression but less effectively.

For Ferret there are 6 Pareto Optimal solutions. This ranges from a 43.19% reduction in energy for an approximation value of 0.154, to a 79.88% reduction in energy for an Approximation Value of 6221.220. In the case of the former, the approximation value translates to a Kendall's τ of 0.73. The next solution on the Pareto front achieves a 60.79% 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' (-1).

There are 5 Pareto Optimal solutions in the Bodytrack case. These range from a 2.69% reduction where there is no approximation to a reduction of 33.68% with an approximation value of 0.452. Figure 7 shows the most energy-efficient solution's output compared to that produced by the original, unmodified software.

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% reduction with an approximation value of 0.003. This approximation value means, on average, 0.15% 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% reduction in energy consumption, when visually inspected, was identical to the original as expected.

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. [15] and Le Goues et al. [26] 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 [35], 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 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. Table 6 shows the average energy reduction per operator type. These tables highlight that *replace* and *delete* are the most frequent and, on average, produce the greatest energy savings.

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

(a) 7zip

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

(b) Ferret

Energy Reduction	Approximation Value
2.69%	0.000
19.26%	0.131
29.97%	0.170
29.13%	0.192
33.69%	0.452

(c) Bodytrack

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

(d) OMXPlayer

Table 4: Pareto fronts for the four subjects investigated showing trade-off between energy consumption and solution quality.

	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%

Table 5: Number of effective modifications using the *delete*, *copy* and *replace* search operators across all applications.

	delete	*copy*	*replace*
7zip	16.60%	0.00%	7.38%
Ferret	56.45%	5.60%	43.50%
Bodytrack	8.27%	3.60×10^{-3}%	8.31%
OMXPlayer	57.01%	26.95%	64.92%
Average	46.11%	8.14%	31.03%

Table 6: The average impact of effective *delete*, *copy*, and *replace* modifications, in terms of % of energy reduction, across all applications.

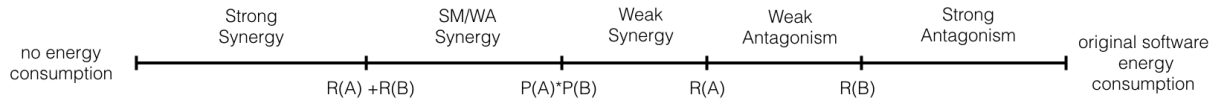


Figure 8: The categorisation system used in this investigation when studying the effects of two energy-saving software modifications (A and B) in respect to the energy consumed by the original software.

6.3 RQ3: Synergy

In order to investigate synergy we start by defining classifications for the various types of energy effects that may occur as a result of applying multiple software modifications.

We begin by introducing a definition of the threshold at which we say a synergistic effect has occurred. If we let $R(x)$ be the reduction in energy consumption of the original software due to the application of modification x and let $R(xy)$ be the reduction in energy by the application both x and y simultaneously, then we may view synergy as occurring at the point where $R(xy) > R(x) + R(y)$. We refer to this as the *Additive Synergy Threshold*

However, there is an alternative definition. If we let $P(x)$ be a proportional decrease of energy consumed by the original software when modification x is applied (e.g. a 50% reduction) and let $P(xy)$ be the proportion of energy consumed the software when both modifications x and y are applied simultaneously then we may view synergy occurring at the point where $P(xy) > P(x) * P(y)$. We refer to this as the *Multiplicative Synergy Threshold*.

As both these thresholds hold true for different views of what synergy is we require a classification system that takes into account these two conflicting definitions. Furthermore, we desire a classification system that provinces information as to whether antagonism has occurred and, ultimately, whether the pairing of two modifications is greater than either one applied exclusively.

All the following definitions assume $R(A) \geq R(B) > 0$ where A and B are modifications that may be applied to a target application.

Definition 1 *We say a pairing has* Strong Synergy *iff* $R(AB) > R(A)+R(B)$. *This is an observed synergistic effect regardless of the threshold observed as* $R(AB) > P(AB)$.

Definition 2 Weak Additive Synergy and/or Strong Multiplicative Synergy *occurs iff* $R(AB) \leq R(A) + R(B)$ *and* $P(AB) > P(A) * P(B)$. *This categorisation signifies that synergy may have taken place depending on the definition of synergy used.*

Definition 3 Weak Synergy *occurs iff* $P(AB) \leq P(A) * P(B)$ *and* $R(AB) \geq R(A)$. *In this categorisation a pairing is not synergistic in a traditional sense of the meaning (that is, the whole being greater than the sum of its parts) but shows that these modifications are an effective pairing. Any of the modifications individually is less effective than that of the pairing.*

Definition 4 Weak Antagonism *occurs iff* $R(AB) < R(A)$ *and* $R(AB) \geq R(B)$. *This categorisation suggests the pairing is not as effective as the most effective constituent member.*

Definition 5 Strong Antagonism *occurs iff* $R(AB) < R(B)$ *or tests fail. This categorisation suggests the pairing is not as effective as either of the constituent modifications.*

The different types of synergy for two energy-saving modifications A and B are shown in Figure 8 using the categorisations defined here.

This investigation involved selecting a subset (randomly selected 15%) of all available pairs of effective modifications (approximation permitted), evaluating them, then classifying them accordingly.

Table 7 shows the distribution of the synergy classifications. As can be seen, at 44.8% of all pairings, 'weak synergy' is the most common classification, followed by weak antagonism. 12.0% of all pairings were found to be be strongly synergistic though this figure is skewed by Bodytrack where 35.3% of all pairings were classified as having 'Strong Synergy'.

App	strong synergy	sm/wa synergy	weak synergy	weak antagonism	strong antagonism
7zip	0.9%	0.0%	60.4%	28.3%	10.4%
Ferret	9.2%	5.8%	43.0%	33.7%	8.2%
Bodytrack	35.3%	4.2%	35.9%	21.3%	3.3%
OMXPlayer	2.6%	9.2%	39.5%	43.4%	5.3%
Average	12.0%	4.8%	44.8%	31.7%	6.8%

Table 7: The percentage of effective modification pairings that are synergistic. 'sm/wa' stands for 'strong-multiplicative / weak-additive' synergy.

In total 38.5% of modification pairings produce 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. With this we would advocate techniques such as GAs with crossover methods suitable for testing the interaction of components.

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

The two modifications responsible for synergy in the case of 7zip altered two separated classes, `LzmaEnc.c` and `LzFind.c`, the latter utilised heavily in the former. 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` is solely applied it results in the skipping of a large portion of a frequently iterated for-loop, reducing execution time and thus reducing energy consumption. When both modifications are present the `LzmaEnc.c` maintains the for-loop skipping behaviour but, in addition, avoids execution of some costly branches.

This additional, synergistic, skipping behaviour is achieved by the `LzmaEnc.c` modification changing the control flow to more frequently arrive at an `IF` condition (`if(cur==lenEnd)`) leading to two separate branches. One branch is 'cheap', immediately calling a return statement. The other branch is 'expensive', executing a series of statements before calling a return. The `LzFind.c` influences the value of `lenEnd` within the `IF` condition which reduces the execution frequency of the more expensive branch by 15.7% Individually, the `LzmaEnc.c` modification achieves a 24.7% reduction in energy consumption. Likewise, in the case of the `LzFind.c` modification a 17.5% reduction in energy reduction is found. When both are combined a 43.4% reduction is achieved. The synergistic effect results in an 'additional' saving of 1.2%.

7 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 may mean that modifications which produce very small but positive changes are undetectable. While we were able to detect energy decreases as small as 0.009% there may modifications that produce even smaller changes that are simply undetectable with our framework. Though our investigations show that modifications which produce detectable, non-trivial, energy reductions are rare.

In this investigation we have been careful to ensure that any modification reported as being effective is. To achieve this aim our requirements for what constitutes as an 'effective modification' have been strict. For a modification to be classified as effective it must have produced a statistically significant decrease across 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 thus avoiding publication of results that may not be applicable to all those in the GI research community.

We chose to investigate the *copy*, *delete* and *replace* search operators because of their frequent use in state-of-the-art genetic improvement work [23, 24, 34]. There is a possibility that other search operators would work better in the context of energy consumption, however, our aim was to investigate the nature of the search space produced in modern GI research.

8 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. [39] were among the first to automatically search for modified versions of existing programs to reduce energy consumption, trading functionality for energy reduction. More recently, the past five years have witnessed an explosion of activity in this area.

Hoffman et al. dynamically tuned parameters to limit power spikes in server clusters [19]. Schulte et al. [36] introduced the Genetic Optimisation Algorithm (GOA) that was able to reduce the energy consumption of existing software systems from the PARSEC benchmark suite by an average of 20%. Manotas et al. [31] used a constrained exhaustive search for modifications to existing open source Java systems, reporting energy improvements of between 2% and 17%. Li et al. [29] demonstrated that by sacrificing some degree of usability, energy savings of up to 40% 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. Bruce et al. [11] showed that searching for specialised versions of a system (tailored to its downstream applications) facilitated energy improvements of up to 25%. Burles et al. [12] used automated search to find an improved version of Google Guava's `ImmutableMultimap` class that reduced its energy consumption by up to 24% and tuned the parameters in Google Guava's `CacheBuilder` to achieve a 9% energy saving [13].

Our work differentiates from the above approaches in that we investigate the characteristics of the search space for software improvement with respect to energy consumption.

9 Conclusions

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

In terms of impact, when using the exact test oracle an average decrease of 0.76% is observed though when using the approximate test oracle the average impact increases to 33.90%. 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 devices used in this investigation (the MAGEEC energy measurement boards) are precise but lack accuracy; highlighting an important and unreported systematic error in measurements given by these devices. However, any proportional energy saving obtained within one device translates into an equal proportional energy saving in another.

We also replicated previous findings (from non-energy-based evolutionary optimisation problems) that the *delete* and *replace* mutation operators are the most frequent and produce the greatest magnitude with effective *copy* modifications being rare and seldom responsible for significant energy reduction. We would therefore advise the removal of the *copy* from this commonly used set of operators.

We also investigated the effects that energy-efficient modifications produce when combined. We found that 61.5% of pairings were worthwhile; that is, the pairing's impact was greater than that of its most effective member with 12.0% being 'strongly synergistic'. The remaining 38.5% of modification pairs were antagonistic and therefore, we conclude there is no guarantee that two good modifications will always produce an energy-efficient software variant. Given this information we advise the use of non-greedy search-based techniques, such as evolutionary algorithms, to combine effective modifications in a controlled manner.

References

[1] 7zip. `http://www.7-zip.org`. [Online; accessed 25-November-2016].

[2] MAGEEC energy measurement board. `http://mageec.org/wiki/Power_Measurement_Board`. [Online; accessed 25-November-2016].

[3] Omxplayer. `https://github.com/popcornmix/omxplayer`. [Online; accessed 25-November-2016].

[4] Raspberry Pi. `https://www.raspberrypi.org`. [Online; accessed 25-November-2016].

[5] Raspbian. `http://www.raspbian.org`. [Online; accessed 25-November-2016].

[6] International Energy Agency. Key world energy statistics 2015, 2015.

[7] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE '14*, pages 588–598, 2014.

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[9] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[10] BP. Workbook of historical data 1985-2013 - Electricity Generation: Statistical review of world energy, 2014.

[11] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using Genetic Improvement. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference - GECCO '15*, pages 1327–1334, 2015.

[12] Nathan Burles, Edward Bowles, Alexander EI Brownlee, Zoltan A Kocsis, Jerry Swan, and Nadarajen Veerapen. Object-oriented Genetic Improvement for improved energy consumption in Google Guava. In *Search-Based Software Engineering*, pages 255–261, 2015.

[13] Nathan Burles, Edward Bowles, Bobby R. Bruce, and Komsan Srivisut. Specialising Guava's cache to reduce energy consumption. In *Search-Based Software Engineering*, pages 276–281, 2015.

[14] Kenneth A De Jong. On using Genetic Algorithms to search program spaces. In *Proceedings of the 2nd International Conference on Genetic Algorithms - ICGA '87*, pages 210–216, 1987.

[15] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A Genetic Programming approach to automated software repair. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference - GECCO '09*, pages 947–954, 2009.

[16] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[17] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: constructing the Pareto program surface using Genetic Programming to find better programs. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, pages 1–14, 2012.

[18] Mark Harman and Phil McMinn. A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis - ISTA '07*, pages 73–83, 2007.

[19] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212, 2011.

[20] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[21] Jonathan G Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3), 2008.

[22] William B. Langdon and Mark Harman. Evolving a CUDA kernel from an nVidia template. In *IEEE World Congress on Evolutionary Computation*, pages 1–8, 2010.

[23] William B. Langdon and Mark Harman. Optimising existing software with Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 99, 2015.

[24] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3D medical image registration CUDA software with genetic programming. In *Proceedings of the 2014 Genetic and Evolutionary Computation Conference - GECCO '14*, pages 951–958, 2014.

[25] William B. Langdon, Justyna Petke, and Bobby R. Bruce. Optimising quantisation noise in energy measurement. Technical Report RN/16/01, Department of Computer Science, University College London, 2016.

[26] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *Proceedings of the 2012 on Genetic and Evolutionary Computation Conference - GECCO '12*, pages 959–966, 2012.

[27] Ding Li and William GJ Halfond. Optimizing energy of http requests in android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 25–28, 2015.

[28] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for Android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA '13*, pages 78 – 89, 2013.

[29] Ding Li, Angelica Huyen Tran, and William GJ Halfond. Making web applications more energy efficient for OLED smartphones. In *Proceedings of the 36th International Conference on Software Engineering - ICSE '14*, pages 527–538, 2014.

[30] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 237–248, 2016.

[31] Irene Manotas, Lori Pollock, and James Clause. SEEDS: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering - ICSE '14*, pages 503–514, 2014.

[32] Melanie Mitchell, Stephanie Forrest, and John H Holland. The Royal Road for Genetic Algorithms: Fitness landscapes and GA performance. In *Proceedings of the first European Conference on Artificial Life*, pages 245–254, 1992.

[33] Vojtech Mrazek, Zdenek Vasicek, and Lukas Sekanina. Evolutionary approximation of software for embedded systems: Median function. In *Proceedings of the Companion Publication of the 2015 Genetic and Evolutionary Computation Conference*, pages 795–801, 2015.

[34] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In *Proceedings of the 17th European Conference on Genetic Programming - EuroGP '14*, 2014.

[35] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISTA '15*, pages 24–36, 2015.

[36] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. *ACM SIGARCH Computer Architecture News*, 42(1):639–652, 2014.

[37] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic Programming for shader simplification. *ACM Transactions on Graphics*, 30(6):152, 2011.

[38] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[39] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs. In *Proceedings of the 2009 Genetic and Evolutionary Computation Conference - GECCO '08*, pages 1775–1782, 2008.