Reducing Energy Consumption Using Genetic Improvement

Bobby R. Bruce
University College London
London
United Kingdom
r.bruce@cs.ucl.ac.uk

Justyna Petke
University College London
London
United Kingdom
j.petke@ucl.ac.uk

Mark Harman
University College London
London
United Kingdom
mark.harman@ucl.ac.uk

ABSTRACT

Genetic Improvement (GI) is an area of Search Based Software Engineering which seeks to improve software's non-functional properties by treating program code as if it were genetic material which is then evolved to produce more optimal solutions. Hitherto, the majority of focus has been on optimising program's execution time which, though important, is only one of many non-functional targets. The growth in mobile computing, cloud computing infrastructure, and ecological concerns are forcing developers to focus on the energy their software consumes. We report on investigations into using GI to automatically find more energy efficient versions of the MiniSAT Boolean satisfiability solver when specialising for three downstream applications. Our results find that GI can successfully be used to reduce energy consumption by up to 25%.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

Keywords

Search based software engineering, SBSE, genetic improvement, GI, optimisation, energy optimisation, energy efficiency, energy consumption, Boolean satisfiability

1. INTRODUCTION

Less than a decade ago the quality of software (outside of end-user design preferences) could broadly be described as the extent to which software met its specification while minimising the prevalence of bugs and usage of traditional computer resources such as CPU time and memory allocation. The growth in two new technologies, mobile computing devices and cloud services, has led to a new environment for software engineers where they must now consider the energy an application consumes; the quality of software is now measured in Joules, as well as bug counts, seconds, and megabytes. At present there are more smartphones in the world than personal computers [22], each containing a limited store of energy between charges that must be used efficiently. The energy required to run large server clusters has grown considerably in the last decade, estimated to be between 1.1% to 1.5% of global electricity consumption in 2010 [26], putting strain on energy suppliers and the budgets of those responsible for purchasing this energy [7]. The total ICT infrastructure generated 1.9% of global CO_2 emissions in 2011 [5] (larger than the entire United Kingdom estimated at 1.47% for the 2010-2014 period [42]) indicating that computer science has a role to play in mitigating climate change.

Thus we believe it important that software engineers find ways of programming computers with energy efficiency in mind to appease the demands from consumers for longer battery life, from companies to reduce their energy bills, and from society's desire to minimise humanity's impact on the environment.

One of the largest hurdles in producing energy-efficient software is the developer's disconnect between the source code they write and the energy that will be consumed from the compiled product they deliver [33]. Without a deep understanding of how a particular compiler works, along with an equally deep understanding of how much energy a given instruction will consume, the problem remains difficult for many developers. It has been found that metrics previously believed to guide developers to more energy efficient solutions are, in reality, poor at doing so [38]. Subtle changes, such as introducing inline methods [41], swapping API implementations [33], and constructing semantically equivalent (but structurally inequivalent) algorithms [8] have all been shown to influence energy consumption. However this influence is difficult to determine outside of the ad hoc and inefficient process of trial-and-error. Tools have been developed to guide users to energy-inefficient areas of their software [2, 11, 30, 19] though the developer retains responsibility for rectifying these inefficiencies.

We suggest that the most under explored method of decreasing software's energy consumption lies in automated processes. Such processes would allow developers to focus solely on meeting the specification requirements with worries about non-functional attributes like energy consumption left to an algorithm capable of refactoring software to a more optimal state.

Genetic Improvement (GI) [20, 25, 27, 28, 29, 36, 37, 45, 44] is a Search Based Software Engineering (SBSE) technique [21] which treats program code as if it were genetic material that can then be evolved to produce optimised solutions. GI has previously been found effective at optimis-

GECCO '15, July 11 - 16, 2015, Madrid, Spain

DOI: http://dx.doi.org/10.1145/2739480.2754752

ing software's execution time [20, 28, 37, 44] and similar genetic techniques have been shown effective at reducing energy consumption, albeit as a post-compilation process [40]. We therefore seek to investigate whether GI can be used at the source-code level to minimise energy consumption in a widely-used piece of software.

In 2014 Petke et al. [37] demonstrated that MiniSAT[1], a popular Boolean satisfiability solver (SAT Solver), can be optimised for execution time, producing a solver 17% faster than any human-written equivalent for the Combinatorial Interaction Testing (CIT) domain. SAT solvers are crucial components in many applications [4, Part 2], from AI planning [24] through to package management [43] and predicting crosstalk in integrated circuits [9]. Due to this inherent flexibility [32] and previous success in improving its non-functional attributes, we shall attempt to improve MiniSAT's energy consumption using GI techniques.

We set out to answer the following research questions:

RQ1 To what extent can MiniSAT's energy consumption be reduced using Genetic Improvement?

RQ2 Do different downstream MiniSAT applications require different optimisations?

RQ3 Does reduction in energy consumption correlate to reduction in execution time when GI is applied?

2. OPTIMISING ENERGY CONSUMPTION WITH GENETIC IMPROVEMENT

We build on an approach to GI first outlined by Langdon and Harman in 2013 [28] and later modified by Petke et al. in 2013 [36] and again in 2014 [37] for their MiniSAT experiments. We apply modifications to the fitness function, taking into account energy estimates made by the Intel Power Gadget (see Section 2.4) instead of the number of lines executed (a metric previously used to optimise for execution time). We also modify the selection and mutation procedure in an attempt to tackle the phenomenon of bloat. As in Petke et al.'s work [36, 37], all modification occurs in MiniSAT's `Solver.C` class which contains the main solving algorithm.

2.1 Program and Genotype Representation

Modification to the software is made at the source code level. The source code is converted into a customised template format, captured using BNF notation. The BNF notation allows lines to be tagged with discrete markers that indicate whether a line can be modified or not. Methods, opening and closing brackets, and initialisation lines are declared unmodifiable in order to reduce the percentage of solutions developed which are uncompilable. Figure 1 shows an example of the tagging format used in our experiments.

A genotype represents a list of modifications made to this format that is then translated to its phenotype by applying these modifications to the original source code. The three modifications permitted are to DELETE, REPLACE or COPY a line of source code. A DELETE operation removes a line, a REPLACE operation replaces one line with another and a COPY operation copies a line to another location. All genetic material used is contained within the original source code. This development is based on the observation that source code is redundant [14, 28] to the extent that genetic material necessary to produce an improved result is likely to be contained within the source code itself. Special cases exist for conditional statements so that predicate expressions can only be replaced or deleted with predicates found in the same type of conditional statement (the DELETE operation replaces a predicate expression with '0' to avoid compilation issues). For example, a while loop (e.g., x>5) can only be replaced with the condition of another while loop (e.g., y==2) and can never be replaced with a random line found elsewhere in the software (e.g., z=z*2;). Figure 2 shows how these modifications are represented inside a population of solutions.

[1]Available at http://minisat.se/MiniSat.html.

```
<Solver_235>      ::=    " if"<IF_Solver_235> " \n"
<IF_Solver_235>   ::=    "(order_heap.empty())"
<Solver_236>      ::=    "{\n"
<Solver_237>      ::=    "" <_Solver_237> "\n"
<_Solver_237>     ::=    "next = (-1);"
<Solver_238>      ::=    " break;\n"
<Solver_239>      ::=    "}\n"
```

Figure 1: A Solver.C snippet converted to our GI format. Lines starting with "<Solver" are unmodifiable.

```
 #DELETE line 205
<_Solver_205>

 #REPLACE if condition in line 154 with if
 #condition in line 307
<IF_Solver_154><IF_Solver_307>

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

#REMOVE line 56 and REPLACE line 78 with
#line 145
<_Solver_56> <_Solver_78><_Solver_145>
```

Figure 2: An example population of four solutions.

2.2 Fitness Function, Selection, Crossover and Mutation

The fitness of a candidate solution is determined by measuring the total energy consumed (see Section 2.4) across all tests selected from the training set (see Section 2.3) when using the original unmodified software divided by the energy consumed by the phenotype across the selected tests. Thus a fitness greater than 1 indicates a solution that consumes less energy while a fitness less than 1 indicates a solution that consumes more energy.

Each selected test case can either be passed or failed. A test is deemed to have passed when the modified version categorises a test as satisfiable or unsatisfiable with that categorisation equal to the categorisation produced by the original MiniSAT. We thereby use the original code as an oracle [3] to guide the GI to functionally correct solutions. When a test is found to have failed, the energy consumption

for that test case is not included in the fitness evaluation and, instead, an appropriate penalty is applied.

To be selected for the next generation, a solution must have a fitness of above 0.95, have passed an appropriate number of the selected test cases, and be in the top 50% of the population. Crossover is carried out by selecting one parent based on fitness and another chosen randomly from the selected individuals. Due to the simplicity of the genotype representation, crossover consists of appending one genotype to another; producing a new individual. Crossover is carried out until the population size, after selection, has doubled.

After crossover, mutations are applied to the selected genotypes. Prior investigations have shown GI frameworks such as this can lead to bloat [28, 37], resulting in effective solutions being encumbered with ineffective mutations. For this reason elitism has been implemented so that the top 5 solutions in each generation move forward to the next without mutation. The remaining selected individuals have a 50% chance of having a mutation applied. Mutations consist of adding a random DELETE, REPLACE, or COPY modification to the genotype. If the population after crossover has not met the preset population size, then single, random mutations are added as entirely new genotypes until the population size is met. The initial generation is seeded in this manner; a population consisting entirely of single, randomly chosen, modifications.

2.3 Training Set

For each MiniSAT downstream application a training and test set are constructed, representative of that application's use. Within each generation only a small subset of tests are selected from the training set to evaluate the solutions. A subset is chosen both to avoid over-fitting and to reduce execution time. The number of tests is dependent on the MiniSAT application: four for CIT (see Section 3.1) and five for Ensemble and AProVE (see Sections 3.2 and 3.3). To ensure this selection is representative of the entire training set, a binning system is introduced (originally presented by Langdon and Harman in 2013 [28]) which decomposes training set tests into bins based on complexity and type. A random test from each bin is selected in each generation. For our experiments we form bins based on execution time and satisfiability. Bins 1 and 3 contain satisfiable solutions while bins 2 and 4 contain unsatisfiable solutions. Bins 1 and 2 are examples of tests with small execution times while bins 3 and 4 are tests with larger execution times. If a 5th bin is present, then it contains both satisfiable and unsatisfiable solutions that have a larger execution time than those in bins 3 and 4. This test case selection process guarantees that fitness evaluation is always carried out against both satisfiable and unsatisfiable tests of varying difficulty.

2.4 Estimating Energy Consumption

We estimate the energy consumed in computing these tests using the Intel Power Gadget API for Mac OS X[2] which estimates the energy consumption of 2nd Generation and higher Intel Core processors. Given MiniSAT's singled-threaded, CPU-bound nature we believe this method of estimation is suitable for our requirements, however it should be noted that Intel Power Gadget estimations do not include energy consumed in main memory nor that consumed during I/O tasks.

Intel Power Gadget uses drivers and libraries to read the processor's special energy model-specific registers (MSRs) over a specified time period. These register readings are then used to calculate the total energy consumed.

We have built this API into a C++ application that takes a terminal command as input, estimates the CPU's energy usage during command execution, and returns the amount of energy consumed over this time in Joules. Running MiniSAT against a test case using this program gives us the energy consumption of MiniSAT for that test case.

When carrying out our experiments we were careful to avoid running processes that may have caused large variations in energy estimations. All unnecessary background processes were terminated while experiments were running. Running the original MiniSAT on a typical test case for 200 iterations, taking energy readings each time, we found a standard deviation of 5.23% in estimates; a variance within the limits we deem acceptable for our experiments.

3. EXPERIMENT SETUP

We modify the main solving algorithm in MiniSAT2-070721 (hereinafter simply referred to as "MiniSAT"). In each experiment the GI framework is run for 20 generations with a population of 100. Once complete each solution that has achieved a fitness greater than 1.05 is run against all tests within the training set to give an overall ranking of the best solutions based on the total energy consumed. This step has been included as it was found that many solutions with a high fitness only performed well on the tests selected for that generation. The step can be considered a method of determining the "true fitness" of the best solutions, as opposed to the fitness value given by the GI framework. The top solution is chosen from this "true fitness" list and declared the "champion" solution.

The champion solution is then run against the test set 20 times with the energy readings averaged to give a value which is then compared to the original MiniSAT's performance against that test set (run 20 times then averaged) to obtain an overall percentage improvement.

We run three experiments, each specialising MiniSAT for a different downstream application (see Sections 3.1, 3.2, and 3.3). In practice each experiment simply requires that we use a different training set (and test set for the final results).

To answer **RQ1** (*To what extent can MiniSAT's energy consumption be reduced using Genetic Improvement?*) we observe the energy improvements between the original, unmodified MiniSAT and the champion for each downstream application. As the champion solution and the original MiniSAT are run 20 times on the test set we are able to determine how statistically significant these results are.

RQ2 (*Do different downstream MiniSAT applications require different optimisations?*) is answered by running the champion solution for each downstream application against the test sets of the other applications. If a comparable energy reduction is observed when using other test sets then the modifications applied are general (in the sense they are improvements across all the downstream applications tested), however if the energy reductions are significantly smaller, crashes occur, or timeout events are triggered then the modifications must be specialised in some manner. We also anal-

[2]Available at software.intel.com/en-us/articles/intel-power-gadget/.

yse the modifications made to the software to produce the champion solutions in an attempt to determine whether any similarities can be found. Where solutions are found to be specialised, we investigate why modifications applied to one champion are less effective (or ineffective) when used on another MiniSAT downstream application.

In order to answer **RQ3** (*Does reduction in energy consumption correlate to reduction in execution time when GI is applied?*), we take each experiment champion and measure the total time required to compute all tests within their respective test set. These times are then compared to the execution time of the original, unmodified software when computing the test set to give a percentage improvement of execution time. This can then be compared to the energy percentage improvement. If the energy improvements are comparable to improvements in execution time we can add weight to the argument the two are related, if not we can claim this relationship is not valid in all cases. To obtain a deeper understanding of the time-energy relationship we also sample random solutions from each experiment (20 from each experiment, 60 in total) and measure their execution times and energy consumption estimates on their respective test sets. Analysing the data we are able to determine how correlated these readings are.

The following subsections describe the MiniSAT downstream applications we specialise for. We have carefully selected three applications areas we believe are suitably diverse in real-world or academic usage. For each we describe the purpose of the downstream application and give a description of the training and test set provided.

3.1 Specialising for CIT

Combinatorial Interaction Testing (CIT) is a black box test sampling technique used to test highly configurable software [34]. With highly configurable software, such as database management systems and architectures like software product lines, testing all configuration combinations is impossible though it remains important to ensure no combination of configuration variables exist that results in the software failing. CIT's role is to produce a test suite which sufficiently covers the configurations while minimising the execution time of such tests. CIT has been successfully translated and run as a Boolean satisfiability problem [1, 34], though running these SAT problems is a computationally intensive task. In 2014 Petke et al. [37] optimised MiniSAT to reduce the computation time for this domain, we aim to reduce the energy consumption.

The CIT training set contains 58 tests, 23 of which are satisfiable, spread evenly over 4 bins. The mean execution time for bins 1 and 2 is 3.33s, the mean execution time for bins 3 and 4 is 10.68s. The test set contains 20 tests, 11 of which are satisfiable with an average execution time of 13.07s.

3.2 Specialising for Ensemble Computation

Ensemble Computation is the study of an NP-complete variant of the Boolean circuit problem where one must find the smallest circuit that satisfies a set of Boolean functions simultaneously [23]. This problem can be translated into a satisfiability problem, which MiniSAT can then seek to solve.

The Ensemble training set contains 25 tests, 12 of which are satisfiable, spread evenly over 5 bins. The mean execution time for bins 1 and 2 is 4.21s, the mean execution time for bins 3 and 4 is 9.89s. The average execution time for Bin 5 is 28.87s. The test set contains 14 tests contain, 4 of which are satisfiable with an average execution time of 14.23s.

3.3 Specialising for AProVE

AProVE, Automated Program Verification Environment[3] [16, 17, 18], is a system for the generation of automated termination proofs of term rewrite systems. AProVE uses a Boolean satisfiability solver to determine which paths can or cannot be reached. Proving termination is a much discussed area in computer science [6, 12, 15, 31] with SAT solvers frequently used to aid analysis [10, 13, 39]. In this example the SAT solver is a component in a much larger application. This distinguishes it from the two other applications we optimise (where SAT solvers take a central role in problem solving). It serves as a reminder that GI need not be applied to an entire application but individual components within an application. Any component, when receiving the correct training data, can be optimised. Improving the parts that, in-turn, improve the whole.

The AProVE training set contains 24 tests, 13 of which are satisfiable, spread evenly over 5 bins. The mean execution time for bins 1 and 2 is 6.27s while the mean execution time for bins 3 and 4 is 19.04. The average execution time for Bin 5 is 25.33s. The test set contains 11 tests, 5 of which are satisfiable, with an average execution time of 17.83s.

4. RESULTS AND DISCUSSION

In this section we report the results obtained from carrying out the experiments described in Section 3. All three experimental runs, each optimising a different MiniSAT downstream application, completed successfully.

4.1 RQ1: Energy Reduction

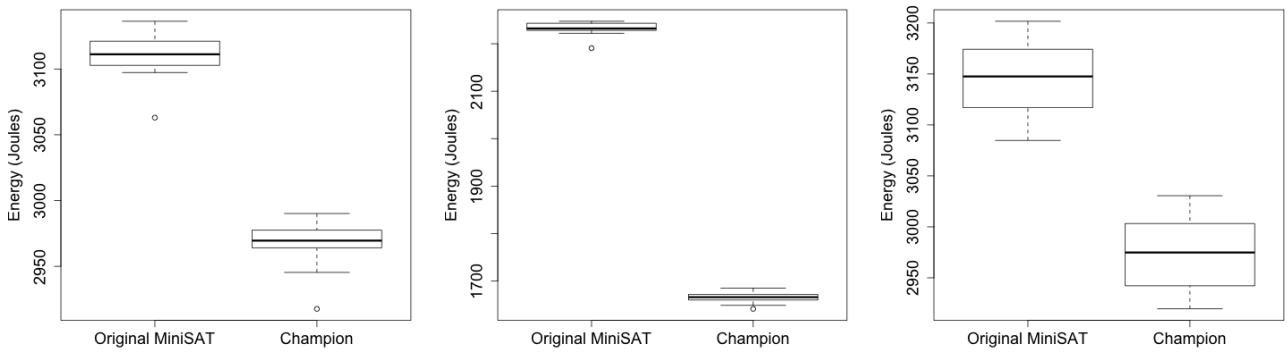
Application	Original(J)	Champ(J)	Reduction(%)
CIT	3111	2969	4.58
Ensemble	2232	1665	25.39
AProVE	3145	2973	5.44

Table 1: The original MiniSAT's total Energy consumption across all test-set tests compared to the Champion solutions' energy consumption.

All three experiments produced champion solutions which out-performed the original MiniSAT, in terms of energy efficiency, for their respective test sets. Table 1 shows the improvements. The CIT and AProVE champions achieve modest energy consumption reductions of approximately 5% while the Ensemble champion achieves 25.39%. Further analysis of the results has shown that these energy estimations are statistically significant ($p <= 0.01$) using the Wilcoxon signed rank test (we found CIT, AProVE, and Ensemble to have the equal p-values of 3.716×10^{-12}).

We carried out the Vargha-Delaney-A statistic on all three experiments and found each to have a score of 1. This score shows that the energy efficiency of the champion solutions are entirely superior to the original MiniSAT for their respective application domains. Figure 3 shows box-plots that visually demonstrate this significant effect size.

[3]Available at http://aprove.informatik.rwth-aachen.de/



(a) CIT (b) Ensemble (c) AProVE

Figure 3: Boxplots of the Champion solutions' Energy Consumption compared to that of the original MiniSAT

When taking into account that MiniSAT is a relatively small program and that it is already considered to be quite efficient (at least in terms of execution time), it is encouraging that a reduction in energy consumption of 25% has been achieved.

4.2 RQ2: Specialisation differences

-	On CIT	On Ensemble	On AProVE
CIT	-	X	X
Ensemble	X	-	X
AProVE	3.56%	3.86%	-

Table 2: The best solutions' energy consumption when computing other test-sets. An X indicates a timeout event when running the test set.

When the champion solution for each application was run against the test sets for the other two (see Table 2) we found that both the CIT and Ensemble champions timeout when attempting to run on the other two test sets (this timeout is set at 5 minutes per test case, no test set used here exceeds 90 seconds when run on the original MiniSAT). This indicates that these champions are "specialised" in such a way that they cannot be generalised as optimisations for all SAT problem sets. The AProVE champion functions correctly on the other test sets but does not achieve the same performance improvement as seen when run in AProVE domain. It appears that this may be a general improvement to MiniSAT unlike the other two solutions.

The AProVE champion solution, already shown to be a general MiniSAT improvement, is found to be the removal of an assert statement. Removing assert statements has previously been shown to produce good results for execution time when optimising MiniSAT [37]. Our experiments show that the same is true for optimising energy consumption. It is also easy to understand why such a modification does not result in specialisation as it will not produce a version of MiniSAT that is functionally different to the original, unmodified version.

The more interesting results come from the specialisation cases (CIT and Ensemble). In the CIT solution we find a mutation that results in a `if` statement being disabled (the predicate replaced with a zero through a DELETE operation) in MiniSAT's `pickBranch` function. The statement is used for picking a random variable for assignment and is called 2% of the time. The condition within the `if` statement itself involves running a random number generator which may be an unnecessary cost for such an under-used piece of code. It is currently unknown why this modification results in the solutions performing so poorly on the AProVE and Ensemble test sets. It is perhaps the case that this rarely entered `if` statement does result in significant impacts on performance for Ensemble and AProVE, to the extent that it is of benefit for them, but not for CIT.

For the Ensemble application, the application with the largest reduction in energy, we found the specialisation made a single modification to a `switch` statement. A modification equivalent in outcome to changing MiniSAT's polarity mode from `polarity_false` to `polarity_true`. This causes the solver to try an assignment of True instead of False to each variable that has been picked for branching. It seems that this polarity mode is of benefit in Ensemble Computation , however, as previously stated, why these changes produce measurable gains in performance is not fully understood.

One of the more unexpected, but nonetheless interesting, observations is that the champion in each experiment has always been a genotype containing only one modification. We reject the idea that single modifications are truly optimal. Not only does this run counter to optimal solutions found in similar experiments [28, 37], the champion solution found within the AProVE experiment is a general optimisation which could easily be crossed-over with the champions found for CIT and Ensemble to produce better results for each. Though mutation, and to a lesser extent crossover, always carries a high risk of producing poor solutions, it may be the case that our policy of elitism and employing only a 50% mutation rate to reduce this risk, make it more difficult to produce longer, fitter genotypes. Further research is needed to investigate these possibilities. However we believe that the results reported here are encouraging and thereby justify further study.

4.3 RQ3: Energy-Time Relationship

Table 3 shows the reduction in execution time for each specialisation. When compared to the energy reduction re-

Application	Unmodified(s)	Champion(s)	Reduction
CIT	268	261	2.58%
Ensemble	219	162	25.89%
AProVE	280	261	6.69%

Table 3: The original MiniSAT's execution time across all test-set tests compared to the champion solutions' execution times.

sults in Table 1 it can be seen that the values appear to correlate to the execution time.

To investigate the correlation further we sampled 20 functionally correct solutions from each experiment, and ran each solution against the entire test set with both energy estimated and time measured for each test. Figure 4 shows the data produced from this analysis with each test case's energy and time costs plotted. Visually the relationship between energy and time is stark, Table 4 gives the Pearson Correlation Coefficient for each experiment showing each to have a very strong energy-time correlation.

Application	N	Correlation	p
CIT	1160	0.988	$< 2.2 \times 10^{-16}$
Ensemble	457	0.995	$< 2.2 \times 10^{-16}$
AProVE	481	0.989	$< 2.2 \times 10^{-16}$

Table 4: The number of Data-Points, the Pearson Correlation Coefficient and p-value for the Energy-Time relationship in each experiment.

These findings show that for CPU-bound processes, such as MiniSAT, optimising execution time exclusively has the side effect of producing more energy-efficient solutions (and vice-versa). This provides further incentive to investigate using execution time as a metric for reducing energy consumption for CPU-bound, single-threaded applications. If energy consumption and execution time can be bundled into a single metric, developers may find it easier to reduce energy as methods to reduce program execution time are already well understood.

Although this result is somewhat expected and unsurprising, the effect has not been demonstrated empirically in the context of SBSE. We are therefore pleased this work may aid future research in providing evidence to their claims about Energy-Time relationships in CPU-bound, single-threaded applications.

5. THREATS TO VALIDITY

It is worthwhile mentioning there exists some threats to validity for the results produced and the conclusions drawn within this investigation. The first is the relatively small area of code optimised (478 lines). Working on such a small example is essentially reducing the search-space of the application, making it easier to navigate and optimise for. Techniques have been introduced for GI to optimise larger applications [28] but they have only been demonstrated for execution time optimisation. It is currently unknown how well these techniques would translate over to optimising for Energy Consumption.

The Intel Power Gadget has been chosen as a power estimation API for this investigation as it produced estimation with low variance, is easy to implement, and is supplied by a reputable hardware manufacturer though questions still remain over how accurate this API is. The true relationship between this estimation and the true energy consumption is unknown. Intel provide no formal documentation on this API, nor has there been any research into its validity meaning this research has been undertaken on the assumption the Intel Power Gadget functions within an acceptable degree of accuracy. Furthermore, the gadget is limited to CPU activity exclusively, therefore results presented here ignore any potential energy increases occurring elsewhere and limits the findings of this paper to CPU-bound applications.

6. FUTURE WORK

Our results show that optimisation is possible though there are still questions that must be answered to allow GI for energy optimisation to move from academia to real-world developers.

The first and foremost of these questions is how energy can be measured to capture usage outside of the CPU. This is of particular importance for mobile applications which are more likely to use components understood to consume relatively large quantities of energy such as GPS or WiFi [35]. Though there has been work in this area [19, 30, 35, 46] it is unknown how effectively these can be integrated into a the GI framework and therefore further research is required.

We highlighted in Section 5 that using GI to improve energy efficiency has yet to be applied to larger applications. Thus we believe future investigations should focus on optimising larger applications for energy consumption in order to better understand how the techniques presented can be scaled.

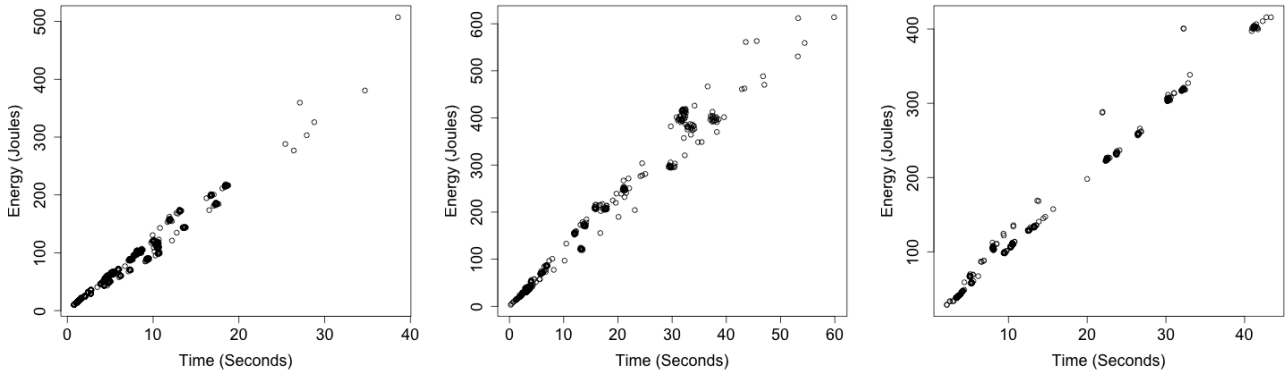
We also believe more work into Genetic Improvement is required. The genotypes produced in our experiments are smaller than we believe to be truly optimal. New techniques require development and testing to ensure the best results are delivered to the user.

7. CONCLUSION

We evolved MiniSAT, a popular Boolean satisfiability solver, to reduce its energy consumption on three applications. We found that energy efficiency can be improved by as much as 25%, though this varies greatly depending on the downstream application being optimised. We also discovered GI is able to find solutions to reduce energy consumption that would be difficult for human developers to find.

Two of the three champion solutions were found to be specialised to their respective downstream applications in a manner that they were no longer applicable to other MiniSAT downstream applications. This finding adds to the arguments presented in previous research that GI can be used to specialise solutions for specific environments [27, 37].

Our further investigations into the results produced in our experiments found that the energy savings corresponded to decreases in execution time most likely due to the CPU-bound nature of the application optimised. The very strong correlations found provides evidence that for applications with similar characteristics to MiniSAT (that is CPU bound, singled threaded, and with limited I/O activity) execution time and energy consumption may be considered interchangeable metrics.



(a) CIT (b) AProVE (c) Ensemble

Figure 4: Scatterplot of the Energy-Time relationship within the three experiments

8. REFERENCES

[1] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue. Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 112–126. Springer, 2010.

[2] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 588–598, New York, New York, USA, Nov. 2014. ACM Press.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 2015.

[4] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[5] Boston Consulting Group. GeSI SMARTer2020: The role of ICT in driving a sustainable future. http://gesi.org/SMARTer2020, 2012. [Online; accessed 10-January-2015].

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination analysis of integer linear loops. In *CONCUR 2005–Concurrency Theory*, pages 488–502. Springer, 2005.

[7] D. J. Brown and C. Reams. Toward energy-efficient computing. *Communications of the ACM*, 53(3):50–58, 2010.

[8] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour. Choosing the" best" sorting algorithm for optimal energy consumption. *ICSOFT*, 2009.

[9] P. Chen and K. Keutzer. Towards true crosstalk noise analysis. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 132–138. IEEE Press, 1999.

[10] M. Codish, I. Gonopolskiy, A. M. Ben-Amram, C. Fuhs, and J. Giesl. SAT-based termination analysis using monotonicity constraints over the integers. *Theory and Practice of Logic Programming*, 11(4-5):503–520, 2011.

[11] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194, 2010.

[12] N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):117–156, 2001.

[13] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. *SAT solving for termination analysis with polynomial interpretations*. Springer, 2007.

[14] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*, pages 147–156, New York, New York, USA, Nov. 2010. ACM Press.

[15] J. Giesl. Termination analysis for functional programs using term orderings. In *Static Analysis*, pages 154–171. Springer, 1995.

[16] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, et al. Proving termination of programs automatically with AProVE. In *IJCAR*, volume 14, 2014.

[17] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Automated Reasoning*, pages 281–286. Springer, 2006.

[18] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *Rewriting Techniques and Applications*, pages 210–220. Springer, 2004.

[19] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 92–101. IEEE, May 2013.

[20] 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 *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, pages 1–14, New York, New York, USA, Sept. 2012. ACM.

[21] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer, 2012.

[22] J. Heggestuen. Business Insider: One In Every 5 People IN The World Own A Smartphone, One in Every 17 Own A Tablet. http://www.businessinsider.com/smartphone-and-tablet-penetration-2013-10, 2013. [Online; accessed 9-January-2015].

[23] M. Järvisalo, P. Kaski, M. Koivisto, and J. H. Korhonen. Finding efficient circuits for ensemble computation. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 369–382. Springer, 2012.

[24] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92)*, pages 359–363, Vienna, Austria, 1992.

[25] Z. Kocsis, G. Neumann, J. Swan, M. Epitropakis, A. E. Brownlee, S. O. Haraldsson, and E. Bowles. Repairing and optimizing Hadoop hashCode implementations. *Search-Based Software Engineering*, pages 259–264, 2014.

[26] J. Koomey. Growth in data center electricity use from 2005 to 2010, Aug. 2011.

[27] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, July 2010.

[28] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 2013.

[29] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[30] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, pages 78 – 89, New York, New York, USA, July 2013. ACM Press.

[31] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic systems. In *Logic Programming: Proceedings of the Fourteenth International Conference on Logic Programming*, page 63. MIT Press, 1997.

[32] S. Malik and L. Zhang. Boolean satisfiability: from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[33] I. Manotas, L. Pollock, and J. Clause. SEEDS: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 503–514, New York, New York, USA, May 2014. ACM Press.

[34] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[35] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*, pages 29–42, New York, New York, USA, Apr. 2012. ACM Press.

[36] J. Petke, W. B. Langdon, and M. Harman. Applying genetic improvement to MiniSAT. In *Search Based Software Engineering*, pages 257–262. Springer, 2013.

[37] J. Petke, W. B. Langdon, M. Harman, and W. Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In M. Nicolau, K. Krawiec, and M. Heywood, editors, *Proceedings of the 17th European Conference on Genetic Programming (EuroGP)*, Granada, Spain, 2014.

[38] C. Sahin, L. Pollock, and J. Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 36. ACM, 2014.

[39] P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In *Frontiers of Combining Systems*, pages 267–282. Springer, 2007.

[40] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14*, pages 639–652, 2014.

[41] W. G. P. Silva, L. Brisolara, U. B. Corrêa, and L. Carro. Evaluation of the impact of code refactoring on embedded software efficiency. In *Proceedings of the 1st Workshop de Sistemas Embarcados*, pages 145–150, 2010.

[42] The World Bank. http://data.worldbank.org/indicator/EN.ATM.CO2E.KT/countries. [Online; accessed 10-January-2015].

[43] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 178–188. IEEE, 2007.

[44] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, Aug. 2011.

[45] D. R. White, J. Clark, J. Jacob, and S. M. Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1775–1782. ACM, 2008.

[46] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES/ISSS '10*, page 105, New York, New York, USA, Oct. 2010. ACM Press.