

Searching for Resource-Efficient Programs: Low-Power Pseudorandom Number Generators

David R. White
Dept. of Computer Science,
University of York, UK
drw@cs.york.ac.uk

John Clark
Dept. of Computer Science,
University of York, UK
jac@cs.york.ac.uk

Jeremy Jacob
Dept. of Computer Science,
University of York, UK
jeremy@cs.york.ac.uk

Simon Poulding
Dept. of Computer Science,
University of York, UK
smp@cs.york.ac.uk

ABSTRACT

Non-functional properties of software, such as power consumption and memory usage, are important factors in designing software for resource-constrained platforms. This is an area where Search-Based Software Engineering has yet to be applied, and this paper investigates the potential of using Genetic Programming and Multi-Objective Optimisation as key tools in satisfying non-functional requirements. We outline the benefits of such an approach and give an example application of evolving pseudorandom number generators and performing power-functionality trade-offs.

Categories and Subject Descriptors

D.1.2 [Software Engineering]: Automatic Programming

General Terms

Design

Keywords

Search Based Software Engineering, Genetic Programming, Multi-Objective Optimisation, Non-Functional Requirements, Automatic Programming

1. INTRODUCTION

Search-Based Software Engineering (SBSE) [7] has yet to tackle the hardware-software interface, and in particular it has yet to be applied to the problem of choosing between potential trade-offs during implementation. This is becoming an increasingly important part of a software engineer's task, as embedded systems (where this interface is most exposed) constitute the vast majority of computing platforms [18] and the continued growth of embedded systems

shows no sign of slowing. As their adoption becomes more and more widespread, the requirements for emerging low-cost and low-resource platforms are including an increasing amount of non-functional constraints and objectives. The resources available on such systems approaches a zero level, for example the power supply to passive Radio-Frequency Identification (RFID) technology.

With multiple and conflicting objectives to satisfy and a diverse and volatile target platform specification, the software engineer is confronted with a very difficult task [4]. For example, consider a situation where a programmer is told they are to provide some given functionality in a memory-efficient way, whilst also restricting the execution time and the power consumed. A designer must consider the impact of each decision in terms of these non-functional properties. This is an extremely challenging problem, because the space of possible solutions may be very large and the relationship between non-functional properties unclear and in some sense unpredictable with the knowledge in hand. Such a situation is a suitable target for the application of SBSE.

Genetic Programming (GP) [14] is a program search technique that has been applied successfully in the past to a diverse set of problems. It is a population-based search method that represents an individual program as a symbolic expression tree. New solutions are generated from two selected parents by exchanging chosen subtrees. Note that other program search methods such as Grammatical Evolution [19] could also be used.

As with other search methods, GP can be used in conjunction with Multi-Objective Optimisation (MOO) techniques [8]. MOO aims to find solutions that satisfy more than one objective, so that an individual's ability to solve a problem is assessed by a set of objective functions $f_1 \dots f_n$. Pareto-based MOO methods return a set of solutions in a single run, attempting to approximate an underlying *Pareto front*, where each solution achieves different balances between conflicting objectives (provided such conflicts exist). Each such solution x within the Pareto front is said to be *Pareto weakly non-dominated*, such that:

$$\neg \exists y \cdot f_i(y) < f_i(x) \text{ for all } i = 1 \dots n \quad (1)$$

There is no alternative solution y in the Pareto front that can improve on x in all of the n possible objectives, assuming the aim is to minimise the value of each objective function.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-131-6/08/07 ...\$5.00.

Therefore, Pareto-based methods can be used in program search to find a set of programs that make different trade-offs.

Hence a combination of MOO and GP can be used to explore solution and objective spaces such as those defined by the problem of low-resource embedded systems programming, both to generate solutions and to provide insight into the trade-offs that can be made between different objectives. Specific uses for such an approach are:

- Satisfying exact constraints specified as a requirement.
- Finding the extent to which objectives may be balanced: is the number of distinct individuals in the Pareto front large or small?
- Identifying how many different distinct values of each objective are represented within the Pareto Front, i.e. what level of granularity of trade-off exists?
- Empirically quantifying the relationships between multiple objectives: just how many joules of energy can be saved through increasing by 100ms the available processing time to the task?
- Finding a set of programs offering multiple different trade-offs. A solution may be chosen from this set depending on future requirements, whether statically at design-time or dynamically at run-time. For example, a new target platform may have fewer resources, or a different mode of operation may reduce the amount of available processing time.
- The discovery of a single source-scalable program [20] that will provide an improved quality of output when given an increased amount of resources.

This paper focuses on the first three uses, and illustrates them by constructing a technique to evolve random number generators that balance power consumption with functionality.

2. CASE STUDY

2.1 Overview

To demonstrate the principle of such an approach, pseudorandom number generators (PRNGs) were generated using GP, as an example of how power consumption can be taken into account when evolving software.

PRNGs are important program components that generate a stream of pseudorandom numbers, where the precise definition of “random” is dependent on the way in which the output will be used [13]. PRNGs are often found in embedded systems. For example, they may be used for key generation in cryptographic applications or within communication protocols for collision resolution. Typically, PRNGs are small code fragments that are used frequently within a system. Thus, random number generators with specific non-functional properties are a useful tool.

PRNGs have been produced using both GP [15, 12, 11, 16] and other bio-inspired techniques (for example, [21]). A complete review is not given here, rather we are most concerned with applying GP to find programs that output pseudorandom numbers. Where heuristic search techniques have been applied in the past, the key difference between

alternative methods has been the fitness function selected to establish how good a candidate PRNG is. Knuth [13] gives an extensive review of tests to measure the “randomness” of a sequence, all of which are potential candidates for a fitness measure.

2.2 Producing PRNGs with GP

Koza [15] demonstrated the ability of GP to evolve pseudorandom number generators, outputting a stream of binary digits when given as input a sequence $1 \dots 2^{14}$. He used an entropy-based measure that summarised the measurement of the distribution of possible subsequences. By this measure, for a sequence of N integers where each integer can take k different values, the desired probability of each possible subsequence occurring is $\frac{1}{k^N}$. Koza’s results successfully produced individual programs that provide sequences with high entropy, and perform well when measured against commercial randomisers under two statistical methods from Knuth [13]. Koza notes that in some sense the distributions are too perfect, in that the divergence from an ideal distribution is so small that the output could be considered unlikely to be from a truly “random” source.

Jannink [12] used GP to predict the outputs of existing commercial generators, as a measure of their quality. He then took a similar approach to creating new randomisers, by using coevolution to competitively evolve generators and predictors, thus using competition to measure fitness in place of statistical measures of randomness. The function set provided to GP was similar to that used in this paper, with the addition of memory reading and writing operations. Evaluating the success of the evolved generators against standard battery tests was not included in the paper, and therefore it is difficult to make comparisons between the effectiveness of this approach and those that define fitness using statistical measures.

Hernández et al.[11] also applied GP to PRNG creation. Part of the stated aim of their work was to consider not only the functionality of PRNGs but also the efficiency of the evolved solution, and hence this work is closely related to the aims of this paper in satisfying non-functional requirements. This work was continued and expanded upon by Lamenca-Martinez et al.[16]. Much of the experimental work reported here is based on this later work. In particular, the fitness measure and choice of function set are taken from these papers. The fitness function used measures the nonlinearity of a PRNG’s output, as an alternative to statistical measures of randomness. Further details of the fitness function are given in Section 3.

In order to produce efficient PRNGs, [11] and [16] restricted the function set to contain operations that could be executed quickly, and could be described as producing a “minimalist’s PRNG”. Direct measurements of efficiency were not made, and both papers comment on whether the inclusion or exclusion of a MULT (multiply) function would be appropriate due to its relative inefficiency. This paper extends their work by explicitly examining one non-functional property of each solution, power consumption. It also provides a method of comparing the impact of the MULT function on efficiency-functionality trade-offs for a specific target architecture.

Problem Description	Find $r(a_0 \dots a_7)$ where r minimises the χ^2 fitness metric defined by the strict avalanche criterion.
Function set	MULT, AND, SUM, NOT, OR, XOR, Logical Shift Left (LSL), LSR, Circular Shift Left (CSL), CSR
Terminal set	$a_0 \dots a_7$ and Integer Ephemeral Random Constants (ERCs)
Population Size	150
Generations	250
Crossover Probability	0.8
Reproduction Probability	0.2
Mutation Probability	0.0
SPEA2 Archive Size	100
Selection method	Tournament Selection, size 7

Table 1: GP Parameter Settings

3. EXPERIMENTAL METHOD

3.1 GP Parameters

Parameters for the GP search are given in Table 1. The table shows the major parameters of the search, which was implemented using the ECJ 16 Toolkit [17]. The settings for parameters not listed here are given by the parameter files `koza.params`, `simple.params` and `ec.params` supplied with the toolkit. All of these parameters were taken from Lamenca-Martinez et al. [16], with the exception of tournament size (since none was given) and the parameters for the multi-objective Strength Pareto Evolutionary Algorithm 2 (SPEA2), since SPEA2 was not used in that paper. The ECJ default was selected for tournament size. The aims of these experiments were to demonstrate the validity and potential of the multi-objective approach to functional trade-offs in general, and so no parameter tuning was attempted.

The fitness evaluation was performed in C, by converting the GP tree to a C expression and enclosing it within a function, and then compiled. As some of the members of the function set are not available as native C functions (CSL, CSR), they were implemented as functions within the test harness code and the output of an individual from ECJ was altered to replace the relevant infix expressions with function calls.

3.2 Fitness Measurement

The fitness of an individual is determined by its ability to satisfy two objectives: reducing power consumption and optimising functionality or “randomness”. In fact, its fitness is also dependent on the other individuals within the population due to the fitness sharing used by SPEA2, which incorporates the concept of pareto dominance (see Section 3.3 for further details). Within this paper, “performance” will be used to refer to the functional objective of improving the quality of the PRNG.

3.2.1 PRNG Quality

An individual’s performance as a PRNG is measured by the way its output varies when a single input bit is changed.

Ideally, when a single bit in the input is flipped, on average half of the output bits should change.

To describe in more detail: to evaluate the performance of the i th individual as a PRNG, a set of 8 random 32-bit integer inputs $a_0 \dots a_7$ is generated by a Mersenne Twister PRNG. The output $r_i(a_0 \dots a_7)$ of the individual for these inputs is evaluated, this is also a 32-bit integer. Then one randomly selected bit of one randomly selected integer input is flipped to provide a new set of inputs $b_0 \dots b_7$ and $r_i(b_0 \dots b_7)$ evaluated. This data constitutes the result of one test case, and 4096 such test cases are used to completely evaluate one individual.

This fitness component of an individual is then calculated as defined by the Strict Avalanche Criterion (SAC), first introduced by Webster and Tavares [22] and analysed in detail by Forré [10]. The criterion measures nonlinearity, specifically the expected distance between outputs given a single bit flip in the input. Each output bit should have a probability of 0.5 of being flipped when a single input bit is changed, in order to maximise the nonlinearity of the PRNG. Hence, the Hamming distance between the two outputs should follow the binomial distribution $B(n, \frac{1}{2})$. By recording the Hamming distance between $r_i(a_0 \dots a_7)$ and $r_i(b_0 \dots b_7)$ for each test case, a χ^2 squared goodness-of-fit measure can be calculated against the ideal binomial distribution of bit flips. The performance measure of an individual i is given by:

$$\sum_{i=0}^n \frac{(C_i - E_i)^2}{E_i} \quad (2)$$

Here, n is the number of possible bit flips ($0 \dots 32$), C_i is the observed number of test cases that resulted in i bit flips, i.e. where the Hamming distance was i between $r_i(a_0 \dots a_7)$ and $r_i(b_0 \dots b_7)$, and E_i is the expected number of tests with i bit flips according to the binomial distribution. Our aim is to minimise this statistic, and reduce the deviation from the ideal distribution.

Note that in order to test an evolved PRNG, we employ a Mersenne Twister PRNG! This circularity does not, however, pose a problem for fitness evaluation. The only danger would be if an evolved PRNG managed to “take advantage” of some predictable characteristics of the input integers, generated by the Mersenne Twister. Given that the Mersenne Twister algorithm is designed not to betray such characteristics, this appeared to be an unlikely outcome!

3.2.2 Power Consumption

To produce programs that trade-off non-functional criteria, we must be able to measure the specified property, in this case the power consumption of a program. We must also be able to perform this evaluation in a way that is computationally acceptable. For this work, we simulated each program’s execution using the `Wattch` [5] extension for the `SimpleScalar` [6] simulator. `Wattch` is a cycle-level power simulator based on a parameterised processor model, and provides overall power estimates for a program’s execution, calculating power consumption P_d for different logic units using the following equation:

$$P_d = CV_{dd}^2 af \quad (3)$$

C is the load capacitance, V_{dd} the supply voltage, f the clock frequency and a an activity measure that estimates the

amount of transistor switching. The values of these parameters are partly estimated; however more detail of how they are derived and a good validation of their results is given by Brooks et al.[5].

Wattch was designed as a tool to explore trade-offs between processor architecture design, and to enable compiler writers to optimise their software. Its intended use to provide “speed versus power consumption” trade-off is replaced here with a “quality of output versus power consumption” goal. As a result of the design objectives of Wattch, care has been taken to ensure it is relatively fast to execute hence a program can be evaluated in tens of seconds, an expensive fitness evaluation, but viable when used in a parallelised framework.

The version of Sim-Wattch we used was an unaltered distribution version, v1.02, compiled for SimpleScalar’s PISA instruction architecture, rather than a custom platform.

3.3 Multi-objective Optimisation

In order to carry out multi-objective optimisation, an implementation of the SPEA2 algorithm was written as an extension to ECJ, which followed precisely the original algorithm specified by Zitzler et al.[23]. The algorithm retains an archive of non-dominated individuals, which are individuals that cannot be improved upon in terms of all objectives by any other single individual within the archive. The algorithm attempts to use the archive to approximate the “Pareto front”, a surface of non-dominated individuals within objective space. The archiving is effectively elitist, and counteracts the emergence of bloat in GP, because a larger individual will only survive if it makes an improvement over the existing archive in at least one objective. This is an important feature in attempting to evolve low-power individuals, because unchecked bloat would increase the number of instructions in individual solutions, which in general tends towards a higher power consumption.

SPEA2 includes a niching function to ensure the sample of the Pareto front defined by the archive is representative of the trade-offs that can be made. See Deb and Kalyanmoy [9] for further details of multi-objective optimisation.

3.4 Problem Summary

To summarise, the problem to be solved was to find a pareto front of programs, where each program implemented a function $r(a_0 \dots a_7)$. The two objectives, both of which were cost functions to be minimised, were the power consumption of the program and the measure of functionality given by the χ^2 goodness of fit measure.

3.5 Implementation

Figure 1 gives an overview of the way that the ECJ-based framework measures the fitness of a program. The PRNG is a symbolic expression, and is written out to a file as a C function representing the PRNG. This file is then compiled and linked with test harness code, which measures the fitness of a program as previously described calculating the χ^2 measure. The test cases are generated using a Mersenne Twister PRNG [2]. The program is then run on the Sim-Wattch simulator, which produces power statistics along with redirected program output from the test harness i.e. the χ^2 measure. These fitness values are used by SPEA2 to assign fitness scores based on Pareto non-dominance and a niching function.

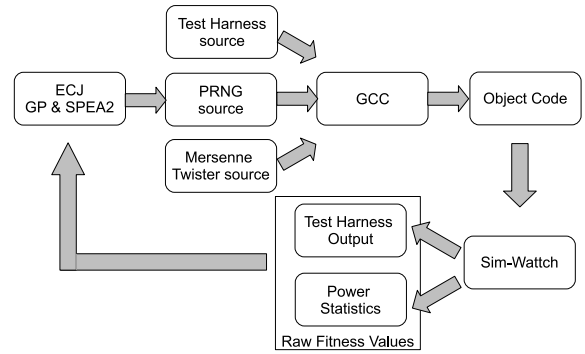


Figure 1: Overview of Fitness Evaluation

An alternative arrangement we considered was to pass test case data to and from the simulator via streams or sockets, but this proved too inefficient to be feasible. By placing the entire fitness evaluation (test case creation, the program under test and the goodness-of-fit calculation) within the simulator, the run-time of a single program evaluation was greatly reduced. As a further improvement, the expected distribution for a given number of test cases was calculated once and hard-coded in the test source. However, even with this efficient arrangement, a single evaluation of an individual under 4096 test cases took approximately half a minute on a 4200+ AMD processor. Most of this time was spent within the Sim-Wattch simulator, hence the only remaining target for optimisation was to reduce the sample size.

3.6 Reducing Sample Size

The paper that this work builds upon [16] used 16384 test cases, whereas in our work we chose to use only 4096 test cases. This section justifies that decision.

To reduce the number of test cases required, we evaluated the variance of functional fitness measures on smaller sample sizes using a bootstrapping resampling technique [3]. Firstly, we executed a single run using the larger test sample size over 4 days and logged each individual in the archive. A selection of 5 individuals were chosen across a range of different fitness values to provide the required data. These individuals were evaluated and the $r(a_0 \dots a_7)$ and $r(b_0 \dots b_7)$ values logged over 16384 test cases. This allowed us to employ statistical bootstrapping methods to determine whether smaller sample sizes were effective in estimating the χ^2 measure.

An example plot is given in Figure 2. This illustrates for a single program the impact of varying the sample size on the resulting fitness measure. The samples sizes run from 0 to 16384 in powers of 2. Each point is the result of using bootstrapping with 30 bootstrap samples. Other results had similar or smaller variance, and from these plots we concluded a sample size of 4096 was sufficient. It was necessary to ensure that the smaller sample size did not adversely affect power consumption statistics, and similarly the variance of the power statistics for each program across each sample size was empirically found to be very low at a sample size of 4096. Note that during the experiments each individual was also re-evaluated at each generation, reducing the chance of a single outlier fitness measure incorrectly giving an individual a higher priority when populating the next archive.

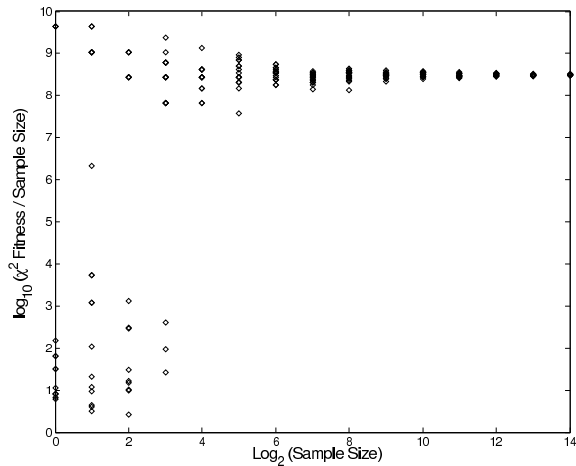


Figure 2: Example Plot of Sample Size against Fitness for one Program

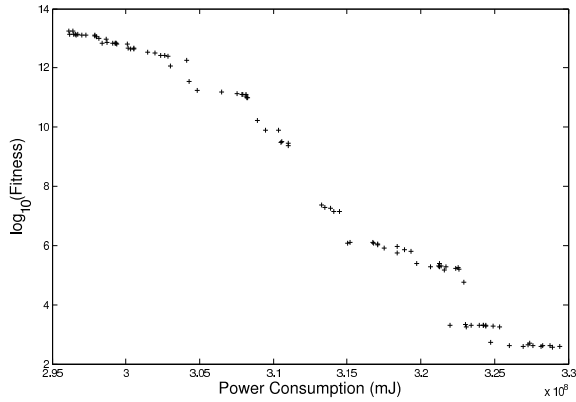


Figure 3: Archive at Generation 249, Experiment 1. The graph shows the trade-offs made by programs within the archive, between total power consumption and error. For both objectives, lower values are better.

4. RESULTS

4.1 Example Pareto Fronts

Figure 3 shows the archive at generation 249 of Experiment 1, where each point corresponds to a program’s properties in objective space. The power consumption is the total power consumed by each individual across all 4096 test cases. Fitness is the goodness-of-fit measure as described in Section 3.2.1. Note that the archive is not always composed entirely of non-dominated individuals for two reasons: firstly it may be “topped up” with dominated ones by the SPEA2 algorithm when there are too few non-dominated individuals to fill the archive completely, and also due to the variance in fitness values (which are input-dependent) when the archive is re-evaluated at each generation.

This graph demonstrates that functional trade-offs are indeed possible for this problem. The most impressive PRNGs, at the bottom-right corner of the diagram, have a small deviation from the binomial distribution, and these require the

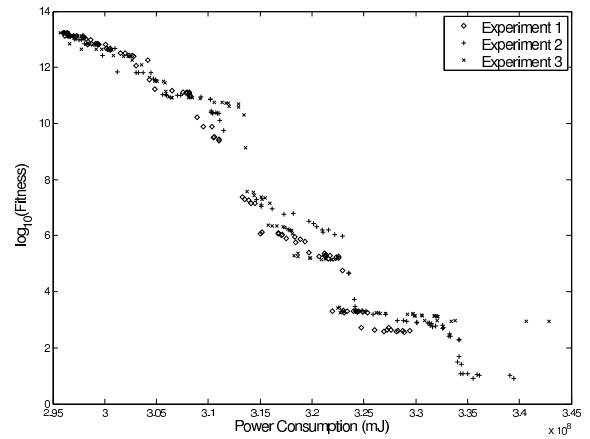


Figure 4: Archives at Generation 249 for Experiments 1, 2 and 3. Similar trade-offs are discovered.

most power. Very poor pseudorandom number generators, with low power requirements, are at the top-left of the diagram. This diagram allows us to visualise the relationship between power and functionality of the problem.

There is a discontinuity in the archive, where a step improvement in the nonlinearity of the evolved PRNGs is observed. This is caused by the simple niching function used by SPEA2 algorithm, which works on Euclidean distance between points in objective space. As the fitness values are not normalised, from this point to the right of the graph the power consumption, rather than PRNG performance, dominates the niching function. This will have some impact on the variety of solutions produced by the search, depending on how often the niching function is used by the search. The use of a more sophisticated niching function enables further control on how the archive approximates the pareto front.

Figure 4 shows three archives resulting from three separate experiments, with different seeds for ECJ being the only difference between each experiment. Similar trade-offs are achieved, although Experiment 2 shows much better performance towards the lower range of the fitness values.

These results are an order of magnitude worse, in terms of the functional fitness (SAC), than the results presented by Lamenca-Martinez et al.[16]. The aims of this work is not to improve on those results, however Experiment 2 was extended for an extra 50 generations to demonstrate that the quality of solutions found was not compromised. The Pareto front was improved by a small amount and χ^2 values as low as 32.0 were obtained, versus the 12.7 reported by Lamenca-Martinez et al[16]. These are both excellent values: see Figure 6 for an indication of the quality of the best individual from Experiment 2.

4.2 Function Usage

Figure 5 shows the number of programs using each function in the function set at least once for the 300 individuals contained in the final archives from Experiments 1, 2, and 3. This diagram gives an intuitive impression of how important each function within the function set was for developing useful trade-offs. The figure begins to answer the question of whether the MULT function should be included in the function set: at this stage it does indeed appear to be a

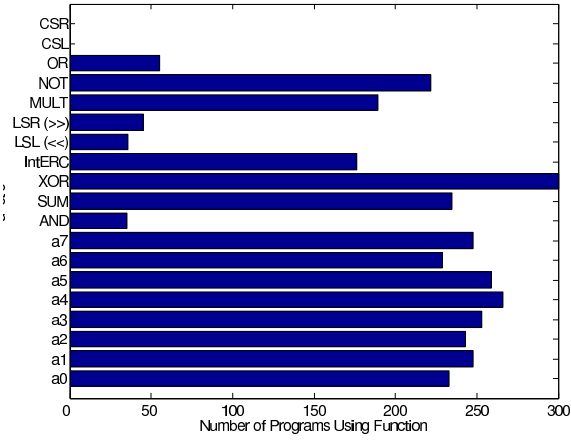


Figure 5: Function set usage across the archives in Generation 249 for Experiments 1, 2 and 3

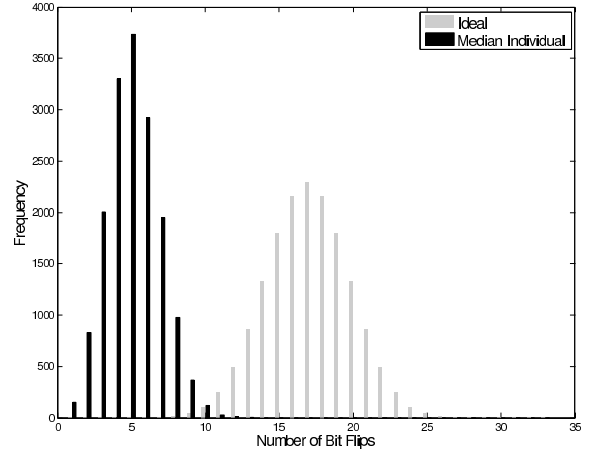


Figure 7: Distribution of Bit flips of the Median (by χ^2 Fitness) individual from Experiment 2

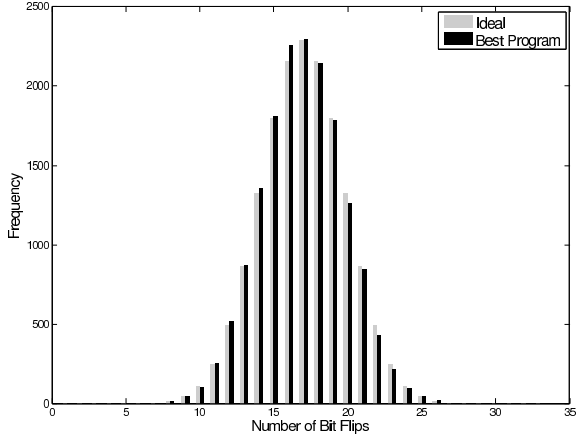


Figure 6: Distribution of Bit flips of the best individual from Experiment 2

useful tool in making trade-offs, as it is used by most of the programs on the Pareto front. This question is addressed in more detail in Section 4.4.

From the chart, the most striking feature is the lack of Circular Shift Left (CSL) and Circular Shift Right (CSR) function calls. As C does not provide these functions as standard, they were provided within the test harness as functions (in the same way as [16]). The function calls are expensive, both because of the overhead in calling a function, and because the CSL and CSR functions required several lines of code. Hence they were discarded by the search. However, instruction sets for processors such as the Z80 do provide such rotate instructions. By evolving programs in assembly language, the search could take advantage of these commands. This raises an important issue of how the target language impacts the ability of search to make trade-offs.

4.3 Example Individuals

Three example individuals are shown in Table 2. These individuals are the best (in terms of SAC) from the extended run of Experiment 2, and the worst and median from Ex-

periment 1. The χ^2 measure of the best individual is the average of 10 runs over 16384 test cases, as the variance in the fitness function becomes significant at very low χ^2 values.

The distribution of the bit flips from test cases for the best individual from Experiment 2 (extended to 300 runs) is given in Figure 6. Comparing this to the median individual of Experiment 1, which is given in Figure 7, these diagrams illustrate the dependence of quality on power, or “bang for your buck” as far as random number generation is concerned.

It is interesting to note that the best individual in Table 2 contains a deal of self-similarity, and also that the use of the MULT function is distributed across different parts of the program tree. This would appear to be a sensible way of managing the “energy budget” of an individual through placement of the most expensive function.

The best individual was then used to generate a 250MB file of random bytes, by initialising $a_0 \dots a_7$ with random numbers and from then onwards feeding the previous output into the next input at position 7, as described by Lamenc-Martinez[16]:

$$a_i^{n+1} = a_{i+1}^n \quad \forall i = 0 \dots 6 \quad (4)$$

$$a_7^{n+1} = r^n(a_0 \dots a_7) \quad (5)$$

The file was then run through the ENT test suite [1], a standard battery test used to evaluate PRNGs. The results are given in Table 3. These results indicate that with this feedback method the individual performs well. However, it performs poorly given a low-entropy input such as feeding sequential numbers as inputs, particularly on the ENT χ^2 test (not to be confused with the avalanche criterion test used in this paper!). It is not surprising that this is the case, because fitness evaluation used random rather than sequential inputs. More surprisingly, the best individual “Lamar” as reported by Lamenc-Martinez [16] performs well under a low-entropy input. It is possible that a comparable number of experiments would have to be completed to achieve comparable results, in particular to achieve a similarly low value for the χ^2 objective. An alternative method may be to incorporate low-entropy tests into fitness evaluation.

Relative PRNG Performance	Program Expression	Power Consumption (mJ)	χ^2 Fitness
Best (Experiment 2)	$(2307363674 \oplus (a2 * a6)) + ((\neg(\neg(\neg(\neg((a2 * a6) \oplus (a7 \oplus a1)) * (a0 \oplus a3)) \oplus (a2 * a6)) + (a5 * a4)) >> 2307363674) \oplus (a0 \oplus a3)) + \neg((a5 * a4) * ((2307363674 \oplus (a7 \oplus a1)) + (a0 \oplus a3)))$	$3.3658 * 10^8$	32.0
Worst (Experiment 1)	$\neg(\neg(1997453768))$	$2.9639 * 10^8$	$1.76 * 10^{13}$
Median (Experiment 1)	$a2 \vee \neg(((a2 + a0) * ((a4 \oplus ((a6 + a5) \oplus a7)) + (a1 \oplus a3))) \wedge \neg(\neg(1997453768)))$	$3.2892 * 10^8$	$2.39 * 10^9$

Table 2: Three example PRNGs of varying quality

Test	Best Individual
Entropy	7.999999 bits/byte
Compression Rate	0%
χ^2 Statistic	264.98 (32%)
Arithmetic Mean	127.5011
Monte Carlo π Estimation	3.141828142 (0.01%)
Serial Correlation Coefficient	0.000010

Table 3: ENT Results for Best Individual

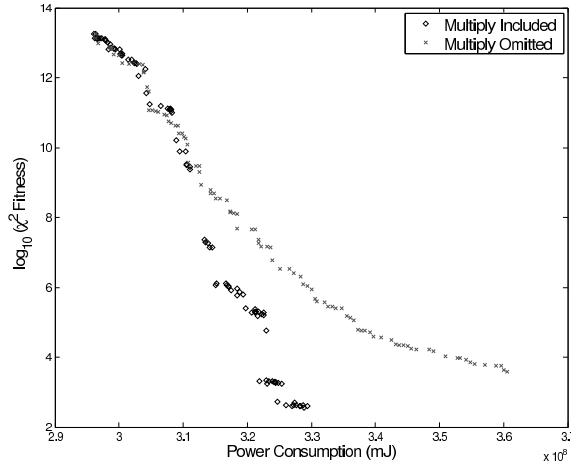


Figure 8: Pareto Front from Experiment 1 compared to Pareto Front without Multiply Function

4.4 Impact of the MULT Function

In previous work [11, 16], there was some discussion as to whether the MULT function should be included in the function set, the argument being that whilst it is likely to aid greatly in achieving the functional goals of pseudorandom number generation, it is an inefficient operation. Using a systematic approach based on multi-objective optimisation, we can address this issue more directly, by comparing the Pareto fronts that result under experiments using (a) the function set described above and (b) the same function set with the MULT function removed.

Figure 8 displays both Pareto fronts, and the difference is quite striking. Without the MULT function, the Pareto front is stretched to the right, in that the same level of functionality requires an increased amount of power. Furthermore, certain levels of functionality do not seem achievable without it, at least given the computational resources provided to the search.

5. CONCLUSIONS

Satisfying trade-offs at the software-hardware interface is an important problem that is amenable to a Search-Based Software Engineering approach. In particular, this work demonstrates that GP can be used in combination with MOO as an effective SBSE method for exploring the trade-offs between power consumption and functionality when creating a pseudorandom number generator. It also demonstrates that continuous trade-offs are possible for the PISA-based SimpleScalar architecture. A designer may choose a satisfactory compromise from the set provided.

To achieve the same level of functionality within the results of a MOO run requires increased computation power, in this case we extended the search from 250 generations to 300, i.e. an increase of 20% in terms of evaluations, genetic operators etc.

For this specific problem, the MULT function is a useful function to include in the function set. Whilst the power consumption of any instruction varies depending on the architecture, it is likely that this conclusion will generalise to other architectures.

6. FURTHER WORK

This work is an initial attempt to use SBSE techniques to help software engineers to produce software that effectively satisfies multiple functional and non-functional criteria. Straightforward extensions of this work would be to apply the same method to other common applications, for example hash function generation, or to other non-functional properties such as memory requirements or footprint size within a FPGA. Whilst only two objectives were examined here, many more exist in practice and incorporating an increased number of objectives is also a future goal of this work. Similarly, the same method can be applied to other architectures provided that appropriate simulation tools are available, and the impact of the selected architecture on the Pareto fronts generated analysed.

The target language used will affect the ability of both software engineers and heuristic search to locate trade-offs.

Access to finer-grained trade-offs through applying search at the assembly language level may improve the overall results obtained. Comparisons between this and compiler-based optimisations can also be carried out.

With the results provided, a human designer would have to select a solution with satisfactory trade-offs. This decision making process is somewhat limiting, as is the amount of information that an engineer must assess, which grows with the number of objectives. One way of circumventing this problem is to evolve resource-scalable programs (such as energy-scalable algorithms, see Sinha et al.[20]). A single resource-scalable program can provide a variety of trade-offs.

Finally, the replacement of simulation to measure fitness could improve the accuracy and reliability of our results. One method we are examining is to use intrinsic power measurement, in order to measure non-functional properties of software running on a physical embedded system.

7. ACKNOWLEDGEMENTS

This work is part of the Software Engineering by Automated Search (SEBASE) Project and is supported by an EPSRC grant (EP/D050618/1). We gratefully acknowledge this support. We would also like to thank Andrea Arcuri, Iain Bate, Julio Cesar and Juan M.E. Tapiador for their feedback and suggestions.

8. REFERENCES

- [1] Ent: A pseudorandom number sequence test program. <http://www.fourmilab.ch/random/>.
- [2] Mersenne Twister PRNG, University of Hiroshima. <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>.
- [3] M. Berthold and D. J. Hand. *Intelligent Data Analysis: An Introduction*. Springer-Verlag, 1999.
- [4] B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*. Springer, 2005.
- [5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.
- [6] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, Computer Sciences Department. University of Wisconsin-Madison, 1996.
- [7] J. Clark, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Software, IEE Proceedings*, 150:161–175, 2003.
- [8] C. A. Coello. An updated survey of GA-based multiobjective optimization techniques. *ACM Computing Surveys*, 32(2):109–143, 2000.
- [9] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.
- [10] R. Forré. The Strict Avalanche Criterion: spectral properties of boolean functions and an extended definition. In *CRYPTO '88: Proceedings on Advances in cryptology*, pages 450–468. Springer-Verlag, 1990.
- [11] J. C. Hernandez, P. Isasi, and A. Sez nec. On the design of state-of-the-art pseudorandom number generators by means of genetic programming. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1510–1516, 2004.
- [12] J. Jannink. Cracking and Co-Evolving Randomizers. *Advances in Genetic Programming*, pages 425–443, 1994.
- [13] D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley, November 1997.
- [14] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 1989.
- [15] J. R. Koza. Evolving a computer program to generate random numbers using the genetic programming paradigm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 37–44. Morgan Kaufmann, 1991.
- [16] C. Lamenc a-Martinez, J. C. Hernandez-Castro, J. M. Estevez-Tapiador, and A. Ribagorda. Lamar: A new pseudorandom number generator evolved by means of genetic programming. In *Parallel Problem Solving from Nature IX*, volume 4193, pages 850–859. Springer-Verlag, 2006.
- [17] S. Luke. ECJ: A Java-based Evolutionary Computation Research System. <http://cs.gmu.edu/~eclab/projects/ecj/>, 2007.
- [18] B. Mesman, L. Spaanenburg, H. Brinksma, E. Deprettere, E. Verhulst, F. Timmer, H. van Gageldonk, L. Eggermont, R. van Leuken, T. Krol, and W. Hendriksen. *Embedded Systems Roadmap – Vision on technology for the future of PROGRESS*. STW Technology Foundation, 2002.
- [19] M. O’Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Springer, 2003.
- [20] A. Sinha, A. Wang, and A. P. Chandrakasan. Algorithmic transforms for efficient energy scalable computation. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 31–36. ACM Press, 2000.
- [21] M. Sipper and M. Tomassini. Co-evolving parallel random number generators. In *Parallel Problem Solving from Nature – PPSN IV*, pages 950–959. Springer, 1996.
- [22] A. F. Webster and S. E. Tavares. On the design of S-boxes. In *Advances in Cryptology – Crypto '85*, pages 523–534. Springer-Verlag, 1986.
- [23] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Swiss Federal Institute of Technology, 2001.