Feedback-based Coverage Directed Test Generation: An industrial evaluation

Charalambos Ioannides [1,2] , Geoff Barrett [2] , Kerstin Eder [3]

[1] Industrial Doctorate Centre (IDC), University of Bristol
Queen's Building, University Walk, Bristol BS8 1TR, UK
[2] Broadcom BBE BU, Broadcom Corporation
220 Bristol Business Park, Coldharbour Lane, Bristol BS16 1FJ, UK
[3] Department of Computer Science, University of Bristol
MVB, Woodland Road, Bristol BS8 1UB, UK

{Charalambos.Ioannides, Kerstin.Eder} AT bristol.ac.uk,
gbarrett AT broadcom.com

Abstract. Although there are quite a few approaches to Coverage Directed test Generation aided by Machine Learning which have been applied successfully to small and medium size digital designs, it is not clear how they would scale on more elaborate industrial-level designs. This paper evaluates one of these techniques, called MicroGP, on a fully fledged industrial design. The results indicate relative success evidenced by a good level of code coverage achieved with reasonably compact tests when compared to traditional test generation approaches. However, there is scope for improvement especially with respect to the diversity of the tests evolved.

Keywords: Microprocessor Verification, Coverage Directed Test Generation, Genetic Programming, MicroGP

1 Introduction

A consistent trend in the semiconductor industry is the increase of embedded functionality in new designs. The drivers are competition on improving the quality of designs, the miniaturization level achievable as well as the standardization and automation (maturity) of the design process (i.e. digital design languages and EDA tools). Conversely, the verification process today requires a significant amount of resources to cope with these increasingly complex designs. Concern is expressed that this has formed a bottleneck in the development cycle making verification a critical and time consuming process now reaching "crisis proportions" [1].

There is intrinsic risk associated with verification for the following reasons. First, the verification process is unpredictable in nature. That is why automation is important in increasing its robustness, repeatability, maintainability and thus ultimately its predictability. Second, seeing that exhaustive simulation is commercially unacceptable, the quality of verification relies on engineers selecting

the scenarios to verify. This by definition makes the process and the quality criteria subjective.

In order to alleviate the problem, industrialists and academics have proposed and improved on many formal, simulation-based and hybrid verification techniques which all aim to make the process less time consuming, more intuitive to the verification engineer and more reliable in accurately indicating the level of its completeness.

In an attempt to further automate the process, especially in simulation-based and hybrid approaches, Machine Learning (ML) techniques have been exploited to close the loop between coverage feedback and test generation. Although most techniques in the literature are reported to help in constructing minimal tests that cover most, if not all, of the Design Under Verification (DUV) in a small amount of time, there is always the question of how useful these techniques are when applied in a real-world industrial-level verification environment. Questions arise as to how long it would take a non-ML expert to setup and use these techniques in his environment, also how effective they would be on more complex real-world designs.

One of these techniques is called MicroGP (uGP). It was developed by a team of researchers at the Polytechnic University of Torino [2]. MicroGP uses an evolutionary algorithm which, based on a customized instruction library corresponding to the instruction set architecture of a particular processor, evolves a set of test programs the best of which aims to achieve maximum code coverage. Their evolution is guided by code coverage metrics [3] and according to the experimental results published in numerous papers [4-8], uGP is capable of evolving small tests that achieve almost complete coverage closure.

This work aimed to evaluate a publicly available version [9] of uGP on several aspects. The first was the effort (man months) taken to incorporate it in an existing verification flow at Broadcom's Bristol site. Second, we needed to assess its potential in generating tests for an in-house SIMD digital signal processor called FirePath™. Additionally, the computational effort (runtime) was to be compared against the benefits provided. Furthermore, the team opted for more experience and insights as to what ML has to offer in coverage directed test generation within an industrial context, in comparison to existing in-house techniques. Finally, this work was also aimed to provide insight into requirements for developing more effective future tools.

This paper is structured as follows. In Section 2 background knowledge is provided on subjects such as Coverage Directed test Generation and Evolutionary Algorithms, as well as on subjects directly relevant to the work presented e.g. the tool MicroGP, the design under verification and the test generator used. Section 3 explains on the incentives behind as well as the experimental setup chosen for this work. The following section illustrates the results obtained after performing the listed experiments, while in Sections 5 and 6 these results are being discussed and concluded with potential future improvements to be added.

2 Background

2.1 Coverage Directed Test Generation

Coverage Directed test Generation (CDG) is a simulation-based verification methodology which aims to reach coverage closure automatically by using coverage data to direct the next round of test generation towards producing tests that increase coverage. There are two main approaches towards CDG: one is *by construction* using formal methods and the other is based *on feedback*. Examples of the former [10] require a formal model of the DUV (e.g. an FSM) which is used to derive constraints for test generation that accurately hit a specific coverage task. This approach has inherent practical limitations because the formal models can be particularly large especially as designs become more complex. In contrast, CDG *by feedback* techniques, e.g. [11] or [12], employ ML methods to close the loop between coverage analysis and test generation. This is achieved by learning from the existing tests and the achieved coverage the cause and effect relationships between tests, or test generation constraints, called bias, and the resulting coverage. These relationships are then utilized to construct new tests or to create new constraints for a test generator in such a way that coverage closure is achieved faster and more reliably.

2.2 Evolutionary Algorithms

Broadly speaking, Evolutionary Algorithms (EA) [13] are search methods aiming to optimize some user defined function, called the Fitness function. The potential solutions can be represented in a variety of forms, e.g. bit strings, graphs, S-expressions [14], etc., depending on the chosen problem and specific technique used. Techniques include Genetic Algorithms (GA), Genetic Programming (GP), Evolution Strategy (ES) and Evolutionary Programming (EP). All these techniques employ a set of genetic operators which mimic biological processes, in order to perform a search in the syntactic space of the problem representation chosen. The most common operators used are recombination, also called crossover, and mutation (see Figure 1).

Recombination requires two candidate solutions (parents) to recombine their genetic material in producing two or more offspring. Mutation requires a single parent and depending on some probability its genetic material is altered to some random legal value.

Evolutionary Algorithms require a population of individuals each representing a solution to the given problem. Each of these has a fitness value assigned to it according to how well it solves the problem. There are many alternative choices in setting up such an algorithm but a typical learning epoch involves a set of new solutions being evaluated and then ranked on their fitness. Then, according to some selection scheme, the best are used as parents which will form the new set of solutions while the rest are discarded. This process continues until some convergence criterion is met. Through a series of learning epochs they guarantee convergence to some local or global optimum in the fitness landscape formed by the Fitness function.

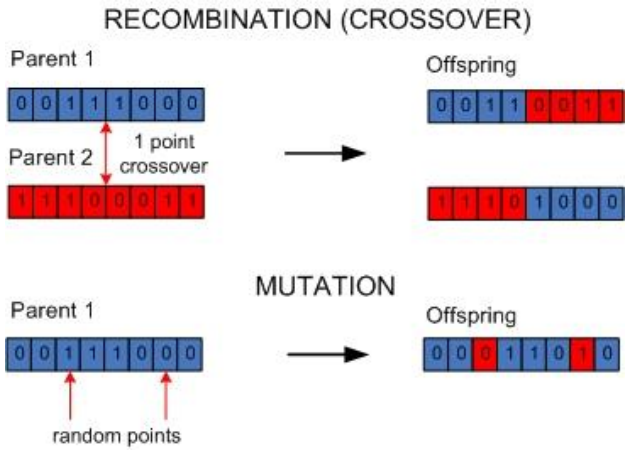

Fig. 1. Crossover and Mutation Genetic Operators

2.3 MicroGP

The purpose of uGP is to generate correct instruction level tests which either achieve maximal code coverage results at register-transfer level (RTL) or detect maximal faults at post-silicon level designs.

MicroGP was developed to be broadly adaptable to any microprocessor design. Its architecture contains three main elements; these are an instruction library, an evolutionary core and an external evaluator (see Figure 2).

Fig. 2. MicroGP overview (from [2])

The instruction library (IL) is a text file describing the Instruction Set Architecture (ISA) of interest. The basic building block is the macro and several of those are grouped into sections. Each macro describes the syntax for a given instruction or list of meaningfully grouped instructions. The sections try to mimic traditional assembly code sections representing register file settings, code main body and subroutines.

The evolutionary core utilizes a μ+λ selection scheme, i.e. it operates on a population of μ individuals in order to produce λ offspring for later evaluation at each learning epoch. Each individual is a test sequence represented by a directed acyclic graph with its nodes pointing to a macro inside the IL (see Figure 3). At each learning epoch a set of genetic operators (crossover and mutation) are applied on the μ (parent) population producing the λ (offspring) population. The offspring are evaluated through simulation (external evaluator) and added to the μ population. The individuals are then sorted according to their fitness (achieved coverage) and since the parent population is of a fixed size any additional individuals are deleted. This makes sure only the better tests are kept from epoch to epoch, thus guiding the evolution of a test program that achieves maximum fitness. The learning process continues until either maximum coverage is achieved, the population has reached a steady state (i.e. no improvements for a preset number of epochs), or if a number of maximum epochs has elapsed without the previous two conditions being met.

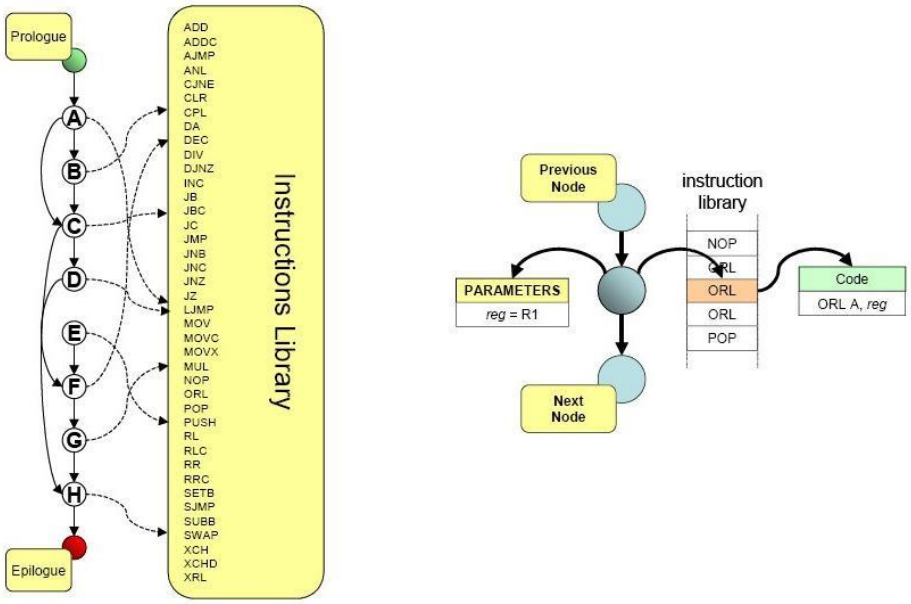

Fig. 3. MicroGP graph representation (from [5])

The external evaluator is any configuration chosen by the user of the tool that should be able to execute the tests produced by the evolutionary core and evaluate them according to fitness. Depending on the level at which verification is performed the external evaluator and hence fitness value it provides can be one of many things. In the majority of applications of uGP, the level of design abstraction was the RTL and the fitness metric was the achieved statement coverage [4-8].

In addition, uGP utilizes a self-adaptive mechanism that updates the activation probabilities of the genetic operators. These probabilities, in general, have the property of speeding or slowing the learning process given the problem solved and the individual representation chosen. The use of a self-adaptive mechanism here means

that this level of detail when applying an EA is hidden, thus making uGP a tool that can be more easily used by non-ML experts.

In the past the tool has been used on various microprocessor designs targeting different coverage metrics. Some of them are the DLX/pII [8] and LEON2 (SPARC V8 architecture) [7] on code coverage. The i8051 microcontroller was verified using code coverage [5], fault coverage [6] and path delay [15] while more generic peripheral modules (PDU, VIA, UART) have been verified using both code and FSM transition coverage in [16]. In all of the aforementioned work uGP consistently provided smaller tests than either randomly created or pre-compiled tests while achieving higher coverage in all and maximum coverage in some of the applications.

2.4 FirePath

FirePath is a fully-featured 64-bit LIW DSP processor [17]. The design has been in production for 8 years and is the main processing element in Broadcom's Central Office DSL modem chips. This chip is installed in the telephone exchange and is estimated to terminate around 50% of all DSL connections worldwide. The processor is in continuous development as new demands for functionality materialise and as new ideas for improvements in speed, area or power are implemented. No logic bugs have ever been discovered in this chip in the field. However, although the verification environment is very thorough, its maintenance is a labour-intensive process which will benefit from greater automation.

In brief, the processor consists of a large register file, 4-6 SIMD execution pipes and 2 load/store pipes (of which a total of 4 can be activated by a single instruction). Figure 4 shows an architectural overview of the FirePath processor.

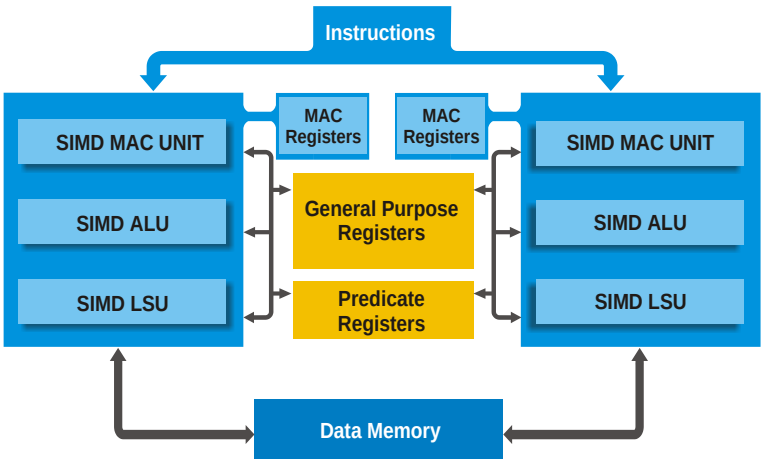

Fig. 4. FirePath architecture overview

2.5 FireDrill

FireDrill is a constrained-random test generator that produces the instruction streams at the heart of FirePath's verification environment. The behavioural model of the FirePath is integrated with the test generator so that the generator has full visibility of the architectural state of the processor while generating each instruction. The generator manages constraints governing correct formation of instructions (which operations can be combined in the same packet, valid combinations of destination registers and addresses, etc.) and ensures that each generated test will execute correctly and terminate (loops are bounded, code and data regions correctly separated, etc).

Test suites are generated to target various functional coverage metrics [3] using hand-written bias files which can specify instruction sequences designed to increase the likelihood of covering certain corner cases. Each time new features are added to the processor, more metrics and associated bias files are created in order to cover the new features. Coding the metrics, debugging them, analysing the coverage and biasing into the gaps consumes most of the verification engineers' time and is on the critical path of RTL delivery. The test generation environment is shown in **Fig. 5.** FirePath Test Generation Environment.

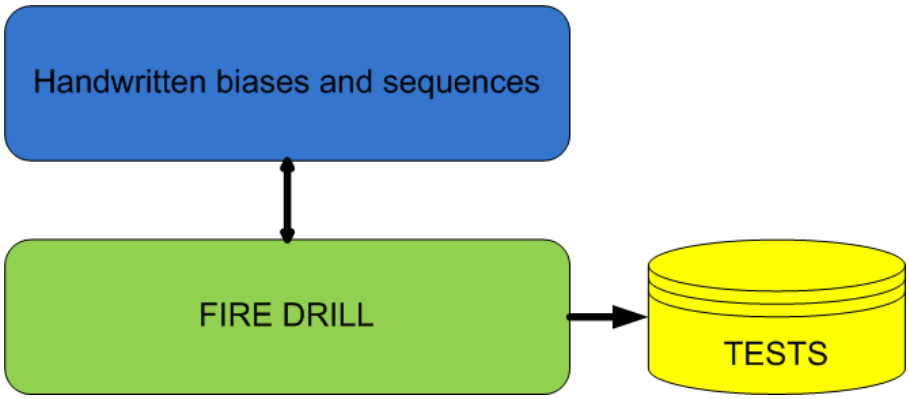

Fig. 5. FirePath Test Generation Environment

3 Experimental Setup

The aim of these experiments was primarily to see whether the latest ML-driven CDG technology, in the form of the uGP tool, was fit to cope with the complexity of a fully fledged industrial design. An additional aim was to assess how other methods like purely random tests generated by our unconstrained pseudo-random test generator (i.e., FireDrill with no handwritten biases or sequences applied), would compare against uGP generated tests. The test generation setup with MicroGP is shown in **Fig. 6.** MicroGP Test Generation Environment.

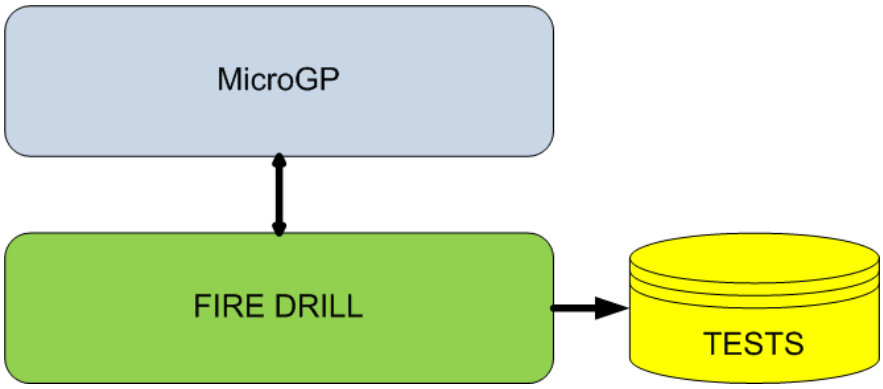

Fig. 6. MicroGP Test Generation Environment

Another element to be assessed was the amount of time (man months) and ML expertise required to setup the environment for such experiments.

In order to reduce the amount of setup required, we avoided instructions with complicated constraints such as loops and configuration instructions which require barriers. Address generation for branch and load/store instructions was handled by the unconstrained FireDrill generator subject to abstract constraints controlled by MicroGP (eg, forward/backward branches, load/store to local or shared memory). If, for any reason, it was not possible to generate an address as specified, the instruction requested by MicroGP was ignored.

3.1 Overview

The uGP version used for performing the tests was 2.7.0 beta as found on [9]. It consists of 31 files containing ~8000 lines of C code. The only requirements of the program are the creation of the IL, making sure the tests created by uGP are fed to the simulator and finally that the coverage feedback is given to uGP in a text file of specific format. The experimental setup chosen is shown on Figure 7.

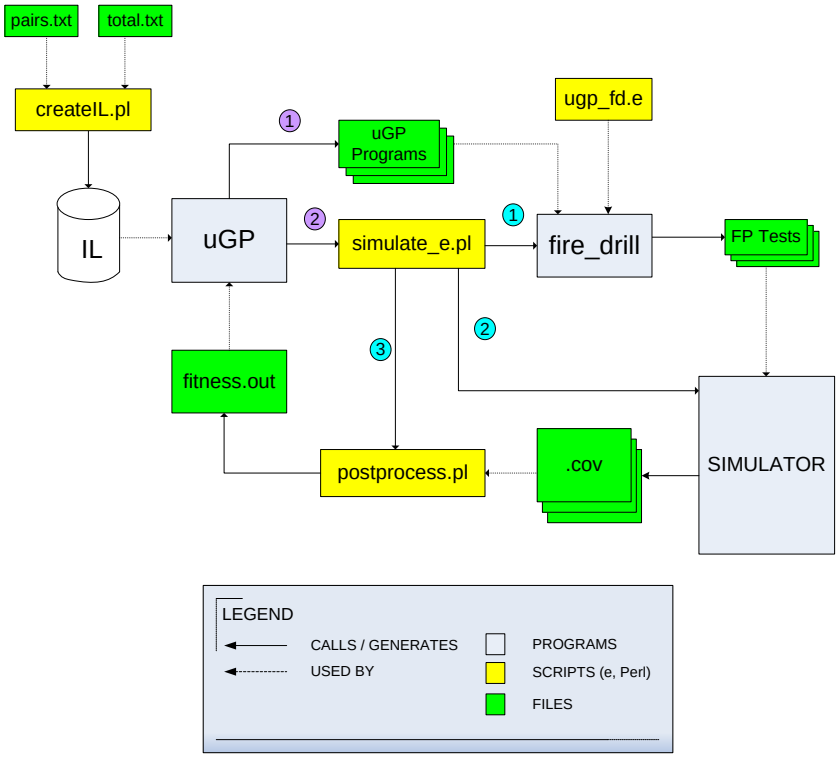

Fig. 7. MicroGP experiment setup

The process starts by creating the IL via the custom made Perl script. The script reads a database containing the operations and constraints between classes of operations according to the SIMD instructions of FirePath and constructs the IL database. Once uGP is executed it uses the IL to create and evolve test programs. At every learning epoch once new individuals (tests) are constructed the executable calls another Perl script (*simulate_e.pl*) which does three things. It first calls the FireDrill test generator to read the uGP tests with the help of a constraints file written in the e language [18] (*ugp_fd.e*) so that appropriate FirePath tests are created. The reason for this step is explained in Section 3.2. Then, once the FirePath tests are created, the script calls the simulator (external evaluator). Once all the tests have run, *simulate_e.pl* calls another script (*postprocess.pl*) to extract the coverage achieved by those and create the *fitness.out* file. This is used by the uGP executable to assign a fitness value to the tests it created. The process described continues until the termination criterion is met, i.e. 100 epochs have elapsed. This termination criterion was chosen in favor of any convergence related ones, so as to assess the full potential of the algorithm.

3.2 Test Program Representation

The IL used by uGP, was automatically constructed from the FirePath instruction set database of over 700 operations. Each macro included in the IL had the structure shown in Figure 8. In order to maximize reuse of existing constraints, shorten the learning process and in particular ensure validity of constructed tests, uGP evolved test programs that would direct FireDrill in constructing the final tests to be simulated. Arithmetic operations were explicitly stated in the macros and thus chosen by uGP, while Load/Store and Branch operations were decided by FireDrill according to the biases put in place by uGP.

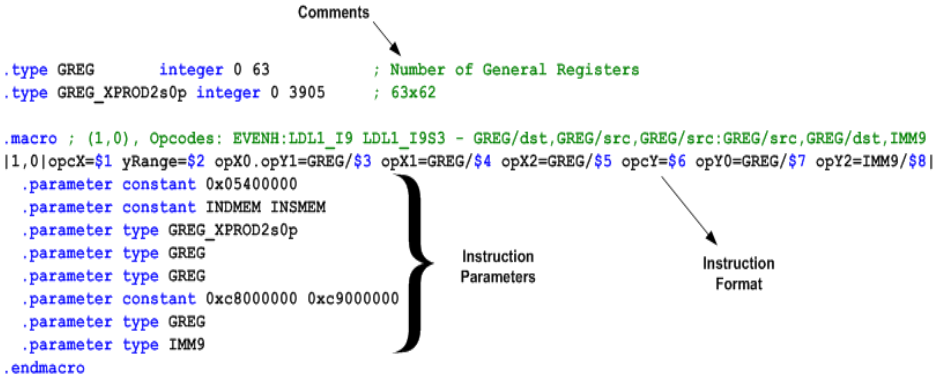

Fig. 8. Implementation of a single macro in the IL

Raising the level of abstraction from using pure FirePath assembly code (i.e. including Load/Store and Branch operations) to effectively semi-biases for FireDrill, as such, has had both positive and negative impact on the setup. A positive effect was the ability to create syntactically correct tests while hiding unnecessary details of the ISA. On the other hand, it has introduced a degree of dissimilarity between uGP generated tests and those finally run on FirePath.

The final IL created, consisted of 2635 macros. Each macro represented a legal pairing of operations which corresponded to a set of legal instructions to be used for constructing uGP tests.

3.3 Comparing Performance

The Evolutionary uGP experiments were contrasted against Random uGP experiments and experiments involving only the FireDrill constrained pseudo-random generator. Thus there were in total 3 types of experiments to be contrasted.

In order to fairly compare the experiments, it was useful to fix or limit some of the parameters. It was decided that the average number and average size of tests created by a standard Evolutionary uGP run would be used as a basis on which the other two experimental settings would be evaluated.

Every Evolutionary uGP run took 100 epochs to elapse. Usually the best individual would be found earlier than epoch 100, thus once it was found, the evaluated number

of instructions and tests, by that epoch, would be kept in order to later calculate the number and size of tests to be created during Random and FireDrill experiments. Thus once a series of 10 Evolutionary runs completed, the aforementioned indicators were extracted from the statistics kept and were used for setting up the Random uGP and FireDrill experiments.

Random uGP tests were produced by the uGP program after first disabling its evolutionary algorithm. In effect, every new test would be created by randomly picking instructions from the IL until the desired test size was achieved.

FireDrill tests were produced according to a bias file that allowed the generation of instructions like those of uGP. The only parameters that were passed to it were the number and size of tests to produce.

The usefulness of each of the 3 experimental results was determined by the quality of the tests produced in terms of code coverage achieved either by stand-alone tests or by a small regression suite.

4 Experimental Evaluation

4.1 Experimental Configuration

During the Evolutionary and Random experiments the main uGP program has been executed on a single Linux machine while the newly generated tests were simulated in parallel on 30 different machines.

In the Evolutionary uGP experimental runs, the GA was set to be guided by a 3-term Fitness function, assigning equal weight to each of the code coverage metrics used, while each experiment has been further divided into 3 configurations. These differed on how uGP sorts the entire population at the end of each epoch. The default primary sorting criterion was the aforementioned fitness value, but once 2 individuals in the population were equally effective, an additional sorting criterion was used. The default secondary sorting method was the individuals' "birth" i.e. the epoch on which they were first created (BIRTH). The tests were sorted with the newest and best test ranking as first in the population.

In the early stages of uGP evaluation it was noted that better coverage results, especially for branch coverage, were recorded as the size of the tests became larger. In order to test the validity of this observation and get most out of the tool during its evaluation, it was decided to assess the effect of test size on the quality of tests evolved. In order to do so, two alternative secondary sorting methods have been employed. Sorting individuals in descending order (DSC) of size was expected to increase the average size of tests in the long run. Doing the converse, i.e. sorting in ascending order (ASC) was expected to keep the size to the minimum without trading off coverage achieved.

4.2 Results – Single Best

In Table 1 the average achieved coverage in Expression (E), Branch (B) and Toggle (T) coverage on each of the Evolutionary uGP (Evo), Random uGP (Rnd) and FireDrill (FD) runs is presented for each of the 3 configurations (ASC, DSC, BIRTH).

Table 1. Average Best Test Coverage

		E (%)	**B (%)**	**T (%)**	**Best Size**	**Final Size**
DSC	Evo	88.10	62.10	94.40	3344	7208
	Rnd	88.00	92.77	95.00		3344
	FD	86.16	65.08	90.62		3344
ASC	Evo	87.30	63.10	94.30	3743	7128
	Rnd	88.00	93.17	95.52		3743
	FD	85.42	58.73	89.17		3743
BIRTH	Evo	87.20	67.50	94.10	4277	7522
	Rnd	88.00	94.48	95.00		4277
	FD	86.58	70.70	91.54		4277

The figures in Table 1 are an average over 10 experiments conducted per configuration. These averages are the achieved coverage by the single best test created at each of the 10 experiments.

As explained in Section 3.3, it was decided that for the generation of Rnd and FD tests the size will be determined by the Evo runs as follows. As soon as the best test was discovered on each of the 10 Evo runs, it would be noted and once all the 10 runs would conclude, the average size of those best tests would be fixed for Rnd and FD tests. This number was preferred over the total average of sizes on Evo runs, i.e. ~7000 instructions, because 100 epochs were only used to explore the full potential of Evo uGP runs. Thus the 'Best Size' column contains this average while the 'Final Size' column shows the average test size at the 100^{th} epoch of the Evo runs.

As can be seen, randomly created uGP tests have been able to achieve better results in terms of code coverage. This outcome differs from the reported experimental behaviour of uGP as given in [4-8]. Upon closer investigation it was found that the most prominent reason behind this behaviour is the lack of test code diversity in Evo uGP tests or conversely the high test code diversity in Rnd uGP tests. Also raising the level of abstraction in the test generation phase, as described in Section 3.2, has not allowed the GA mechanism to guide more effectively the test evolution.

At the same time, randomly created tests were better than those generated by FireDrill. It was known that FireDrill's default biasing was not optimally configured for targeting code coverage metrics and the results obtained pointed to this fact.

Despite the poor performance of Evo uGP in comparison to Rnd, it has managed to produce tests that achieved on average better coverage than the FD created ones consistently on all coverage metrics apart from Branch coverage.

4.3 Results – Cumulative

In terms of cumulative coverage, Table 2 shows the results of running a regression suite from tests developed by each of the experimental configurations.

Table 2. Cumulative Coverage

		E (%)	B (%)	T (%)	Total Tests	Regr. Suite Size	Instr. At Peak
DSC	Evo	87	98	97	29	16	117326
	Rnd	89	99	98	2448	20	66880
	FD	89	99	93	1021	39	130416
ASC	Evo	88	98	97	25	20	141809
	Rnd	89	99	98	2740	61	228323
	FD	89	99	98	887	72	269496
BIRTH	Evo	88	98	96	18	11	82895
	Rnd	89	99	98	2373	126	538902
	FD	89	99	98	1644	65	278005

The table contains the results of a single run rather than the average over 10 runs as is the case for Table 1. It is included here solely as an indication of the quality of tests created by each of the experimental settings.

The column ‘Total Tests’ shows the number of tests being considered for inclusion into a regression suite. The column ‘Regr. Suite Size’ shows at which test (after they were sorted in descending order of coverage) the maximum cumulative coverage is reached; these tests were selected for the regression suite. To make the comparison more complete the column ‘Instructions at Peak’, shows the total number of instructions in each regression suite.

Although Rnd regressions have produced equal, if not in some cases better cumulative coverage in comparison to the FD generated tests, these results also point to the fact that FireDrill is more likely to produce tests that collectively reach maximum attainable coverage than single best tests. This is because FireDrill was designed for regression test generation, in contrast to uGP. The comparison between the two, although not necessarily fair, is also done as an indicative measure of how well the two test generators perform on roles other than their original intent.

Another observation made is the low count of tests and instructions needed to reach comparably near-optimal coverage with Evo uGP generated tests. This is due to the quality of tests evolved as a result of the evolutionary search. The downside is the evolution and simulation time required; so although far fewer tests manage to cover almost the same amount of RTL code, the time it takes to evolve them is considerable. Thus it is worth using uGP to evolve tests that will be part of a regression suite that should contain few tests and with as low instruction count as possible.

4.4 Results – Coverage Progression

The following graph is provided as an indication of the progress of learning achieved while running the Evolutionary uGP algorithm.

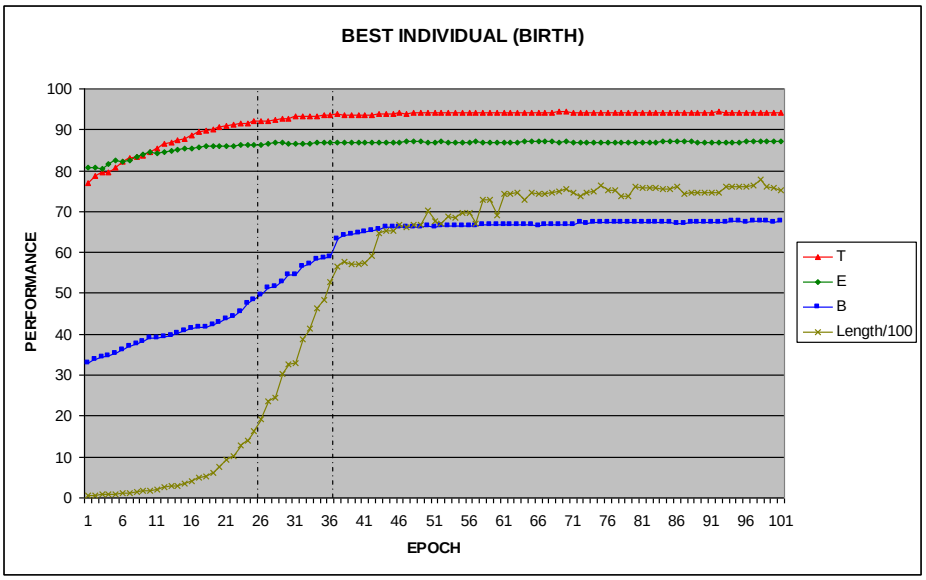

Fig. 9. Best Individuals progression (BIRTH)

The graph shows the progression of code coverage and test size for the best individual in the parent population at each of the 100 learning epochs the algorithm was run for, averaged over 10 runs.

One of the key points noted was that regardless of the secondary sorting order chosen (i.e. ASC, DSC, BIRTH) a convergence towards the best test to be evolved occurred around epochs 25-35. This behaviour is explained in the Discussion section. Due to similarity of graphs only the BIRTH configuration was chosen to be shown.

All three sorting order experiment results have shown that the size of tests after epoch 35 tails off while within additional 10-20 epochs it remains with some variation stable. In the DSC results we noticed a sharper increase of test size compared to the other two configurations but with no added coverage achieved. In fact, ASC runs have achieved better coverage results with smaller tests.

Another trend observed was the increased Expression coverage in relation to Branch coverage achieved. These results are counter-intuitive as one would expect normally branch to be exceeding expression coverage count. The reason for this behaviour was that the missing branch coverage is made of conditions that are not very complicated, or that depend on a single signal or register. Thus it was discovered that the remaining Expression coverage consisted of many different branches that could potentially be reached if simpler antecedent variable values were satisfied.

Concluding on the progression of coverage, the best of the Evo experiments was the BIRTH configuration as it was able to produce tests that on average achieved

better Branch coverage results while matching the Expression and Toggle coverage achieved by the other two configurations. The BIRTH results also reaffirmed the original impression on the correlation between test size and Branch coverage.

5 Discussion

The experiments described in this paper are the first to be produced on an industrial level design and verification flow created without contribution from the original authors of uGP; this alone points to the importance of the points made herewith.

During testing no bugs were found something that is consistent with the maturity of the DUV used. The full time implementation effort on the project is estimated to be 5 man months. The initial set of experimental results which informed the final experiments conducted (as presented in this work), took an extra month, while the data presented herewith took an additional month. The estimated time required to set up a similar environment for another verification project, within the company, is expected to be less than one month. This is due to two reasons. Firstly, all of the elements comprising the verification environment, as shown in Figure 7, are fully implemented. Future users of this setup would only need to construct the IL. A further reason is the experience gained in using uGP to conduct CDG on a given DUV. This initial study had quite a steep learning curve and future tool use will clearly benefit from this experience.

Each of the Evolutionary uGP runs required 72-96 hours to finish with the bottleneck being the creation of increasingly large offspring test programs during the execution of the main uGP program. During Rnd uGP experiments the time required for both creation and simulation/evaluation of tests required about 24 hours. For FireDrill the tests were created and simulated in batches of 20 tests thus the creation and simulation of tests would finish in 4-5 hours.

Judging from uGP's previous experimental results [4-8], it was expected that Evo uGP runs would produce better quality tests than Rnd uGP. The observed behaviour though, has given valuable insight into the reasons behind this discrepancy. Inspecting the uGP induced tests it was discovered they contained varying blocks of repeating code. In effect, improvement on learning stops after the 35^{th} epoch because diversity in the population decreases, despite test size increase. A local maximum is reached and then the probability of activation of genetic operators (especially mutation), that would be expected to increase diversity are reduced by the auto-adaptation algorithm. Although this correlates with the Genetic Algorithm discovering "useful" building blocks that one might argue are able to achieve good coverage, it also points to a lack of instruction diversity in the test. In fact it was noted that only a small fraction of the 2635 macros in the IL was represented in each test belonging to the final population (i.e. the existent test population at the 100^{th} learning epoch).

Additionally, FireDrill's stand alone performance in comparison to the Evo and Rnd uGP configurations has pointed to its selective effectiveness as a regression test generator and the need to improve some of its default biasing in increasing attained code coverage.

Given the results obtained it was concluded that μ and λ parameters and the GA selection strategy could be improved. Thus, a useful future improvement proposed

would be utilising a non-steady state, instead of the default steady state, population selection strategy. This means that a single best individual would be kept from epoch to epoch while the rest of the tests created would be variants of it via random mutations. This would potentially increase the diversity by simulating to a great extent the observed test generation behaviour of the Rnd uGP runs.

Finally, it was noted that using uGP did not require knowledge on evolutionary algorithms, thus increasing its usability. On the other hand, given the non-optimal behaviour observed, expertise would be required to understand and fix any issues.

6 Conclusions

MicroGP has proven to be a tool that can effectively guide a test generator such as FireDrill in producing compact single tests that achieve near optimal coverage.

On the other hand, quite unexpectedly, the increased effectiveness of the randomly induced uGP tests over the evolved ones pointed to the inability of the uGP algorithm to maintain diversity of tests during the learning phase. This weakness was made particularly evident in our work and this would in general be the case for LIW processors where there are several hundred thousand possible instructions.

MicroGP v.2.7 would benefit from the inclusion of alternative selection strategies and possibly more effective genetic operators that would enable the evolution of more diverse tests faster, while increasing the overall coverage attained by them.

Additionally, a more expressive syntax for creating more dynamic macros inside the IL would be useful. This would enable choosing calculated parameter values instead of a preset range of values, e.g. based on constraints that describe how current parameters depend on preceding ones. The end effect would be to increase the diversity and expressiveness, thus effectiveness, of tests produced by uGP.

Concluding on the overall experience, it needs to be said that the objective of verification is to eliminate as many bugs from the design as possible using the least resources. A useful first step towards this goal is maximising efficiently the coverage and test diversity achieved by the test generation methodology used. Random test generation has proved effective because, once a generation system has been set up, in the general case, many unexpected corner case bugs can be found, while in the specific case of this work, many coverage points can be hit with little intervention from the engineers. The generation and testing process is highly parallelisable and therefore achieves a lot of coverage in little elapsed time. In contrast, the evolutionary approach has a sequential bottleneck around the evaluation and breeding process which tends to dominate the elapsed time of the entire process.

Inevitably, the best answer is not going to be either random or evolutionary and will be a synthesis of both. Recent developments in uGP (i.e. uGP v.3 [19]) appear to be intended to address this.

7 Acknowledgements

The authors would like to thank Dr. G. Squillero, one of the original authors of uGP, for his constructive feedback and comments. Also A. Dawson, C. Randall, H. Tao and M. Desai, all engineers at Broadcom UK, for their invaluable support during the course of this project. Charalambos Ioannides is an EPSRC funded research engineer at the IDC in Systems, University of Bristol.

8 References

[1] ITRS, "International Technology Roadmap for Semiconductors, Design Chapter, 2008 Edition," 2008.

[2] G. Squillero, "MicroGP—An Evolutionary Assembly Program Generator," *Genetic Programming and Evolvable Machines*, vol. 6, 2005, pp. 247-263.

[3] A. Piziali, *Functional verification coverage measurement and analysis*, Berlin: Springer, 2007.

[4] F. Corno, G. Squillero, and M.S. Reorda, "Code Generation for Functional Validation of Pipelined Microprocessors," *Proceedings of the 8th IEEE European Test Workshop*, IEEE Computer Society, 2003, p. 113.

[5] F. Corno, G. Cumani, and G. Squillero, "Exploiting Auto-adaptive μGP for Highly Effective Test Programs Generation," *Evolvable Systems: From Biology to Hardware*, Trondheim (Norway): Springer Berlin / Heidelberg, 2003, pp. 262--273.

[6] F. Corno, G. Cumani, M.S. Reorda, and G. Squillero, "Fully Automatic Test Program Generation for Microprocessor Cores," *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, IEEE Computer Society, 2003, pp. 1006--1011.

[7] F. Corno, E. Sánchez, M.S. Reorda, and G. Squillero, "Automatic Test Program Generation: A Case Study," *IEEE Design & Test of Computers*, vol. 21, 2004, pp. 102-109.

[8] F. Corno, G. Cumani, M.S. Reorda, and G. Squillero, "Automatic test program generation for pipelined processors," *Proceedings of the 2003 ACM symposium on Applied computing*, Melbourne, Florida: ACM, 2003, pp. 736-740.

[9] Politecnico di Torino, "Research: MicroGP," Nov. 2007. Available from http://www.cad.polito.it/research/microgp.html, Accessed 2010-03-14

[10] S. Ur and Y. Yadin, "Micro architecture coverage directed generation of test programs," *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, New Orleans, Louisiana, United States: ACM, 1999, pp. 175-180.

[11] M. Bose, J. Shin, E.M. Rudnick, T. Dukes, and M. Abadir, "A genetic approach to automatic bias generation for biased random instruction generation," 2001, pp. 442-448.

[12] S. Tasiran, F. Fallah, D.G. Chinnery, S.J. Weber, and K. Keutzer, "A functional validation technique: Biased-random simulation guided by observability-based coverage," Institute of Electrical and Electronics Engineers Inc., 2001, pp. 82-88.

[13] D. Ashlock, *Evolutionary Computation for Modeling and Optimization*, Springer, 2005.

[14] J.R. Koza, "Evolution and co-evolution of computer programs to control independently-acting agents," *Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats*, Paris, France: MIT Press, 1990, pp. 366-375.

[15] P. Bernardi, K. Christou, M. Grosso, M.K. Michael, E. Sanchez, and M.S. Reorda, "Exploiting MOEA to automatically generate test programs for path-delay faults in

microprocessors," Heidelberg, D-69121, Germany: Springer Verlag, 2008, pp. 224-234.
[16] D. Ravotto, E. Sanchez, M. Schillaci, and G. Squillero, "An evolutionary methodology for test generation for peripheral cores via dynamic FSM extraction," Heidelberg, D-69121, Germany: Springer Verlag, 2008, pp. 214-223.
[17] S. Wilson, *The FirePath Processor Architecture Guide*, Broadcom Corporation, 2008.
[18] D. Robinson, *Aspect-Oriented Programming with the e Verification Language: A Pragmatic Guide for Testbench Developers*, Morgan Kaufmann, 2007.
[19] E. Sanchez, M. Schillaci, and G. Squillero, *Evolutionary Optimization: the µGP toolkit*, Springer, 2010.