Fine-Grained Timing Using Genetic Programming

David R. White[1], Juan M.E. Tapiador[1],
Julio Cesar Hernandez-Castro[2], and John A. Clark[1]

[1] Dept. of Computer Science, University of York, York YO10 5DD, UK
{drw,jet,jac}@cs.york.ac.uk
[2] School of Computing, University of Portsmouth, Buckingham Building,
Lion Terrace, Portsmouth PO1 3HE, UK
Julio.Hernandez-Castro@port.ac.uk

Abstract. In previous work, we have demonstrated that it is possible to use Genetic Programming to minimise the resource consumption of software, such as its power consumption or execution time. In this paper, we investigate the extent to which Genetic Programming can be used to gain fine-grained control over software timing. We introduce the ideas behind our work, and carry out experimentation to find that Genetic Programming is indeed able to produce software with unusual and desirable timing properties, where it is not obvious how a manual approach could replicate such results. In general, we discover that Genetic Programming is most effective in controlling statistical properties of software rather than precise control over its timing for individual inputs. This control may find useful application in cryptography and embedded systems.

1   Introduction

In previous work, we have combined Genetic Programming (GP), Multi-objective Optimisation (MOO) and simulation of hardware platforms to produce software with low power consumption [1] and reduced execution time [2]. We found that GP was indeed able to meet both functional and non-functional requirements, and also provide trade-offs between the two. Here we propose the notion of fine-grained resource control. For example, can we control the execution time of a program p on input x, $T(p, x)$, such that T can be an arbitrary function? Such control might allow us to solve some problems more efficiently than with functional computation alone, and to create software with useful security properties.

In this paper, we investigate the types of control that GP can achieve, and the effectiveness of the search algorithm in achieving these goals. We do not attempt to demonstrate that our solutions are the best, or that our parameter settings are optimal: only that GP has the potential to finely control timing behaviour in a way not previously considered. The results of single experimental runs are presented as proof of concept. We discuss example potential applications of these results to the domain of cryptography. Whilst we address timing properties, alternatives include power consumption and memory usage.

2   Evolving Code with Specific Time Complexity

In this section, we demonstrate the difficulty of trying to manually control the
low-level timing properties of software, as we attempt to create code that has
a very specific time complexity relationship. This demonstrates the potential
superiority of GP against manual design alone as a tool for such tasks.

We evolve individuals in the ECJ 19 Toolkit [3] and evaluate them by writing
them out as C code, compiling them with a test harness using a cross-compiler,
and executing them within the M5 Simulator [4]. The simulator is targeted for an
Alpha architecture, using its default parameters. A trace file is produced, which is
parsed by ECJ to measure the number of cycles used by the evolved code in each
test case. Where handwritten solutions are examined in the following sections,
the C source code has been manually written, and the rest of the evaluation
method remains unchanged to ensure a fair comparison with evolved solutions.

2.1   Designing Linear Complexity

It is second nature to think about a program in terms of complexity: quadratic is
good, linear is better, exponential is undesirable. For a moment, let us focus only
on the complexity of a program rather than its purpose. Consider a program
A that has linear complexity with respect to quantity n. Quantity n may be
problem size, in which case we assume the conventional notion of the complexity
of the program. If n an input, we have a program that has a linear relationship
between its *input values* and its resource consumption. This is related to the
theory of pseudo-polynomial time algorithms.

If we wish to construct code that may have this behaviour, we find that it is
actually not as straightforward as one might assume, because the complex inter-
action of hardware and software can easily lead to outliers in the relationship.
For example, if we wish to construct a solution such that $T(p, k) = mk + c$, we
can suggest such a program without regard to its functionality as thus:

```
float tmp = 0;
while (tmp < k)
    tmp++;
```

We have used `float` rather that `int` to maintain generality in the following sec-
tions, and because floating point operations have interesting timing properties.
We now test this program with $k = 0 \ldots 99$. A test program provides these in-
puts, then parses a trace file to measure cycle usage for each call. Rather than
implement this as a function, we have actually used a C `#define` macro to force
the compiler to inline this code. This improves efficiency, and allows us to ma-
nipulate code embedded within a program rather than as an external function,
interacting fully with the machine context of the surrounding code.

Figure 1(a) gives a graph of the results. Note that we do not experience
a perfect relationship, due to the interaction between the test program and
machine state. The primary cause of the outliers is data cache misses for the
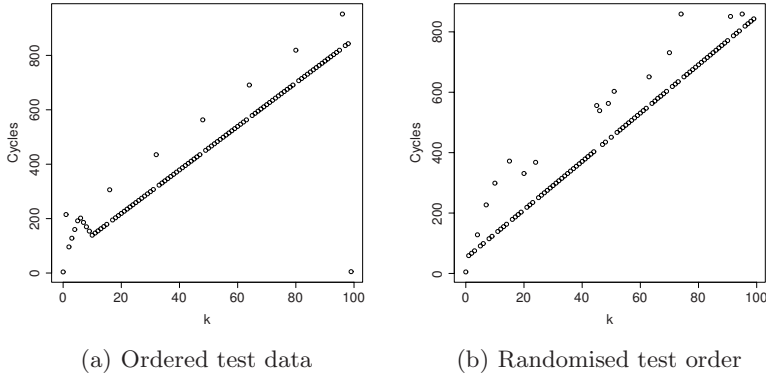input values, such that the comparison operation causes a long delay prior to

(a) Ordered test data   (b) Randomised test order

Fig. 1. Test results for handwritten linear solution

the next instruction. If we randomise the order of the inputs, we can measure the relationship in a more robust manner. The result is given in Figure 1(b).

Here we have even larger outliers, but they are spread throughout the input range rather than concentrated at the beginning. We can quantify the relationship by estimating Pearson's correlation coefficient for this sample:

$$r(k,T) = \frac{\sum_{i=1}^{n}(k_i - \bar{k})(T_i - \bar{T})}{\sqrt{\sum_{i=1}^{n}(k_i - \bar{k})^2}\sqrt{\sum_{i=1}^{n}(T_i - \bar{T})^2}} \tag{1}$$

Where k_i is the ith test case and T_i is the time taken to execute a code fragment p on this input. A perfect positive correlation would be 1, a negative correlation -1 and 0 for no correlation. The correlation coefficient for this manual attempt at linearity is 0.978, and by no means a perfect correlation due to the subtle interaction with the program's context. An interesting question, therefore, is: *can evolution find a better solution?*

2.2   Experiment A: Linear Behaviour

Experimental parameters are given in Table 1, chosen arbitrarily, and no tuning has been attempted. The intention is to demonstrate what is possible with GP, and not the most efficient way to achieve these results. The maximum number of instructions is limited to a similar value to that used by the handwritten solution, preventing GP from using larger solutions to mask the "noise" of program context. The randomised order of tests is varied at each generation, to avoid overfitting. "Increment tmp" increases the temporary variable by 1, whereas "update tmp" assigns a new value. `FixedLoop(n)` will loop for n iterations.

Genetic Programming is indeed able to locate a better solution, with a correlation coefficient of 0.993. A graph of this function's relationship and the code of the individual evolved is given in Figure 2. The key differences between the evolved and handwritten solutions are:

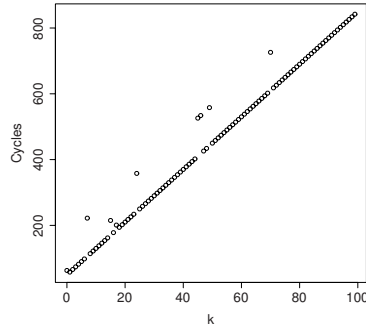**Table 1.** Experiment A: Settings to evolve linear time behaviour

Objective	Find a program $p_*(k)$ such that $\Theta(f) = mk + c$
Terminal set	k, tmp
Function set	FixedLoop, $\leq$, increment tmp, +, $*$, $-$, $/$, if, sequence, skip, update tmp
Fitness cases	$k \in \{i\|0 \leq i < 100\}$
Fitness function	$r(k, T)$
Parameters	Initial tree depth = 3, generations = 10, population size = 20, prob(xo) = 0.9, prob(mutation) = 0.1

```
if (k) {
    for (c=0;c<k;c++) {
    };
} else {
    tmp++;
    tmp++;
    tmp++;
}
```


(a) Source code   (b) Time against input

Fig. 2. Evolved linear solution

- When $k = 0$, work will still be performed in the evolved version, namely testing for this case, and moving through further branches.
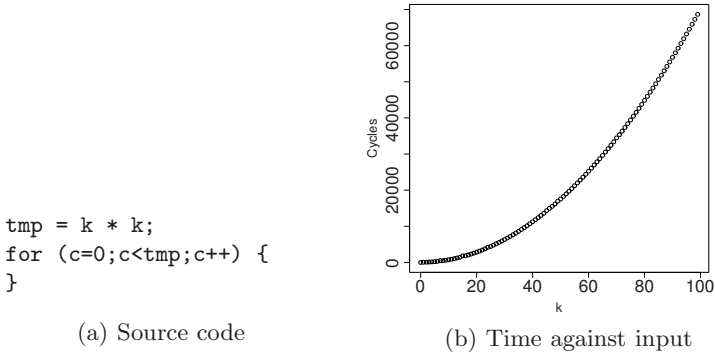- When $k > 0$, an additional if statement evaluation will be performed.

If we examine the trace output for this evolved solution, we see that it includes exactly the same instructions executed in the manual solution, and adds some to either side of the loop. This effectively "smooths" the response (time taken) by suitably padding the instruction pipeline, as well as increasing the time taken for $k = 0$, i.e. setting a larger value for the intercept.

Note that we can now (approximately) calculate $f(k)$ by running the program and observing the time taken to execute, where in this case $f(k) = mk + c$.

2.3   Experiment B: Quadratic Behaviour

What if we wish instead to create a nonlinear relationship such as a quadratic curve with no linear term? We may try a handwritten solution as in Figure 3.

This solution takes the general approach of calculating $f(k)$ prior to repeating the strategy of looping to perform a multiple of a minimal unit of work

```
tmp = k * k;
for (c=0;c<tmp;c++) {
}
```

(a) Source code

(b) Time against input

Fig. 3. Test results for handwritten quadratic solution

(increasing c). This relationship appears to be a great improvement on the manual attempt at a linear solution, but we must quantify the relationship to be sure. We can estimate the derivative of the evolved function as:

$$f(k+1) - f(k) = m(2k+1) \tag{2}$$

Thus we can measure the correlation between the differences of successive timings and k in order to evaluate the fit of the quadratic relationship. For the handwritten solution above, we find a correlation of 0.962. Can we improve on this using evolution? We first used the correlation measure as a fitness function, and after some manual experimentation and analysis it became apparent that rewarding a certain amount of first-order correlation is also desirable in order to guide the search, and hence the fitness function was modified thus:

$$F(p) = w_0 \cdot r(k, T(p,k)) + w_1 \cdot r(k, T(p, k+1) - T(p,k)) \tag{3}$$

A simple selection of 0.2 and 0.8 for w_0 and w_1 respectively was sufficient to guide the search to a successful solution, found using the same parameter settings as given in in Table 1. The best individual evolved had a correlation of 0.967 between the input and derivative, a very modest improvement over the manually written version. The individual was:

```
for (c=0;c<(k * k );c++)
    tmp = tmp;
```

3   Time as a Functional Output

In the preceding section, we demonstrated that it is possible to search for programs that have simple relationships between their numerical input and total absolute execution time, and that exhibit those relationships more accurately than "obvious" handwritten alternatives.

Consider now a more complex relationship such as a Boolean function. Given two Boolean inputs a and b, can we evolve a program p with execution time $T(p, a, b) = f(a, b)$, where f could be a Boolean OR? For this to be literally true, the program's execution would have to take either 0 or 1 cycles, which is unrealistic as even a simple if will take more than a single cycle. Hence, we must find an *interpretation* of the timing output, denoted $I(T(p, a, b))$. This is an idea with much generality, and is illustrated in Figure 4.
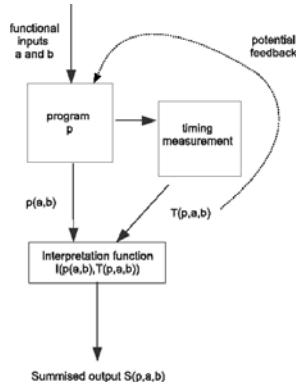

Fig. 4. Visualising the use of timing in calculation

A very simple interpretation that can be employed here is to measure the number of cycles the machine has executed and then take modulo 2 of this number. Thus the lowest "bit" of the execution time is used as output, and there are two output states: an even and odd number of cycles consumed. An interpretation such as this is arbitrary, and the idea can be generalised: perhaps interpretations can be cooperatively coevolved? The most interesting question here is: do such code fragments even exist, given the context of the test program within which they will be used?

It is not obvious how to go about manually constructing a solution, but we can imagine two ways in which it may be achieved: through something we could describe as either *implicit* or *explicit* time variation. In the former, we hope to rely purely on the low-level mechanics of the processor and memory subsystem to provide variation, such as floating point operations that consume a variable number of cycles. In the latter, we rely on some logical test to choose a path through the code. In practice, the latter cannot succeed without the former in order to "iron out" variation as in the handwritten solution from Section 2.2.

3.1   Experiment C: Timing OR Function

In this experiment, we give as input the four possible combinations of a pair of Boolean-valued floats and try to evolve a program that outputs their OR on the lowest bit of the cycle count. Initial experimentation revealed the following:

 – the test cases could be passed by code that did not even evaluate the inputs.
 – degenerate behaviours taking a single cycle ("always 1") were common.
 – the ordering of test cases affected the performance of the search.

The code was therefore subsequently tested with the four possible inputs over four repetitions, and a randomised ordering of these 16 test cases used. In order to ensure that both inputs were evaluated at least once in any solution, the fitness function was specified as follows:

$$f(p) = w_1 \cdot c(p, a) + w_2 \cdot c(p, b) + w_3 \cdot (n - \sum_1^n |(T(p, a_n, b_n)\%2) - (a_n \vee b_n)|) \quad (4)$$

Where $c(p, a)$ is 1 if program p reads variable a, and 0 otherwise. The weightings w_1, w_2 and w_3 were 0.25, 0.25 and 0.5 respectively (an arbitrary choice). The parameters to the experiment are given in Table 2. Note that we added the constants 1 and Float_MAX ($3.40282e38$) to the terminal set. GP was indeed able to provide such a solution:

```
if (( 3.40282e+038f - b )) {
    for (c=0;c<(b + a );c++) {
        tmp++;
    };
} else {
    tmp =  3.40282e+038f;
}
```

This code passed all 16 test cases; it is quite straightforward to see how this might work. However, running the code again with a different test case order resulted in the code failing two test cases. This is because T is a function not only of p, a and b, but also of the machine context ϕ. We are trying to evolve code such that $T(p, a, b, \phi) \% 2$ is equivalent to $a \vee b$. The context is the machine state: stack

Table 2. Experiment C: Settings to evolve a Boolean OR function using time as an output channel

Objective	Find a program $p_*(a, b)$ such that $T(p, a, b)\%2 = OR(a, b)$
Terminal set	a, b, tmp, 1, Float_MAX
Function set	FixedLoop, $\leq$, increment tmp, +, *, −, /, if, sequence, skip, update tmp
Fitness cases	$a, b \in \{\{0, 0\}, \{0, 1\}, \{1, 0\}, \{1, 1\}\}$ (four reps, order randomised)
Fitness function	$f(p) = w_1 \cdot c(p, a) + w_2 \cdot c(p, b) + w_3 \cdot (n - \sum_{i=1}^n |T(p, a_i, b_i) - (a_i \vee b_i)|)$
Parameters	Initial tree depth = 3, generations = 20, population size = 100, prob(xo) = 0.9, prob(mutation) = 0.1

contents, the instruction pipeline, cache etc. This reliance on context is both a difficulty, where we wish to eliminate its effects, and also a useful resource, in the case where we want to exploit the machine state to achieve certain timing properties. This context caused problems for the manual solution in Section 2.1.

If we wish to implement this OR gate robustly, we must test on all $4! = 24$ input sequences. Even then, however, we cannot be sure that the context of the test program is not being exploited, such that if we wish to use the code in a new program we would have to again evolve and exhaustively test a new solution. Thus any such "absolute time manipulation" must be done in situ.

We did indeed attempt to evolve a program that is robust to input ordering, but failed to successfully do so even when testing each bit of timing output as a potential interpretation. Every run provided an individual with a few failed test cases out of 96. To achieve such control over absolute cycle usage it may be necessary to construct a function set exhibiting a variety of timing behaviours.

4   Timing Avalanche Criterion

In previous work [1], we have demonstrated that it is possible to evolve low-power pseudorandom number generators (PRNGs) using GP and simulation. As a measure of the randomness of the PRNG, we used the Strict Avalanche Criterion (SAC) [5], a cryptographic measure that estimates the nonlinearity of a function. SAC analyses the expected distance between outputs given a single bit flip in the input. SAC is an efficient measure of randomness that generalises well to other statistical qualities. Note that in this experiment we have used a single 32-bit input rather than 8 as in previous work, for the sake of simplicity.

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 $p(a)$ and $p(a')$ 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 program p is given by:

$$SAC(p) = \sum_{i=0}^{n} \frac{(C_i - E_i)^2}{E_i} \tag{5}$$

C_i is the counted frequency of i bit flip events, and E_i the expected number. In this experiment, *we take the radical step of applying the avalanche criterion to its timing behaviour*, $T(a, p, \sigma)$. We refer to this as the Timing Avalanche Criterion (TAC). This is a fascinating concept, which we suggest has never been considered due to the lack of any manual method capable of implementing it. It may enable us to evolve programs resistant to side-channel cryptanalysis, a major problem in designing secure algorithms. Kelsey [6] notes "It is probably not possible to protect against side-channel attacks in the design of algorithms."

Kocher et al. [7] were amongst the first to demonstrate that cryptographic primitives provably secure in the mathematical domain can become exposed to

unseen vulnerabilities when implemented in a physical system. By monitoring the timing properties or power consumption [8] of a system, it is possible to deduce information about the state of executing software and compromise its security. A wide variety of attacks exist, from simple timing of operations to statistical analysis of power traces.

Counteracting such attacks is difficult: one method is to design code to consume the same number of cycles regardless of both the data input or output and the state of the system. For example, this can be achieved through the insertion of NOP instructions through the implementation. This defence is usually vulnerable to other forms of attack, such as detecting NOPs through other means, and a more robust method of defence is desirable.

4.1   Experiment D: Simple TAC

In this section, we attempt to evolve an expression that has a good TAC measure across the lowest 10 bits of its cycle count, which allows for up to 1023 cycles, sufficient for the evaluation of small expressions. Does such an expression even exist, given the context of a simple test loop that repeatedly uses the code with inputs separated by a single bit flip? It is not clear to us how we would proceed in designing such an expression, but can GP design one for us?

Table 3. Experiment E: Settings to evolve an expression with a good Timing Avalanche Criterion measurement

Objective	Find a program $p_*(a)$ such that $HD(T(p, a, \sigma), T(p, a', \sigma)) \approx B(n, \frac{1}{2})$ where $HD(a, a') = 1$.
Terminal set	a, Integer ERCs
Function set	If, $<$, LogicalShiftLeft (LSL), LSR, MULT, SUM, AND, NOT, OR, XOR
Fitness cases	a, a' where $HD(a, a') = 1$, sample size 4000
Fitness function	$\sum_{i=0}^{n} \frac{(C_i - E_i)^2}{E_i}$ over $n = 10$ bits of timing measurement, where C_i is the resulting frequency of i bits flipping and E_i is the expected frequency.
Parameters	Initial tree depth = 3, generations = 25, population size = 100, prob(xo) = 0.9, prob(mutation) = 0.1

We use a sample size of 4000, which is generous according to our previous work, and the same function set as we have previous employed. The experiment is summarised in Table 3. The best individual was subsequently tested over a sample size of 10000, which gave a good TAC measure of 0.0228. The p value of this result, effectively giving the probability that this sample is drawn from the ideal Binomial distribution, is 1.00 (to 3 s.f.). The distribution of bit flips is give in Figure 5. The individual is also reasonably small:

```
tmp = (((( a > (3594493887 u)) ^ ((a * (1408948682 u) ) > ((a > (3594493887 u
      )) ^ (a * (609711807 u) ) )) ) ? (((a * (1408948682 u) ) >
      ((2302909662 u)) ^ a )) << (((a > (a * (609711807 u) )) ^ (a *
      (609711807 u) ) ) % 32 u) ) : ((((2390510013 u) * (((a > (3594493887 u))
      ^ ((((a * (1408948682 u) ) > (((a * (1408948682 u) ) > ((2302909662 u)
      ^ a )) << (((a > (a * (609711807 u) )) ^ (a * (609711807 u) ) ) % 32 u
      ) )) * (2540811676 u) ) * (((a * (1408948682 u) ) & ((a > (3594493887 u
      )) ^ (a * (609711807 u) ) )) + a ) ) ) & (2490185230 u)) ) ^ (a *
      (609711807 u) ) ) & (1135240832 u)))
```

4.2 Experiment E: TAC and SAC

We now ask: is it possible to produce programs that perform two tasks at once,
with both desirable functional and timing outputs? We attempt to produce code
that has both good functional SAC (that is, it is a good random number genera-
tor) and also good TAC (it is resistant to side-channel analysis). Such a solution
provides evidence that GP can evolve primitives resistant to side-channel analy-
sis. It also suggests the feasibility of combining timing and functional properties
of the software to improve the random number generator, or even to feedback
the timing measure as input to improve the nonlinearity of the code over a se-
ries of evaluations, assuming we have the capability to measure time from the
code.

The fitness function used was an even weighted combination of SAC and TAC,
i.e. we assumed that the two objectives were not necessarily conflicting and that
consequently Multi-objective Optimisation was not required. With a larger run
of 25 generations, one of the best individuals produced was:

```
tmp = (((( a * (1408948682 u) ) ^ ((((((((1106785805 u) * a ) >>
      (((((1167812458 u) + (2754164240 u) ) & (~ a)) + ((2240102872 u) > a) )
      % 32 u) ) * ((((((1106785805 u) * a ) >> ((a ^ a ) % 32 u) ) >> ((a ^
      a ) % 32 u) ) * ((((((2390510013 u) * a ) >> (((1106785805 u) * a ) %
      32 u) ) * (((1106785805 u) * a ) > (2390510013 u))) ^ (~ a) ) * a ) )
      ^ (~ ((((1106785805 u) * a ) > (3594493887 u)) ^ a )) ) ) ^
      ((1106785805 u) * (~ ((((1106785805 u) * a ) > (3594493887 u)) ^ a )) )
      ) >> (a % 32 u) ) * (1408948682 u) ) * (((2390510013 u) * a ) >> ((
      a ^ (((1876056559 u) + (3922502476 u) ) < ((1106785805 u) * a ) ) ) %
      32 u) ) * (((2390510013 u) * a ) >> ((a ^ ((1106785805 u) * a ) ) % 32 u
      ) ) ) ^ (~ a) ) * a ) ) ) * (1408948682 u) ) ^ (((((2390510013 u) * a
      ) >> ((((((((a ^ (((2641736152 u) * (1408948682 u) ) ^ ((3594493887 u)
      * a ) ) ) ) >> ((a ^ a ) % 32 u) ) * ((((((2390510013 u) * a ) >>
      (((1106785805 u) * a ) % 32 u) ) * (((1106785805 u) * a ) > (2390510013
      u)) ) * a ) >> ((a ^ ((1106785805 u) * a ) ) % 32 u) ) ) ^ (~ a) ) >>
      ((a * (609711807 u) ) % 32 u) ) ^ (((((1670273053 u) | a ) & (
      ((3825661740 u) + (3105575741 u) )) ) ^ ((((((2390510013 u) * a ) >>
      (((1106785805 u) * a ) % 32 u) ) * (((1106785805 u) * a ) > (2390510013
      u)) ) ^ (~ a) ) * a ) ^ (~ ((((1106785805 u) * a ) > (3594493887 u))
      ^ a )) ) ) ^ ((1106785805 u) * a ) ) >> (a % 32 u) ) % 32 u) ) *
      (((2390510013 u) * a ) >> ((a ^ ((1106785805 u) * a ) ) % 32 u) ) ) ^
      (~ a) ) )
```

Comparing this to past results, there is a wider use of constants. Constants
require memory access, introducing variation into the timing of the individual.
The TAC and SAC distributions over a sample of 10000 are given in Figure 6.
This individual had a SAC of 0.0189 and a TAC of 0.0399, equivalent to a p
value of 1.00 (to 3 s.f.). These are excellent values, achieved surprisingly easily.
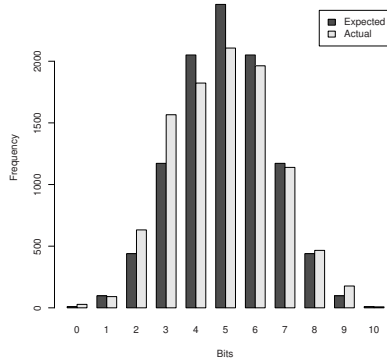
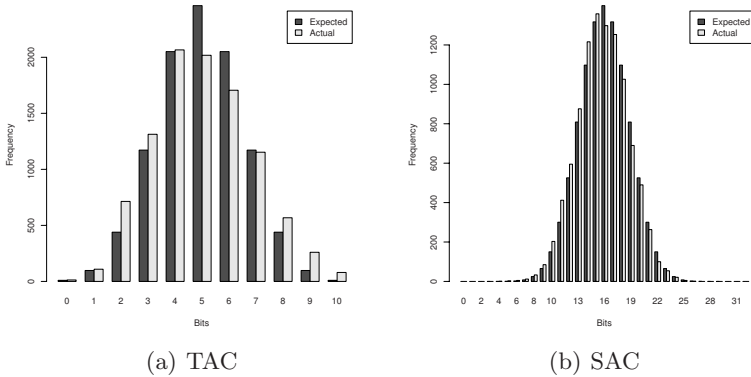**Fig. 5.** Bit flip distributions for simple TAC

(a) TAC

(b) SAC

Fig. 6. Bit flip distributions for best individual

5 Conclusion

In this paper we have introduced several ideas regarding the application of Genetic Programming for fine-grained control over resource consumption. GP was shown to be most useful in controlling the behaviour of code over a series of evaluations, rather than the absolute value of a single evaluation. This opens up an avenue of further exploration in the concept of "doing two things at once", to liberate software from its role as an object of abstract calculation into a process that interacts with the host hardware platform and its execution context. At such a complex level of interaction, methods such as Genetic Programming may be essential to achieve desired behaviour.

Example applications can be found in cryptography, both in defending against side-channel attacks and exploiting side-channel properties. By exploiting covert

timing channels [9], programs such as that evolved in Section 4.2 can be used to transfer information such that a *program is a key*, and the timing output of that program on a given input reveals the message. Only a party with the same code and platform can decode it. Another scenario is using such a program as a keystream generator, incorporating the timing properties of the software by (for example) XORing the functional output with the time taken.

References

1. White, D.R.: Searching for resource-efficient programs: Low-power pseudorandom number generators. In: GECCO 2008: Proceedings of the 10th annual conference on Genetic and evolutionary computation (2008)
2. Arcuri, A., White, D.R., Clark, J., Yao, X.: Multi-objective improvement of software using co-evolution and smart seeding. In: International Conference on Simulated Evolution And Learning (SEAL), pp. 61–70 (2008)
3. ECJ: Evolutionary computation in Java,
   `http://www.cs.gmu.edu/~eclab/projects/ecj/`
4. Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., Reinhardt, S.K.: The M5 simulator: Modeling networked systems. IEEE Micro 26(4), 52–60 (2006)
5. Webster, A.F., Tavares, S.E.: On the design of s-boxes. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 523–534. Springer, Heidelberg (1986)
6. Kelsey, J., Schneier, B., Ferguson, N.: Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In: Heys, H.M., Adams, C.M. (eds.) SAC 1999. LNCS, vol. 1758, pp. 13–33. Springer, Heidelberg (2000)
7. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
8. Kocher, P., E, J.J., Jun, B.: Differential power analysis, pp. 388–397. Springer, Heidelberg (1999)
9. Kemmerer, R.A.: A practical approach to identifying storage and timing channels: Twenty years later. In: Computer Security Applications Conference, p. 109 (2002)