Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 266 (2010) 49–61
www.elsevier.com/locate/entcs

Test Case Generation for Adequacy of Floating-point to Fixed-point Conversion

Tuan-Hung Pham[1], Anh-Hoang Truong[2]

Department of Software Engineering
College of Technology, Vietnam National University, Hanoi
144 Xuan Thuy, Hanoi, Vietnam

Wei-Ngan Chin[3]

Department of Computer Science
School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117543

Takenobu Aoshima[4]

Panasonic R&D Center Vietnam
No. 1 Lang-Hoa Lac, Hanoi, Vietnam

Abstract

Porting an application written for personal computer to embedded devices requires conversion of floating-point numbers and operations into fixed-point ones. Testing the conversion hence requires the latter be as close as possible to the former. The closeness is orthogonal to code coverage and requires different strategies to generate a test suite that reveals the gap between the two functions. We introduce a new test adequacy criterion and propose several metrics to quantify the closeness of two functions. After that we propose a method to generate a better test suite from a given one for the test adequacy criteria. We also show experimental results on some well-known mathematical functions.

Keywords: Quality assurance, Test cases generation, Fixed-point conversion.

1 Introduction

Software testing is an essential phase in any software development project to guarantee the quality of the software product. During the testing process, test suites,

[1] Email: phamtuanhung@vnu.edu.vn
[2] Email: hoangta@vnu.edu.vn
[3] Email: chinwn@comp.nus.edu.sg
[4] Email: aoshima.takenobu@jp.panasonic.com

doi:10.1016/j.entcs.2010.08.048

which contain collections of test cases, are used to test programs. In a test suite, each test case consists of, among others, an input and an expected result, which will be compared with the actual returned result to determine how good the input program is in respect of the test case. We usually consider a good test suite is the one that has small size but has high code coverage, meaning that it not only consumes little memory for running but also can reveal as many behaviors of input programs as possible.

Because most embedded microprocessors do not have floating-point units (FPU) for computing arithmetic operations with high precision, developing software for embedded devices and mobile phones usually requires a lot of work to convert floating-point numbers and operations into fixed-point ones. In addition, sophisticated algorithms are usually modeled on personal computer using floating-point computation before they are ported to embedded systems. The porting process in many cases keeps the original algorithm of the floating-point function and increases the preciseness of the fixed-point arithmetic operations by some techniques, for example scaling up the dividends before divisions and scaling back the result. In some other cases, it is obligatory to make a new implementation of the algorithm to meet the requirements. For instance, to avoid slow arithmetic operations like multiplications and divisions in particular cases, programmers may exploit shift operations.

The main problems with fixed-point operations are precision loss and overflow caused by rounding and truncating. Consequently, one main goal of converting a floating-point function into fixed-point one is to minimize the precision loss. In other words, the two functions should be as *close* as possible. Although the floating-to-fixed-point conversion has been theoretically studied [17,9,22] as well as partially implemented in several tools like MATLAB, determining how close the two corresponding functions are after the conversion process is still an open issue. Because the number of input space's dimensions of the functions may be very large (for example, fixed-point audio or video processing functions may have thousands of parameters), traditional testing methods, which would require executing the two functions with *all* possible input values, become practically infeasible.

The current practice of testing the fixed-point conversion is to execute the fixed-point version on the emulator or actual devices and then observe the results. In some application domains where observing the results as above is not enough or impossible to guarantee the quality of the software, one has to compare two versions (floating-point and fixed-point) of each computational function with the random input or sample test data and check the results. If the difference between the two results of an input is above a predetermined threshold, the fixed-point function is refined. In this case the test case is said to be *good* because it shows a critical point where the fixed-point function is not close enough to the original function. The code refinement process for the conversion is repeated until all the test cases in the test suite are not good, which means at that time the refined fixed-point function is acceptable.

Note that at this moment we can only say that the fixed-point version is adequate

with respect to the chosen test suite. There may exist a test case that is not in the test suite and is good. The quality assurance process is hence incomplete. In order to improve the quality, programmers or testers have to produce more test cases and test suites to get a certain level of confidence about the quality of the fixed-point function. But this task is not easy and requires both human effort and computing power. Simply generating random test cases is of course not a good strategy. It is better to have a method to generate the best test cases or at least a better test case from a given one.

Treating the two types of functions as black-boxes, we introduce several metrics to assess two test cases of fixed-point conversions. From these metrics we present a novel test adequacy criterion for test case generation process to stop when a good test case is found. The metrics are then used to measure the adequacy of a fixed-point function compared to its floating-point version. From our point of view, our testing approach is specification-based where floating-point function can be viewed as a specification for the fixed-point function. In addition, a big difference between the two functions on a given input indicates a possible overflow problem in fixed-point function. To generate test cases that fulfill the adequacy criterion, we apply mathematical results on local optimization to generate a more efficient test suite from a given one and on global optimization to find the best test cases. The better test suite allows us to evaluate the quality of the fixed-point function quickly while the best test cases guarantee a certain level of quality of the conversion. We implemented a prototype tool using Genetic Algorithm [6,19] and Tabu Search [5] and tested it on several well-known mathematical functions.

The rest of the paper is organized as follows. Section 2 introduces several metrics for assessing the closeness of two functions. Based on these metrics, Section 3 shows how to generate a better test suite from a given one, and Section 4 points out a method to find the best test cases. We show some experimental results on several mathematical functions in Section 5. Related work is discussed in Section 6. Section 7 concludes.

2 Closeness metrics and an adequacy criterion

Before presenting our novel test adequacy criterion and defining several metrics to measure the closeness between two functions, we formalize several relevant notions. Let $fl : \mathbb{L}^n \to \mathbb{L}$ be the original n-ary floating-point function and $fi : \mathbb{I}^n \to \mathbb{I}$ be its corresponding fixed-point version where $\mathbb{L}$ is the set of floating-point numbers and $\mathbb{I}$ is the set of fixed-point numbers. These sets are finite, and $\mathbb{I} \subset \mathbb{L} \subset \mathbb{R}$ where $\mathbb{R}$ is the usual set of real numbers.

2.1 Closeness metrics

The key notion for closeness metrics is precision loss function l defined in Definition 2.1. Note that since $fl : \mathbb{L}^n \to \mathbb{L}$, $fi : \mathbb{I}^n \to \mathbb{I}$, and $\mathbb{I} \subset \mathbb{L}$, we can consider $l : \mathbb{I}^n \to \mathbb{L}^+$ where $\mathbb{L}^+$ is the set of non-negative floating-point numbers. Our aim is to analyze the differences between fl and fi over the domain $\mathbb{I}$ of fixed-point function fi. For

brevity, we do not discuss in detail how to perform typecasting from fixed-point to floating-point and vice versa to make l mathematically valid.

Definition 2.1 [Precision loss function]

$$l(x) = abs(fl(x) - fi(x))$$

where $abs(x)$ is the absolute value of x.

From the precision loss definition, we define two simple but useful metrics. The first metric is the maximum loss of the two functions.

Definition 2.2 [Max metric]

$$FM(fl, fi) = \max_{x \in \mathbb{I}^n} \{l(x)\}$$

The second metric is defined over a given test suite S and a threshold $T \in \mathbb{I}^+$. It is used to generate an optimal test suite from a given one in Section 3.

Definition 2.3 [Threshold metric]

$$FM(fl, fi, S, T) = [\{x \mid x \in S,\ l(x) > T\}]/[S]$$

Here [.] denotes the cardinality of a set. The threshold metric is the percentage of the number of good test cases in a given test suite. If this value is smaller than, say 1%, we say that the function is good up to 99% or good with probability of 0.99. In certain cases a threshold value like this is quite acceptable, especially for functions used in compressing audio signals. Otherwise, one is free to require this threshold value must be zero so that we have some preliminary assurances about the quality of the conversion with respect to the threshold T.

The two metrics define different quality criteria for the closeness of two functions. Sometimes it is unacceptable to have the Max metric to be above some values. In other cases, having only few glitches (the Max metric) is acceptable as long as the Threshold metric is good enough for all test cases.

2.2 Adequacy criterion

Our adequacy criterion is error-based [24,11] in the sense that we need to show that testing should demonstrate that a fixed-point version does not deviate from its floating-point version. In other words, the floating-point version can be viewed as the specification that the fixed-point version should respect. Based on the Max metric and Threshold metric defined in the previous section, we have two corresponding notions of adequacy.

Definition 2.4 [Max adequacy] Given a threhold T we say that function fi satisfies specification fl if $FM(fl, fi) < T$.

Note that the Max adequacy criterion does not specify a test suite as the whole domain is taken into account. The next *relative* [11] adequacy criterion is defined over a test suite S and a probability.

Definition 2.5 [Threshold adequacy] Given a threshold T, a probability $p \in (0, 1]$, and a specification fl, test suite S is adequate for fi if $FM(fl, fi, S, T) > p$.

3 Generating a better test suite from a given one

Having the metrics in the above section is already quite useful because among many given test cases which are taken from the real data (such as audio and video files) or randomly generated, we can always choose a better or the best ones to test first and stop when the converted function is not good enough. This may allow us to select a smaller set of test cases that can be just as effective in detecting problems for our code. But even the real test data or the randomly generated data may not reveal enough good test cases. Therefore, we propose a method to generate a better test suite from a given one. We start with generating a better test case from a given one.

The idea of deriving a better test case from a given one is based on graph of the loss function. A better test case is the point of the domain where the loss function has a bigger value. This leads us to finding the local extrema of the given point (test case). The search process can stop when the adequacy criterion is violated.

For a test suite S we find all local extremum of all test cases in it. Because several points in a graph may share the same local extrema, the set of all local extremum, say S^*, usually has fewer elements than those in S. Assume that S^* is the better test suite that we want to find. Note that S^* is more efficient to test because it not only has fewer number of test cases but also shows bigger precision loss.

Now we prepare some formal notions of our problem domain including fixed-point numbers and local maximizer before sketching the algorithm for finding local extremum, which is also known as the combinatorial optimization problem [14]. Even though the set of floating-point and fixed-point numbers is finite, we restrict the input space to a subdomain $\mathfrak{X}$ like in optimization problems.

Definition 3.1 [Fixed-point neighborhood] Let $x = (q_1, q_2, ..., q_n) \in \mathfrak{X} \subseteq \mathbb{I}^n$. The fixed-point number $q_i \in \mathbb{I}$, $(i = 1, ..., n)$ has a_i bits in its fractional part and b_i bits in its integer part. The smallest step Δ_i we can add to or subtract from q_i is the value that the least significant bit of q_i can represent: $\Delta_i = 2^{-a_i}$. The set of neighbors of x is then defined by:

$$N(x) = \{x' = (q_1 + k_1 \times \Delta_1, ..., q_n + k_n \times \Delta_n) \in \mathfrak{X} \mid x \neq x', k_i \in \{-1, 0, 1\}\}.$$

Definition 3.2 [Local maximizer] A point $x \in \mathfrak{X}$ is called a local maximizer of l over $\mathfrak{X}$ if $l(x) \geq l(x')$ for all $x' \in N(x)$.

To search for local maximizer of l, we use the discrete steepest ascent in Algorithm 1, a gradient-based algorithm to find local maximizers of discrete functions.

This algorithm simply finds the steepest ascent direction d from an initial point to its neighbors, and then goes in direction d by an integer stepsize $\lambda \in \mathbb{N}^+$ which maximizes l in direction d. We iterate the process until a local maximizer is found. This process will stop because the input space $\mathfrak{X}$ is finite and $l(x_0)$ always increases inside the while-loop. Note that in mathematics $\underset{x}{\text{argmax}}\ f(x) := \{x \mid \forall y : f(y) < f(x)\}$.

Algorithm 1: Our steepest ascent for finding local maximizer of l

Input : an initial point $x_0 \in \mathfrak{X}$
Output: a local maximizer

```
1 Procedure SteepestAscent(x_0)
2 begin
3     while x_0 is not a local maximizer do
4         x_max ∈ argmax_x ((l(x) − l(x_0)) | x ∈ N(x_0))
5         d ← x_max − x_0                    /* the steepest ascent direction */
6         λ_max ∈ argmax_λ (l(x') | λ ∈ ℕ⁺, x' = x_0 + λd ∈ 𝔛)
7         x_0 ← x_0 + λ_max d                /* go by the stepsize in direction d */
8     return x_0
9 end
```

Note that even though we have to generate a better test suite every time we update the floating-point or fixed-point functions, the current better test suite will be the good starting one to generate instead of the original or a random one. In practice programmers usually update the number of bits used for fractional parts to have more precise fixed-point version, so most of the local maximizers often are the same.

4 Finding the best test cases

Of course it is the best to generate the best test cases which are the global extremum of the precision loss function. During the search process we also check the chosen adequacy criterion from Section 2 to stop the search process.

To find the global extremum, we consider the following global optimization problem:

$$\max_{x \in \mathfrak{X} \subset \mathbb{I}^n} l(x)$$

where $\mathfrak{X} \subset \mathbb{I}^n$ is a predetermined, finite, and non-empty set of fixed-point numbers.

According to the operation of each approach, we can categorize global optimization methods [15] into deterministic and probabilistic group. Deterministic algorithms such as State Space Search, Branch and Bound, and Algebraic Geometry are often used when the relation between objective function l and its constraints is clear and the search space $\mathfrak{X}$ is small. However, our objective function l is a black-box; in general, we do not know its shape and characteristics. In addition, our $\mathfrak{X}$ often has a large number of dimensions, which obviously hinders deterministic ones.

Probabilistic algorithms (so-called meta-heuristics) are consequently appropriate in this case. These methods often employ heuristics with iterative explorations to find good solutions, which need to be as close as possible to the optimum. Unlike simple heuristics, these algorithms do not stop in the first local optimum.

Among the most well known probabilistic algorithms, we choose Tabu Search (TS) [5] and Genetic Algorithm (GA) [6,19] to solve the problem in the paper; one belongs to neighborhood search methods and the other is in the category of evolution approach. They are two of the most effective, popular, and successful approaches for solving a wide range of combinatorial optimization problems [14]. To discover the global optimizer of l over $\mathfrak{X}$, they may complement each other perfectly.

4.1 Genetic algorithm

Genetic algorithm (GA), a population-based meta-heuristic optimization method, simulates biotic activities such as crossovers, mutations, and natural selections to gain fittest ones from initial individuals through a number of generations. GA bases on a fitness function, which is l in this case, to decide which individual is better than others. One of the most powerful features of GA is the combination of exploitation (such as crossover process) and exploration (mutation, for instance). The goal of the combination is to find not only new, similar candidates from currently known solutions, but also novel and superior ones.

Algorithm 2: Our genetic algorithm for finding global maximizer of l

Input : ps: population size
Input : ss: selection size
Input : mr: mutation rate
Output: The best found element
Data : t: the generation counter
Data : P_i: the population of generation i

```
1  Procedure GeneticAlgorithm(ps, ss, mr)
2  begin
3      t ← 0
4      generate ps elements of P_0 by randomization
5      while ¬terminationCriterion() do
6          t ← t + 1
7          evaluate every x ∈ P_{t−1} by fitness function l
8          select P_t ⊂ P_{t−1} so that P_t contains ss best individuals
9          carry out crossover and add offspring to P_t while [P_t] < ps
10         perform mutation on P_t with mutation rate mr
11     return the best element in P_t
12 end
```

We design a GA in Algorithm 2 to find global maximizer of l over $\mathfrak{X}$. Definition 4.1 and 4.2 show how we perform crossover and mutation in our GA. Our GA stops when one of the following termination criteria happens:

- The number of generations t reaches a threshold.
- The maximum computation time granted by user is exceeded.
- Current population P_t has not any better individuals than its previous population P_{t-1}.

Definition 4.1 [Crossover] Given two individuals $x_i, x_j \in \mathfrak{X}$ in population P. Mark each element in odd positions of x_i with a dot and even ones with a double dot, and do vice versa for x_j. The crossover between x_i, x_j generates two offspring $x_{ij1}, x_{ij2} \in \mathfrak{X}$ as follows:

$$\begin{aligned} x_i &= (\dot{q_{i1}}, \ddot{q_{i2}}, \dot{q_{i3}}, \ddot{q_{i4}}, \dot{q_{i5}}, ...) \\ x_j &= (\ddot{q_{j1}}, \dot{q_{j2}}, \ddot{q_{j2}}, \dot{q_{j4}}, \ddot{q_{j5}}, ...) \\ \overset{crossover}{\rightsquigarrow} x_{ij1} &= (\dot{q_{i1}}, \dot{q_{j2}}, \dot{q_{i3}}, \dot{q_{j4}}, \dot{q_{i5}}, ...) \\ \overset{crossover}{\rightsquigarrow} x_{ij2} &= (\ddot{q_{j1}}, \ddot{q_{i2}}, \ddot{q_{j3}}, \ddot{q_{i4}}, \ddot{q_{j5}}, ...) \end{aligned}$$

Definition 4.2 [Mutation] Given an individual $x = (q_1, q_2, ..., q_n) \in \mathfrak{X}$ in population P. Randomly choose $i \in \mathbb{N}^+, 1 \leq i \leq n$ and assign a random number r to q_i so that $(q_1, q_2, ..., q_{i-1}, r, q_{i+1}, ..., q_n) \in \mathfrak{X}$. The mutation process generates $x' \in \mathfrak{X}$ from x as follows:

$$\begin{aligned} x &= (q_1, q_2, ..., q_{i-1}, q_i, q_{i+1}, ..., q_n) \\ \overset{mutation}{\rightsquigarrow} x' &= (q_1, q_2, ..., q_{i-1}, r, q_{i+1}, ..., q_n) \end{aligned}$$

4.2 Tabu search

Because traditional neighborhood-based methods like steepest ascent are easy to be stuck in local optima, it is ineffective to use them to find global optima of complex functions, for instance, Rosenbrock [23]. TS improves this drawback by providing a tabu list to remember which points are recently visited, helping us leave local optima to enter new, promising areas. Also, TS uses intensification and diversification to search not only deeper but also broader, as GA does with exploitation and exploration. In Algorithm 3, we introduce our version of TS to deal with our problem. Note that each time a new initial solution is randomly created, our TS sets a limited computation time for the process of finding the best solution started from the initial one. If the time is exceeded, we stop discovering and prepare for the next start.

5 Experimental results

We conducted experiments to point out the best test case for testing the conversion from a floating-point function into a corresponding fixed-point one. From the most

Algorithm 3: Our tabu search algorithm for finding global maximizer of l

Input : s: the size of Tabu list
Input : ni: the number of initial solutions
Output: x^*: the best element
Data : t: the counter of initial solutions
Data : L: the tabu list

```
1  Procedure TabuSearch(s, ni)
2  begin
3      L ← ∅
4      randomly choose x* ∈ 𝔛
5      t ← 0
6      while t < ni do
7          t ← t + 1
8          generate x0 ∈ 𝔛 by randomization
9          while ¬terminationCriterion() do
10             x_max ∈ argmax_x ((l(x) − l(x0)) | x ∈ N(x0), x ∉ L)
11             if l(x_max) > l(x*) then
12                 x* ← x_max                    /* update the best record */
13             if [L] = s then                   /* L has no more than s elements */
14                 delete the first member in L
15             L.append(x_max)
16             x0 = x_max                        /* jump to the best neighbor not in L */
17     return x*
18 end
```

well-known functions used for examining the performance of global optimization methods, we choose a list of complex and popular functions in Table 1. Deriving from each f in the list, we define $fl : \mathbb{L}^n \to \mathbb{L}$, $fi : \mathbb{I}^n \to \mathbb{I}$, and $l : \mathbb{I}^n \to \mathbb{L}^+$ as stated in Section 2. Note that in our problem, we do not take into account finding the global maximizers of functions in Table 1; our goal is to find the global maximizer of the corresponding l over $\mathfrak{X}$ of each function f in Table 1.

Both the first column of Table 1 and that of Table 2 denote the function number. Each function has a domain presented in column $\mathfrak{X}$ in Table 2. Due to the domain and the environmental precision of fixed-point number in each function (the number of bits used for representing the fractional part, which we set by 6 for all functions), the number of (discrete) possible solutions of each function is shown in column **Total**. Column **MLoss** contains the maximum loss of each function we find out by brute-force search. Columns **GA** and **TS** show the highest loss values of the best test cases returned by our GA in Algorithm 2 and TS in Algorithm 3 with maximum accepted computational time is 5 seconds, respectively. For clarity, all the numbers of **MLoss**, **GA**, and **TS** are rounded to have six digits after the decimal point. For our GA, we set $ps = 100, ss = 30, mr = 20$ and the maximum allowed number of

Table 1
List of functions

No.	Name	Definition
1	Beale	$f(x,y) = (1.5 - x(1-y))^2 + (2.25 - x(1-y^2))^2 + (2.625 - x(1-y^3))^2$
2	Booth	$f(x,y) = (x + 2y - 7)^2 + (2x + y - 5)^2$
3	Branin	$f(x,y) = (y - \frac{5}{4\pi^2}x^2 + \frac{5}{\pi}x - 6)^2 + 10(1 - \frac{1}{8\pi})\cos x + 10$
4	Freudenstein & Roth	$f(x,y) = (-13 + x + ((5-y)y - 2)y)^2 + (-29 + x + ((y+1)y - 14)y)^2$
5	Himmelblau	$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$
6	Hump	$f(x,y) = 4x^2 - 2.1x^4 + \frac{x^6}{3} + xy - 4y^2 + 4y^4$
7	Matyas	$f(x,y) = 0.26(x^2 + y^2) - 0.48xy$
8	Rosenbrock's Banana	$f(x,y) = 100(y - x^2)^2 + (1 - x)^2$

generations is 1000. Our TS uses tabu lists of length 1000 and the number of initial solutions is 500.

Table 2
Experimental results

No.	$\mathfrak{X}$	Total	MLoss	GA	TS
1	$x, y \in [-20, 20]$	6558721	9.085015	9.085015	7.421680
2	$x, y \in [-60, 60]$	58997761	20.855000	19.505000	19.935000
3	$x, y \in [-5, 5]$	410881	38.656643	38.656643	35.992882
4	$x, y \in [-10, 10]$	1640961	14091.282934	14091.282934	14089.997935
5	$x, y \in [-9, 9]$	1329409	53.580487	53.580487	37.727734
6	$x, y \in [-4, 4]$	263169	71.266085	71.266085	71.266085
7	$x, y \in [-50, 50]$	40972801	39.838400	39.838400	34.381496
8	$x, y \in [-3, 3]$	148225	140.157744	140.157744	140.127744

Table 2 demonstrates that overflow can cause fixed-point functions to be dramatically different from their original floating-point ones, especially for function Freudenstein & Roth at No. 4. We also observe from Table 2 that our GA outweighs TS in most cases of our experiments; however, we should combine the two methods to obtain good results.

The experiments were carried out on an Intel(R) Pentium(R) Dual CPU 1.81 GHz with 1GB of RAM. We use *MathFP 2.0.6 KVM, CLDC/MIDP and WABA*[5] to implement fixed-point environment.

[5] http://home.comcast.net/~ohommes/MathFP/ [accessed 2009-10-28]

6 Related work

In Digital Signal Processing, algorithms are often designed with floating-point data types, but their corresponding implementations need to be executable in hardware that only supports fixed-point operators. Therefore, fixed-point conversion is indispensable in both software [17] and hardware implementation [9,22]. Although fixed-point arithmetic is simpler, cheaper, and faster than floating-point one, it results in degrading the performance of applications, which can be evaluated by simulation-based [1,12] and analytical approaches [16,17,21]. Nonetheless, to the best of our knowledge, no method evaluates the worst case of the degradation, which happens when the difference between a floating-point function and its fixed-point version is highest, as this paper does.

Our work also relates to roundoff error analysis, which aims to calculate the difference between an approximate number, function, or program method and their exact value. The difference results from using finite digits to represent numbers in computer. There are several different approaches on the topic. Simulation-based approach [2] attempts to run both types of programs with a selected set of inputs, and then observes the corresponding roundoff error. Although this approach is easier to be carried out, it leads to a trade-off between the accuracy and the speed of the method. Another approach is using mathematical reasoning, in which Giraud et al. [4] works with classical Gram-Schmidt algorithm and Fang et al. [3] uses affine arithmetic. Their inspiring methods, however, are highly theoretical and inconvenient for complex real-world functions. Eric Goubault and Sylvie Putot [7,8] overcomes the problem by statically analyzing and propagating range values of program's variables using affine arithmetic, but their method incurs over-approximation. Our approach, unlike them, does not need to analyze given programs, but we also get the above trade-off when the number of dimensions of input arguments becomes large.

Automatic test case generation recently draws a lot of attention in research community. A comprehensive discussions and references about software testing can be found in the book chapter [11] by Kapfhammer. The work of Zhu et al. [24] also contains a comprehensive survey on unit test coverage and adequacy criteria. Test case generation techniques are divided into three main groups: random, symbolic and dynamic. Our work belongs to the dynamic one even though we only observe the output of the test functions and treat them as a black-box. Michael et al. [18] uses genetic algorithms to generate test data. Korel [13] also generates test data by executing the program but he monitors the flow of execution to generate test data for path coverage criteria. We are not aware of related work that directly generates test cases for assessing the closeness of two functions like us. Note that our work, however, does not exclude other test case generation techniques such as symbolic execution. They are still needed to include test cases that cover other test adequacy criteria such as control-flow based, data-flow based.

One important technique in our testing method is the use of optimization methods. They are ongoing research topics in not only theoretical fields but also practical applications [15]. Our Steepest Ascent in Algorithm 1 shares the same ideas with the discrete steepest descent method in the paper of Ng et al. [20]. Their method,

entitled the discrete global descent method, can return optimal global minimizers for functions that agree with their three reasonable assumptions. Recently, Hvattum and Glover [10] combine traditional direct search methods and Scatter Search to find local optima of high-dimensional functions. However, because our function l is black-box, these inspiring methods are not viable to be used here. To solve our problem, we choose GA and TS, the most popular, powerful probabilistic global optimization methods from which many hybrid algorithms are derived.

7 Conclusions

We have introduced several metrics to measure the quality of a fixed-point function converted from a floating-point one. When testing data are inconvenient to be generated manually or randomly, these metrics allow us to compare any two test suites and to quantify the quality of the fixed-point function with respect to its floating-point one. The generation can be stopped by our novel adequacy criterion that takes the source function as a specification for the converted one. In addition, we also presented a method to generate a smaller but more efficient test suite from a given one and applied some results from global optimization problem to generate the best test case. Experimental results have shown that our approach can be quite useful in practice. It allows us to reduce testing time, and can also increase the quality of software products written for embedded systems.

For future work, we plan to investigate Quasi-Monte Carlo methods to calculate a metric based on integral of the loss function and some variations of it. We also plan to strengthen our approach by taking into account the implementation of the functions. This source code analysis may help us identify parameters or variables of the functions that have a stronger effect on the precision loss. This information may help heuristic algorithms to converge faster and may provide the programmers some hints to refine or fix their code.

Acknowledgement

This work is supported by the research project No. QGTD.09.02 (granted by Vietnam National University, Hanoi) and Panasonic R&D Center Vietnam. We also thank Hong-Hai Nguyen, Ngoc-Khoa Pham, and Hieu Tran at Panasonic R&D Center Vietnam as well as anonymous reviewers for their comments on the earlier version of the paper.

References

[1] Pavle Belanovic and Markus Rupp. Automated floating-point to fixed-point conversion with the fixify environment. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, pages 172–178, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Richard Burch, Farid Najm, Ping Yang, and Timothy Trick. McPOWER: a Monte Carlo approach to power estimation. In *ICCAD '92: 1992 IEEE/ACM international conference proceedings on Computer-aided design*, pages 90–97, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[3] Claire F. Fang, Rob A. Rutenbar, and Tsuhan Chen. Fast, Accurate Static Analysis for Fixed-Point Finite-Precision Effects in DSP Designs. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, pages 275–282, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Luc Giraud, Julien Langou, Miroslav Rozložník, and Jasper van den Eshof. Rounding error analysis of the classical Gram-Schmidt orthogonalization process. *Numerische Mathematik*, 101(1):87–100, 2005.

[5] Fred Glover and Fred Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[6] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[7] Eric Goubault and Sylvie Putot. Static Analysis of Numerical Algorithms. In *SAS*, pages 18–34, 2006.

[8] Eric Goubault and Sylvie Putot. Under-Approximations of Computations in Real Numbers Based on Generalized Affine Arithmetic. In Hanne Riis Nielson and Gilberto Fil, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2007.

[9] N. Herve, D. Menard, and O. Sentieys. Data wordlength optimization for FPGA synthesis. In *SIPS '05: Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 623–628, Athens, Grece, November 2005.

[10] Lars Magnus Hvattum and Fred Glover. Finding local optima of high-dimensional functions using direct search methods. *European Journal of Operational Research*, 195(1):31–45, May 2009.

[11] Gregory M. Kapfhammer. *The Computer Science and Engineering Handbook*, chapter Chapter 105: Software Testing. CRC Press, Boca Raton, FL, second edition, 2004.

[12] Holger Keding, Markus Willems, Martin Coors, and Heinrich Meyr. Fridge: A fixed-point design and simulation environment. In *DATE 98: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 429–435, 1998.

[13] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.

[14] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, Germany, 4th edition, 2007.

[15] Leo Liberti and Nelson Maculan. *Global Optimization: From Theory to Implementation (Nonconvex Optimization and Its Applications)*. Springer, 2006.

[16] D. Menard, R. Serizel, R. Rocher, and O. Sentieys. Accuracy constraint determination in fixed-point system design. *EURASIP J. Embedded Syst.*, 2008(6):1–12, 2008.

[17] Daniel Menard, Daniel Chillet, and Olivier Sentieys. Floating-to-fixed-point conversion for digital signal processors. *EURASIP Journal on Applied Signal Processing.*, 2006:1–19, 2006.

[18] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, 2001.

[19] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[20] Chi-Kong Ng, Duan Li, and Lian-Sheng Zhang. Discrete global descent method for discrete global optimization and nonlinear integer programming. *Journal of Global Optimization*, 37(3):357–379, 2007.

[21] Emre Özer, Andy Nisbet, and David Gregg. Stochastic bit-width approximation using extreme value theory for customizable processors. In Evelyn Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2004.

[22] Romuald Rocher, Daniel Menard, Nicolas Herve, and Olivier Sentieys. Fixed-point configurable hardware components. *EURASIP J. Embedded Syst.*, 2006(1):20–20, 2006.

[23] Yun-Wei Shang and Yu-Huang Qiu. A Note on the Extended Rosenbrock Function. *Evol. Comput.*, 14(1):119–126, 2006.

[24] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.