Static Analysis on Floating-Point Programs Dealing with Division Operations

MG Thushara[1]
Department of Computer Science and Applications
Amrita School of Engineering, Amritapuri
Amrita Vishwa Vidyapeetham, India

K. Somasundaram[2]
Department of Mathematics
Amrita School of Engineering-Coimbatore
Amrita Vishwa Vidyapeetham, India

***Abstract*—Numerical accuracy is a critical point in safe computations when it comes to floating-point programs. Given a certain accuracy for the inputs of a program, the static analysis computes a safe approximation of the accuracy on the outputs. This accuracy depends on the propagation of the errors on the data and on the round-off errors on the arithmetic operations performed during the execution. Floating point values disposes a large dynamic range. But the main pitfall is the inaccuracies that occur with floating point computations. Based on the theory of abstract interpretation, in the paper an upper bound to the precision of the results of these computations in program have been demonstrated.**

***Keywords*—Abstract interpretation; static analysis; forward analysis; abstract domain**

I. Introduction

The optimization of floating point computations in high performance computing is a critical problem. Most of the programming language is restricted in the ability to optimize the computations. The error propagation factor in the computations are considered to the minimal. Here, the paper introduces manual static analysis for floating point computation that deals with division operations in programs. The semantics of floating-point numbers are complicated and range of values comes in accordance with the precision.

There are different sources of inaccuracies in floating-point numbers like limited precision arithmetic, error accumulation as a result of floating-point computations, the result of a floating point computation when it becomes an input to some other function or process.

In programs with floating-point computations, it is demanding to have numerical accuracy in the results. Our approach is to combine a forward and a backward static analysis, done by abstract interpretation. The forward analysis is a classical approach where the errors on the inputs and on the results of the intermediary operations are safely propagated to determine the accuracy of the results. Based on the results of the forward analysis and on assertions indicating the accuracy required by the user for the outputs at the end of the execution, the backward analysis will be carried out. Backward analysis computes the minimal accuracy needed for the inputs and intermediary results of the program in order to satisfy the assertions made. In order to refine the results until a fixed-point is reached, the forward analyses and backward analyses can be applied repeatedly.

Static analysis are useful in several safety critical contexts. For instance, the explosion of the rocket-Ariane 5 [1], owing to a software error in the inertial reference system. Specifically, a 64-bit floating-point number was converted to a 16 bit signed integer which was larger than 32,767, the largest integer in a 16 bit signed integer, and that lead to the failure. Another instance was Patriot Missile [2] failed in detecting and intercepting an incoming Iraqi Scud missile and killing 18 American army men during the Gulf war. The cause of the incident was an inaccurate calculation of the time due to computer arithmetic errors.

Technically,abstract values are used in the form $[a, b]_p$ where a and b are floating-point numbers defining an interval and p is an integer giving the accuracy. Intuitively, $[a, b]_p$ is the set of numbers between a and b which have at least p correct digits.

Using the principles of abstract interpretation, an abstract domain is defined for floating point numbers using intervals. By static analysis with forward and backward analysis the input and results of the computations are optimized using the division operations.

II. Background Study

A. Abstract Interpretation

Static analysis involves defining a abstract domain [3] and computing automatically the program text with the abstract semantics according to predefined abstractions. Abstract interpretation uses theory of sound approximation of computer program semantics. Using the control-flow or data-flow, without doing all computations, information can be obtained about semantics of the program. In formal static analysis, these information can be used to analyze the behaviour of the possible executions of computer programs. The concept of Abstract interpretation was coined by the computer scientist working couple Patrick Cousot and Radhia Cousot in 1970s.

Real numbers are approximated by floating-point arithmetic [4], so error may propagate due to rounding during computations. Even though this seems to be accurate, losing precisions in safety critical applications will make the results useless. For dealing with this issue the authors in [4], came up with a tool using static analysis which allows to find the possible programming errors.

B. IEEE 754 Floating-Point Arithmetic

The programs containing floating-point computations [5] is critical when it comes to verification. This is mainly due to rounding errors, infinities, non-numeric objects (NaNs), signed zeroes, denormal numbers, different rounding modes, etc. Here, the authors has define and proved the correctness of algorithms which are using the values bound with variables x, y or z. This filtering algorithms are defined and formally proved for their correctness.

David Goldberg in his paper [6] discusses most of the aspects related to floating-point arithmetic. A look through on the implications of rounding for operations like addition, subtraction, multiplication and division. Also, explains the IEEE floating-point standard and also discusses different aspects of computer systems include design of instruction set, compiler optimization and exception handling.

In [7], a classification of floating point formats, including IEEE Standard 754 is presented by the author.

In [8], the author discusses about Interval mathematics that guarantees result but the arithmetic on existing processors makes these methods very slow. The paper looks into the efficiency of interval arithmetic on computers. Interval arithmetic can be considered as an extension of floating-point arithmetic. This controls the precision of a computation as well as the accuracy of the computed result.

C. Literature Reviews

FLUCTUAT [9] is a static analyzer for analysing the errors generated from approximations of floating-point arithmetic operations on real numbers. It mainly focuses on error computation using relational methods.

Round-off error in floating-point is formally verified using the tool FPTaylor in [10]. The approach used in the paper is Symbolic Taylor Expansions and the implementation of the tool called FPTaylor which is built on this approach.

The tool -VCFloat [11] automatically deals with rounding errors in real-number expressions of C floating-point computations and does the correctness proof using Coq.

In [12], a static analysis computing round-off error bounds on floating-point computation is introduced. The paper uses denotational semantics for the estimation of round-off errors.Authors have developed a prototype - PRECiSA (Program Round-off Error Certifier via Static Analysis) for verification in NASA for floating-point computations.

III. Methodology

In Fig. 1, a sample code that can lead to inaccuracies is depicted an the expected and actual result is shown in Fig. 2. An error of 0.00000000000000011102 is found in the result.

In this paper, static analysis for numerical accuracy is introduced. The information gained by this analysis will be used for the optimizing the floating-point representation. In order to show the correctness, experimental results will be presented in the coming sessions. Concrete semantics of a program is the set of all possible executions in all possible

```
double x=0.1; int i=1;
while (i <= 10)
{
  x = x + 0.1;
  printf("x=%0.20f", x);
  i++;
}
```

Fig. 1. Error Propagating code.

```
Expected output: At 10th iteration x = 1.0

Actual output:
(1) x=0.10000000000000000555
(2) x=0.20000000000000001110
(3) x=0.30000000000000004441
(4) x=0.40000000000000002220
(5) x=0.50000000000000000000
(6) x=0.59999999999999997780
(7) x=0.69999999999999995559
(8) x=0.79999999999999993339
(9) x=0.89999999999999991118
(10)x=0.99999999999999988898
```

Fig. 2. A simple Code snippet.

environment. Whereas Abstract semantics is a super set of the concrete semantics. In general, Abstract Interpretation [3] is a theory of semantics-based program analysis.

An abstract domain α is chosen replacing the objects of concrete domain S as $\alpha(S)$. For every program there is a corresponding computation tree or control flow graph.

```
(0) a = 1; b = 0;
(1) while (a < 10)
    {
      (2)  b = b + 1;
      (3)  a = a + 1;
    }
(4) Print b
(5) End
```

Fig. 3. A simple Code snippet.

Consider the code in Fig. 3, the corresponding control flow graph is shown in Fig. 4. The rules that allows us to compute the precision of the noes in a control flow graph is known as transfer functions. Two classes of transfer functions are used- the one that computes the past behaviour for each control point is transfer functions for forward analysis and the one that computes information about the future behaviour for each program point is transfer functions for backward analysis.

Here, floating point numbers are represented as intervals in the abstract domain. The transfer functions for division operation is then formulated. Using the principle of interval arithmetic the computations are further proceeded.

In this paper the approach is by defining the abstract domain whose elements are floating-point intervals with an associate precision. Then the transfer function for the division

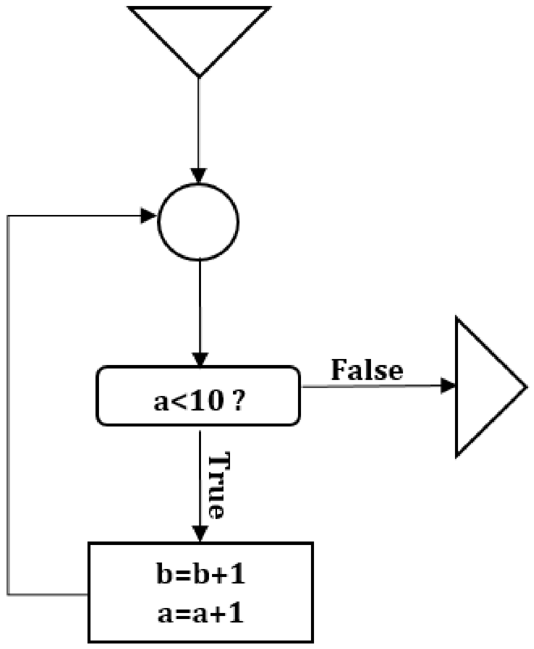
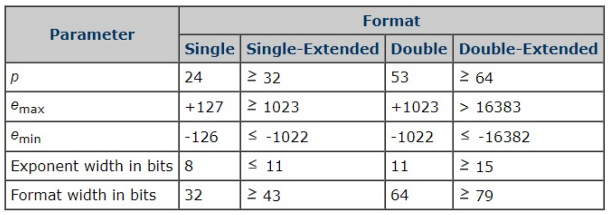


Fig. 4. A sample control flow graph

operator is formulated. Here the abstract domain is defined as $[a, b]_p$ where a and b are interval bounds and p is the precision. The IEEE754 standard Fig. 5 defines four different precisions: single, double, single-extended, and double-extended. In IEEE 754, single and double precision correspond roughly to what most foating-point hardware provides.



Parameter	Format			
	Single	Single-Extended	Double	Double-Extended
p	24	≥ 32	53	≥ 64
e_{max}	+127	≥ 1023	+1023	> 16383
e_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent width in bits	8	≤ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

Fig. 5. IEEE standard

IV. Abstract Domain and Running Example

Let I_p be the set of all floating-point intervals with a precision p. An element $i \in I_p$ is denoted as $i = [a, b]_p$ where a and b are two floating-point numbers and p is the precision (length of the mantissa). Consider β_p as the set of all binary representations with p as the mantissa length.

Then,

$$I_p = [a, b]_p = \{c \in \beta_p : a \leq c \leq b\}$$

and

$$I = \bigcup_{p \in \mathbb{N}} I_p.$$

The abstract domain is defined as $\langle I, \sqsubseteq, \sqcup, \sqcap, \bot_I, \top_I \rangle$. The elements are ordered by

$$[a, b]_p \sqsubseteq [c, d]_q \iff [a, b] \subseteq [c, d] \text{ and } q \leq p.$$

a) Transfer Functions: The computations in a program can be represented using computation tree which demonstrates the control-flow of the program. For this, transfer functions are used which analyses the value at each computation node using the information from the past behaviour of the node.

To define Control Flow Graph(CFG) semantics, an operational semantics can be defined which is like an interpreter where the entry node is the input and is the initial state which is the mapping from variables to values and the final state is the output of the program. These semantics are defined in terms of transfer functions. The transfer function holds the execution semantics of that node and specifies the next node to be executed. If an abstract interpretation is done and then formulate a transfer function, the result is exactly the same as that of the results of doing operational semantics to actual value and then do abstraction. Consider the division of two intervals $x = [a, b]_{p1}$ and $y = [c, d]_{p2}$ with $a = s_1 \cdot m_1 \cdot 2^{e_1}$, $b = s'_1 \cdot m'_1 \cdot 2^{e'_1}$, $c = s_2 \cdot m_2 \cdot 2^{e_2}$ and $d = s'_2 \cdot m'_2 \cdot 2^{e'_2}$.

In the forward analysis, in order to estimate the precision for z different cases are taken. Let $d = e_2 - e_1$.

$$p = \begin{cases} max(p_1, p_2), \text{if } e_1 = e_2, \\ min(p_1 - 1, p_2) + d, \text{if } e_1 > e_2, \\ min(p_1, p_2 - 1) + d, \text{if } e_1 < e_2. \end{cases}$$

Rounding error [6] is part of floating-point computations. If z is the floating-point number represented by $d.d..dx\beta^e$, then the error can be represented as $d.d..d - (z/\beta^e)|\beta^{p-1}$.

In this section, the numerical analysis is demonstrated using a sample code snippet given in Fig. 6. Each value is represented as a floating point interval in the form $[a, b]_p$ where a and b are the interval range and p is the precision of the float value. All the variable values are represented in the form of interval of float value in their abstract form. Each time an operation is performed, the precision is affected and in the paper optimization of the precision is tried by applying forward analysis.

```
(1) float b=102.0 , a=2.0_15;
    (2) while(b>1.0)
        {
    (3)         b=b/a;
    (4)         printf("%f",b);
        }
```

Fig. 6. An example Code snippet.

In Fig. 6, two variables a and b are used and they are initialized as 102.0 and 2.0_{15}. Which is internally mapped to abstract domain in the form $[102.0, 102.0]_{32}$ and $[2.0, 2.0]_{15}$. Here, the while loop is used to reduce the value of b by dividing b by a. Static forward analysis is applied on the given code snippet which is demonstrated in Fig. 7.

The predefined abstractions are used to compute the abstract semantics automatically from the program text which is then optimized automatically or manually by the user. At control point (3), b' is calculated by interval division of b and a. The new value is obtained as $[51.0, 51.0]_{32}$ by using transfer function for division. At beginning of each iteration a join operation is performed on the values of b. After control point (3), a join ($\cup$) operation based on previous value of b from control point (1) and current value of b from (3) is performed.

To over approximate the value of b a widening operation is performed after control point (3)". It is observed that the value of b becomes unchanged after every iteration. Then the loop is stopped and determines the forward analysis result of b. This is shown in Fig. 7.

```
(1) float b=[102.0,102.0]32,
    a=[2.0,2.0]15;

(3)       b′= [102.0,102.0]32/ [2.0,2.0]15
        =[51.0,51.0]32

    b=b ∪ b′
    =[102.0,102.0}32 ∪ [51.0,51.0]32
    =[51.0,102.0]32

(3)′ b′′=[51.0,102.0}32/ [2.0,2.0]15
        =[25.5,51.0]15

    b=b′ ∪ b′′
    =[51.0,102.0}32 ∪ [25.5,51.0]15
    =[25.5,51.0]32

(3)′′ b′′′=[25.5,51.0]32/ [2.0,2.0]15
        =[12.75,25.5.0]15

    b=b′′ ▽ b′′′
    =[25.5,51.0]32 ▽ [12.75,25.5]15
    =[12.0,26.0]32

(3)′′′ b′′′′=[12.0,26.0]32/ [2.0,2.0]15
        =[6.0,13.0]15

    b=b′′′ ∪ b′′′′
    =[12.0,26.0]32 ∪ [6.0,13.0]15
    =[12.0,26.0]32
```

Fig. 7. An example Code snippet.

V. Conclusion

In this paper, A static numerical analysis is demonstrated on floating-point computation with the help of a code snippet. The abstract domain of the form $[a, b]_p$ is used where a and b are the range of float values and p is the precision. Here, forward analysis is applied to show the float-interval division operation. The aim of the paper is to prove that a minimal accuracy is achieved on the result of the floating-point computations where mostly due to rounding errors and error propagation, the results get affected. With the help of transfer function, the analysis result is demonstrated in Fig. 7. Transfer functions are used to show the numerical analysis applied on the floating-point computations.

References

[1] European Space Agency. European Space Agency, Ariane 501 Inquiry Board Report. Technical report, , 1996.

[2] Information Management and D.C. Technology Division, Washington. Patriot missile defense: Software problems led to system failure at Dhahran, Saudi Arabia. Technical report, Information Management and Technology Division, US General Accounting Office, 1992.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[4] Eric Goubault, Matthieu Martel, and Sylvie Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In Daniel Le Métayer, editor, *Programming Languages and Systems*, pages 209–212, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[5] Roberto Bagnara, Abramo Bagnara, Fabio Biselli, Michele Chiari, and Roberta Gori. Correct approximation of ieee 754 floating-point arithmetic for program verification, 2019.

[6] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[7] David M. Russinoff. *Floating-Point Formats*, pages 63–75. Springer International Publishing, Cham, 2019.

[8] Ulrich Kulisch. Mathematics and speed for interval arithmetic: A complement to ieee 1788. *ACM Trans. Math. Softw.*, 45(1):5:1–5:22, March 2019.

[9] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In Kwangkeun Yi, editor, *Static Analysis*, pages 18–34, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[10] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods*, pages 532–550, Cham, 2015. Springer International Publishing.

[11] Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. A unified coq framework for verifying c programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 15–26, New York, NY, USA, 2016. ACM.

[12] Mariano Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security*, pages 213–229, Cham, 2017. Springer International Publishing.