Overflow and Roundoff Error Analysis via Model Checking

Do Thi Bich Ngoc
School of Information Science
Japan Advanced Institute of Science and Technology
dongoc@jaist.ac.jp

Mizuhito Ogawa
School of Information Science
Japan Advanced Institute of Science and Technology
mizuhito@jaist.ac.jp

Abstract

This paper proposes a framework for statically analyzing overflow and roundoff errors of C programs. First, a new range representation, "extended affine interval", is proposed to estimate overflow and roundoff errors. Second, the overflow and roundoff error analysis problem is encoded as a weighted model checking problem. To avoid widening, currently we focus on programs with bounded loops, which typically appear in encoder/decoder reference algorithms. Last, we implement the proposed framework as a static analysis tool CANA. Experimental results on small programs show that the extended affine interval is much more precise than classical interval.

Key words: program analysis, model checking, roundoff error, overflow error, affine interval.

1. Introduction

In computer algorithms, real numbers are often represented as floating point numbers. However, hardware typically uses *fixed point number* representations for lower cost and higher speed. Direct transformation from a reference algorithm to a hardware algorithm with fixed point numbers often returns different computational results. This causes serious problems, especially in digital signal processing, such as in Mpeg decoder and OpenGL libraries.

There have been several works on overflow and roundoff error (ORE) analysis [2], [6], [8], [4], [5]. These works mostly focus on analyzing roundoff errors in floating point systems. In this work, we aim to analyze OREs of fixed point (reference) algorithms.

Based on abstract interpretation technique [1], OREs are analyzed by automatically propagating the ranges of variable values and their roundoff errors. There are two main techniques to propagate ranges. The first method uses *classical interval* (CI) [9] to represent possible ranges. This method is simple but imprecise, because it does not handle correlations between variables. The second method uses *affine interval* (AI) [4], [13], [14], which introduces symbolic manipulations on noise symbols, to handle correlations between variables. AI arithmetic supplies higher precision, especially in linear operations (e.g., addition, subtraction). However, for nonlinear operations (e.g., multiplication, division), AI arithmetic introduces a fresh noise symbol each time. This leads to high complexity if there are many nonlinear operations.

This paper proposes a framework for statically analyzing OREs for C programs with bounded loops.

- First, we propose an *extended affine interval* (EAI) by assigning a CI coefficient to each noise symbol of AI form. Compared to AI arithmetic, EAI arithmetic does not need to introduce new noise symbols for any operations.
- Second, ORE analysis is encoded as a weighted model checking problem [11]. We represent C programs by weighted transition systems, in which weights are designed using range representations (e.g., CI, AI, EAI). To avoid widening, currently we focus on programs with bounded loops only, which typically appear in encoder/decoder reference algorithms.
- We implement the proposed framework as a static analysis tool CANA, to analyze OREs of fixed point algorithms. The backend engine is Weighted PDS library [1]. The input of CANA is C program with nested loops (64×64), without procedure calls or pointer manipulations. Experimental results on small programs demonstrate that EAI is more precise than CI and is comparable to AI.

The rest of this paper is organized as follows. Section 2 presents ORE problem for fixed point numbers. In Section 3, we recall the basic notions of CI and AI arithmetics. Then, we propose a new range representation, *EAI*, and its arithmetic. We introduce an abstract domain for ORE problems in Section 4.

Section 5 shows how to encode ORE analysis as a weighted model checking problem. Section 6 shows CANA implementation and its experimental results. Section 7 mentions related works, and Section 8 concludes the paper and indicates future work.

[1] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/

2. Overflow and roundoff error problems

2.1. Fixed point numbers and errors

Fixed point numbers are a simple and easy way to express real numbers, using a fixed number of bits. Fixed point numbers are generally used when hardware cost, speed, or complexity are important issues. We recall some notations related to fixed point numbers, as follows:

Definition 1 (Fixed point number): A **fixed point** number a on base **b** is represented in the form: $a = sp\; a_1a_2\ldots a_{ip}\cdot a_{ip+1}\ldots a_{ip+fp}$, where **sign part** $sp\in\{0,1\}$ determines if a is sign or unsign, $a_k \in [0, b-1]$ $\forall k \in [1, ip+fp]$, ip is the width of **integer part**, and fp is the width of the **fraction part**.

A real number x is represented by a pair (x_f, x_r) where x_f is the **fixed point** value and x_r is the corresponding **roundoff error**.

There are two types of fixed point errors (FE): *roundoff error* due to the finite fraction part, and *overflow error* due to the finite integer part. In programs, one must strike a balance between overflow and roundoff errors; by scaling down the data, the occurrence of overflow error is reduced, but the relative size of the roundoff error is increased. Hence, we need to consider the largest value of roundoff error and the largest value of fixed point number. The overflow and roundoff error (ORE) problems are stated as follows:

Given a program, initial ranges of variables, and fixed point format, there are some natural questions:

1) *Whether roundoff error of a result lies within threshold bound or not?*
2) *Whether overflow error may occur? Where?*

For clarity, let us illustrate the above problem by a simple example:

Example 1: Assuming that the input C program as shown in Figure 1, the inputs of the program satisfy $x \in [0,1]$, $n = 100$, and real variables (x, rst) are represented by using fixed point format $(sp = 1, ip = 7, fp = 8)$. The questions are:

1) Does roundoff error of rst lie within $[-0.01,\ 0.01]$?
2) May overflow error occur? Where?

2.2. FE operations

In fixed point arithmetic, the result must be rounded or truncated to fit the result into the same number of bits as the operands. The computation of fixed point arithmetic has roundoff error ϵ, such that $|\epsilon| \leqslant b^{-fp}/2$. The roundoff error of a result is then the sum of ϵ and the result of propagating the roundoff error of operands. The roundoff error of arithmetic on fixed point numbers is computed by using FE arithmetic, as follows:

```
 /* CANA
 CANA ALL sign 7 8
 global x range 0 1
 global n range 100 100
*/
double x; int n;
int main () {
    int i;
    double  rst;
    rst = x;  i = 0;
    while (i <= n) {
        rst = rst + i ;   i++; }
        rst = rst/10.0f ;
        rst = rst  - x * x;
    return 1;
}
```

Figure 1. An example of a C program

Definition 2 (FE arithmetic): Let $(x_f,\ x_r)$ and $(y_f,\ y_r)$ be representations of $x,\ y$, and let ϵ be a noise symbol such that $|\epsilon| \leqslant b^{-fp}/2$. FE arithmetic $\boxed{\ast} = \{\boxplus, \boxminus, \boxtimes, \boxed{\div}\}$ is defined below.

$$\begin{aligned}
(x_f, x_r) \boxplus (y_f, y_r) &= (x_f + y_f,\ x_r + y_r)\\
(x_f, x_r) \boxminus (y_f, y_r) &= (x_f - y_f,\ x_r - y_r)\\
(x_f, x_r) \boxtimes (y_f, y_r) &= (x_f \times y_f,\\
&\qquad x_r \times y_f + x_f \times y_r + x_r \times y_r + \epsilon)\\
(x_f, x_r) \boxed{\div} (y_f, y_r) &= (x_f \div y_f,\\
&\qquad (x_f + x_r) \div (y_f + y_r) - x_f \div y_f + \epsilon)
\end{aligned}$$

Because of roundoff error, the result of fixed point conditional expression is sometimes different from the result of real number conditional expression. Therefore, the fixed point program leads to incorrect results. We define the FE comparison operations for ORE problems by comparing the range values of real number representations. For a given real number representation (x_f, x_r), the corresponding range values are $\tilde{x} = [x_f - |x_r|, x_f + |x_r|]$. The results of FE comparison operations may be **true**, **false**, or **unknown**. **Unknown** means that the result of real number expression may differ from that of fixed point expression. Formally, FE comparison operations are defined as follows:

Definition 3 (FE comparison operations): Let (x_f, x_r), and (y_f, y_r) be representations of two numbers.

$$(x_f, x_r) \leqslant (y_f, y_r) = \begin{cases} \textbf{true} \text{ if } \forall u \in \tilde{x}\ \forall v \in \tilde{y}. u \leqslant v \\ \textbf{false} \text{ if } \forall u \in \tilde{x}\ \forall v \in \tilde{y}. u > v \\ \textbf{unknown} \text{ otherwise} \end{cases}$$

$$(x_f, x_r) = (y_f, y_r) = \begin{cases} \textbf{true} \text{ if } (x_f = y_f \;\wedge\; x_r = y_r = 0) \\ \textbf{false} \text{ if } (\forall u \in \tilde{x}\ \forall v \in \tilde{y}. u < v) \\ \qquad\qquad \vee\; (\forall u \in \tilde{x}\ \forall v \in \tilde{y}. u > v) \\ \textbf{unknown} \text{ otherwise} \end{cases}$$

where $\tilde{x},\ \tilde{y}$ are range values of (x_f, x_r), (y_f, y_r) respectively.

Remark 1: Other comparison operations (e.g., $>$, $! =$) can be defined using the above operations.

3. Range representations

To estimate ORE of arithmetic on fixed point numbers, there are two known over-approximations: classical interval [9] and affine interval [13], [14]. In this section, we describe these two methods. We then propose a new range representation method, called "extended affine interval".

3.1. Classical interval

Classical interval (CI) was introduced in the 1960s by Moore [9] as an approach to putting bounds on rounding errors in mathematical computations. In CI, each quantity is represented by the set of all possible values. Formally, CI is defined as follows:

Definition 4: A **classical interval** of x is an interval $\overline{x} = [x_l, x_h]$ with $x_l \leqslant x \leqslant x_h$. The set of classical intervals is denoted by $\overline{R}$.

The result of CI arithmetic is also a CI that binds all possible results. In particular, CI arithmetic is evaluated as follows:

Definition 5: **CI arithmetic** consists of operations $\overline{\bullet} = \{\overline{+}\ , \overline{-}\ , \overline{\times}\ , \overline{\div}\}$ on pairs of CIs defined below:

$$\begin{aligned} [x_l,\ x_h]\ \overline{+}\ [y_l,\ y_h] &= [x_l + y_l,\ x_h + y_h] \\ [x_l,\ x_h]\ \overline{-}\ [y_l,\ y_h] &= [x_l - y_h,\ x_h - y_l] \\ [x_l,\ x_h]\ \overline{\times}\ [y_l,\ y_h] &= [min(x_l y_l, x_l y_h, x_h y_l, x_h y_h), \\ &\qquad max(x_l y_l, x_l y_h, x_h y_l, x_h y_h)] \\ [x_l, x_h]\ \overline{\div}\ [y_l, y_h] &= [x_l, x_h]\ \overline{\times}\ [\tfrac{1}{y_h}, \tfrac{1}{u_l}] \text{ if } 0 \notin [y_l, y_h] \end{aligned}$$

For $\overline{x},\ \overline{x}_1,\ \ldots,\ \overline{x}_n \in \overline{R}$, $\overline{\circ} \in \overline{\bullet}$, and a constant c, we denote:

- $\overline{x}_1 \overline{x}_2 = \overline{x}_1 \overline{\times} \overline{x}_2$, $c\overline{x} = \overline{x}c = \overline{x}\ \overline{\times}\ [c,\ c]$,
- $c \circ \overline{x} = [c, c] \circ \overline{[x]}$, $\overline{x} \circ c = \overline{x} \circ [c, c]$, and
- $\sum_{i=1}^{n} \overline{x}_i = \overline{x}_1\ \overline{+}\ \overline{x}_2\ \overline{+}\ \cdots\ \overline{+}\ \overline{x}_n$.

CI assumes that all intervals are independent, even if their corresponding quantities are dependent. This assumption leads to a great loss of precision in a long computation chain, which is called "error explosion". The next example illustrates such a problem.

Example 2: Let b = 10, sp = 1 (sign), ip = 5, and fp = 3. For an arbitrary number t, the roundoff error is $\overline{t}_r = [-10^{-3}/2,\ 10^{-3}/2]$. It is easy to see that:

$$\begin{aligned} \overline{t}_r \overline{-} \overline{t}_r &= [-10^{-3}/2,\ 10^{-3}/2] \overline{-} [-10^{-3}/2,\ 10^{-3}/2] \\ &= [-10^{-3},\ 10^{-3}] \end{aligned}$$

CI arithmetic assumes the first operand and the second operand to be independent, while in fact, they represent the same quantity $\overline{t}_r$ and the result must be $[0, 0]$.

3.2. Affine interval

Affine interval (AI) was introduced by Stolfi [13], [14] as a model for self-validated numerical analysis. It was proposed to address the "error explosion" problem in conventional CI. Unlike CI, in AI, the quantities are represented as affine combinations (affine forms) of certain primitive variables, which stand for sources of uncertainty in the data or approximations made during the computation.

Definition 6: An **Affine interval** of x is a formula

$$\ddot{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n$$

with $x \in [x_0 - \sum_i^n |x_i|, x_0 + \sum_i^n |x_i|]$. x_0 is called the **central value**. For each $i \in [1, n], \varepsilon_i \in [-1, 1]$ is a **noise symbol**, which stands for an independent component of the total uncertainty. The set of affine interval forms is denoted by $\ddot{R}$.

In AI arithmetic, the results of linear operations (addition, subtraction) are straightforward operations on AIs. However, the results of nonlinear operations (multiplication, division) are not AI forms. Hence, we need to approximate the nonlinear parts of the results by introducing new noise symbols.

Definition 7: **AI arithmetic** consists of operations $\ddot{\bullet} = \{\ddot{+},\ \ddot{-},\ \ddot{\times},\ \ddot{\div}\}$ on pairs of AIs as defined below. Let $\ddot{x} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i$ and $\ddot{y} = y_0 + \sum_{i=1}^{n} y_i\varepsilon_i$. AI operations are as defined below:

$$\begin{aligned} \ddot{x}\ \ddot{+}\ \ddot{y} &= (x_0 + y_0) + \textstyle\sum_{i=1}^{n}(x_i + y_i)\varepsilon_i \\ \ddot{x}\ \ddot{-}\ \ddot{y} &= (x_0 - y_0) + \textstyle\sum_{i=1}^{n}(x_i - y_i)\varepsilon_i \\ \ddot{x}\ \ddot{\times}\ \ddot{y} &= (x_0 + \textstyle\sum_{i=1}^{n} x_i\varepsilon_i)\ \ddot{\times}\ (y_0 + \textstyle\sum_{i=1}^{n} y_i\varepsilon_i) \\ &= x_0 y_0 + \textstyle\sum_{i=1}^{n}(x_0 y_i + x_i y_0)\varepsilon_i + B\varepsilon_{n+1} \end{aligned}$$

where $\varepsilon_{n+1} \in [-1, 1]$ is a new noise symbol, and B is the maximum value of $(\sum_{i=1}^{n} x_i\varepsilon_i)(\sum_{i=1}^{n} y_i\varepsilon_i)$. An easy approximation of B is $(\sum_{i=1}^{n} |x_i|)(\sum_{i=1}^{n} |y_i|)$

$$\ddot{x}\ \ddot{\div}\ \ddot{y} = \ddot{x}\ \ddot{\times}\ (\tfrac{1}{\ddot{y}}), \text{ if } 0 \notin [x_0 - \textstyle\sum_i^n |x_i|, x_0 + \sum_i^n |x_i|],$$

where $\frac{1}{\ddot{y}}$ is computed by Chebyshev approximation [13].

AI is more precise than CI for linear operations, as shown in the following example.

Example 3: Let us consider base, fixed point format, and number t as in Example 2. Hence, the roundoff error is

$\ddot{t}_r = 0 + (10^{-3}/2)\varepsilon$. It is easy to see that:
$\ddot{t}_r \ddot{-} \ddot{t}_r = (0-0) + (10^{-3}/2 - 10^{-3}/2)\varepsilon = 0$.
This result is the correct result of the subtraction $(t_r - t_r)$.

In AI arithmetic, each time we perform a nonlinear operation, we introduce a new noise symbol, which is problematic for a program with a large number of nonlinear operations.

3.3. Extended affine interval

To deal with the limits of the above two methods, we propose a new interval called *extended affine interval* (EAI). EAI is extended from AI by assigning for each noise symbol one CI coefficient, which allows EAI multiplication *without* new noise symbols.

Definition 8: An **extended affine interval** of x is a formula

$$\hat{x} = \overline{x}_0 \mathbin{\overline{+}} \sum_{k=1}^{n} \overline{x}_k \varepsilon_k$$

with $x \in \overline{x}_0 \mathbin{\overline{+}} \sum_{k=1}^{n} \overline{x}_k[-1,1]$, where $\varepsilon_i \in [-1,1]$ is a noise symbol for each $i \in [1,n]$ and $\overline{x}_j \in \overline{R}$ for each $j \in [0,n]$. The set of extended affine intervals is denoted by $\hat{R}$.

The linear operations of EAI arithmetic are designed similarly to those of AI arithmetic. For nonlinear operations, unlike AI, EAI arithmetic does not need to introduce new noise symbols. The results of nonlinear operations are guaranteed to be EAIs by approximating nonlinear parts. For example, let us consider the multiplication of two EAIs. Let $\hat{x} = \overline{x}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{x}_i \varepsilon_i$, $\hat{y} = \overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{y}_i \varepsilon_i$. Without loss of generality, assume that $\sum_{k=1}^{n} \overline{y}_k[-1,1] \sqsubseteq \sum_{k=1}^{n} \overline{x}_k[-1,1]$. We have: $\hat{x} \mathbin{\hat{\times}} \hat{y} = (\overline{x}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{x}_i \varepsilon_i) \mathbin{\hat{\times}} (\overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{y}_i \varepsilon_i)$ $= \overline{x}_0\overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^{n}(\overline{x}_0\overline{y}_i \mathbin{\overline{+}} \overline{x}_i\overline{y}_0 \mathbin{\overline{+}} \overline{x}_i B)\varepsilon_i$, where $B = \sum_{i=1}^{n} \overline{y}_i \varepsilon_i$. An easy approximation of B is $\sum_{k=1}^{n} \overline{y}_k[-1,1]$. Formally, EAI arithmetic is defined as follows:

Definition 9: **EAI arithmetic** consists of operations $\hat{\bullet} = \{\hat{+},\ \hat{-},\ \hat{\times},\ \hat{\div}\}$ on pairs of EAI as defined below. Let $\hat{x} = \overline{x}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{x}_i \varepsilon_i$ and $\hat{y} = \overline{y}_0 \mathbin{\overline{+}} \sum_{i=1}^{n} \overline{y}_i \varepsilon_i$. The EAI operations are defined below:
$\hat{x} \mathbin{\hat{+}} \hat{y} = (\overline{x}_0 \mathbin{\overline{+}} \overline{y}_0) \mathbin{\overline{+}} \sum_{i=1}^{n}(\overline{x}_i \mathbin{\overline{+}} \overline{y}_i)\varepsilon_i$
$\hat{x} \mathbin{\hat{-}} \hat{y} = (\overline{x}_0 \mathbin{\overline{-}} \overline{y}_0) \mathbin{\overline{+}} \sum_{i=1}^{n}(\overline{x}_i \mathbin{\overline{-}} \overline{y}_i)\varepsilon_i$
$\hat{x} \mathbin{\hat{\times}} \hat{y} =$

$$\begin{cases} \overline{x}_0\overline{y}_0 \overline{+} \sum_{i=1}^{n}(\overline{x}_0\overline{y}_i \overline{+} \overline{x}_i\overline{y}_0 \overline{+} \overline{x}_i \sum_{k=1}^{n} \overline{y}_k[-1,1])\varepsilon_i \quad \text{if} \\ \quad \sum_{k=1}^{n} \overline{y}_k[-1,1] \sqsubseteq \sum_{k=1}^{n} \overline{x}_k[-1,1] \\ \overline{x}_0\overline{y}_0 \overline{+} \sum_{i=1}^{n}(\overline{x}_0\overline{y}_i \overline{+} \overline{x}_i\overline{y}_0 \overline{+} \overline{y}_i \sum_{k=1}^{n} \overline{x}_k[-1,1])\varepsilon_i \ \text{otherwise} \end{cases}$$

$\hat{x} \mathbin{\hat{\div}} \hat{y} = \hat{x} \mathbin{\hat{\times}} (\frac{1}{\hat{y}}) \quad$ if $0 \notin \overline{x}_0 \mathbin{\overline{+}} \sum_{k=1}^{n} \overline{x}_k[-1,1]$ where $\frac{1}{\hat{y}}$ is computed by Chebyshev approximation [13].

Similar to AI arithmetic, the commutative property holds for both addition and multiplication; the associative property only holds for addition; and the distributive property does not hold.

Although EAI does not introduce new noise symbols, this does not mean EAI arithmetic is always less precise than AI arithmetic. AI arithmetic only advances in cases when we reuse the results of some nonlinear parts. Let us consider the example below:

Example 4: Let $y = x \times x, z = y - y$ and the initial bound of x be $[-1,1]$. The bound of z is computed based on AI and EAI arithmetics as follows:

- AI arithmetic: $\ddot{x} = \varepsilon_1,\ \ddot{y} = \varepsilon_1 \ddot{\times} \varepsilon_1 = \varepsilon_2$ where ε_2 is introduced for multiplication. $\ddot{z} = \varepsilon_2 \ddot{-} \varepsilon_2 = 0$
- EAI arithmetic: $\hat{x} = \varepsilon_1,\ \hat{y} = \varepsilon_1 \hat{\times} \varepsilon_1 = [-1,1]\varepsilon_1,\ \hat{z} = [-1,1]\varepsilon_1 \hat{-} [-1,1]\varepsilon_1 = [-2,2]\varepsilon_1$.

The bound of $\ddot{z}$, [0,0], lies within the bound of $\hat{z}$, [-2,2]. So, AI arithmetic is more precise in this case.
However, if we compute the bound of $t = x \times x - x \times x$ without reusing the multiplication $x \times x$, then both AI arithmetic and EAI arithmetic return the same bound.
$\ddot{t}$, $\hat{t}$ *can be computed in a similar way. We omit the details of these computations here due to space limitations.*

4. Abstract domain for ORE problem

4.1. Abstract domain

The abstract value of a variable aims to cover all of its possible values at one program location. For the ORE problem, the abstract value is a pair of fixed point and roundoff error ranges. We will show three kinds of abstractions based on CI, AI, and EAI range representations.

Definition 10: Let fxp and rdf be corresponding range representations of fixed point and roundoff error.

- **CI abstract domain** $\overline{\Phi} = \{(fxp, rdf) | fxp, rdf \in \overline{R}\}$
- **AI abstract domain** $\ddot{\Phi} = \{(fxp, rdf) | fxp, rdf \in \ddot{R}\}$
- **EAI abstract domain** $\hat{\Phi} = \{(fxp, rdf) | fxp, rdf \in \hat{R}\}$

For a fresh symbol $\perp$ (which stands for *undefined* or *uninitialized*), we define $\Phi_\perp = \Phi \cup \{\perp\}$.

The following example illustrates how to initial EAI abstract values.

Example 5: Assume that we use fixed point format $sp = 1, ip = 7, fp = 8)$*, base* $b = 2$*,* $x \in [0,2]$*, and* $y \in [1,3]$*.*

Since $fp = 8$*, the initial roundoff error of* x *lies within the range* $[-2^{-9}, 2^{-9}]$*. Let* $(\hat{x}_f, \hat{x}_r)$*,* $(\hat{y}_f, \hat{y}_r) \in \hat{\Phi}$ *be EAI abstractions of* x *and* y*. We have:*

- $\hat{x}_f = [1,1] \mathbin{\overline{+}} [1,1]\varepsilon_{fx}$ *and* $\hat{x}_r = [2^{-9}, 2^{-9}]\varepsilon_{rx}$
- $\hat{y}_f = [2,2] \mathbin{\overline{+}} [1,1]\varepsilon_{fy}$ *and* $\hat{y}_r = [2^{-9}, 2^{-9}]\varepsilon_{ry}$

4.2. Abstract arithmetic

Abstract arithmetic aims to propagate both fixed point ranges and roundoff error ranges of variables.

Definition 11: Replacing (x_f, x_r), (y_f, y_r), $\boxed{*}$, and ϵ in the definition of FE arithmetic (Definition 2) with

- $(\overline{x}_f, \overline{x}_r)$, $(\overline{y}_f, \overline{y}_r)$, $\overline{\boxast} = \{\overline{\boxplus}, \overline{\boxminus}, \overline{\boxtimes}, \overline{\boxdiv}\}$, and $\overline{\epsilon}$, we obtain **CI abstract arithmetic**,
- $(\ddot{x}_f, \ddot{x}_r)$, $(\ddot{y}_f, \ddot{y}_r)$, $\ddot{\boxast} = \{\ddot{\boxplus}, \ddot{\boxminus}, \ddot{\boxtimes}, \ddot{\boxdiv}\}$, and $\ddot{\epsilon}$, we obtain **AI abstract arithmetic**, and
- $(\widehat{x}_f, \widehat{x}_r)$, $(\widehat{y}_f, \widehat{y}_r)$, $\widehat{\boxast} = \{\widehat{\boxplus}, \widehat{\boxminus}, \widehat{\boxtimes}, \widehat{\boxdiv}\}$, and $\widehat{\epsilon}$, we obtain **EAI abstract arithmetic**,

where

$$\begin{cases} \overline{\epsilon} = \widehat{\epsilon} = [b^{-fp}/2, b^{-fp}/2] \\ \ddot{\epsilon} = (b^{-fp}/2)\varepsilon_r \text{ with a fresh noise symbol } \varepsilon_r \end{cases}$$

For example, let $(\widehat{x}_f, \widehat{x}_r)$, $(\widehat{y}_f, \widehat{y}_r) \in \widehat{\Phi}$. The EAI abstract operations are evaluated as follows:

$$\begin{aligned}
(\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{\boxplus}} (\widehat{y}_f, \widehat{y}_r) &= (\widehat{x}_f \mathbin{\widehat{+}} \widehat{y}_f,\ \widehat{x}_r \mathbin{\widehat{+}} \widehat{y}_r) \\
(\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{\boxminus}} (\widehat{y}_f, \widehat{y}_r) &= (\widehat{x}_f \mathbin{\widehat{-}} \widehat{y}_f,\ \widehat{x}_r \mathbin{\widehat{-}} \widehat{y}_r) \\
(\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{\boxtimes}} (\widehat{y}_f, \widehat{y}_r) &= (\widehat{x}_f \mathbin{\widehat{\times}} \widehat{y}_f, \\
&\qquad \widehat{x}_f \widehat{\times} \widehat{y}_r \widehat{+} \widehat{x}_r \widehat{\times} \widehat{y}_f \widehat{+} \widehat{x}_r \widehat{\times} \widehat{y}_r \widehat{+} \widehat{\epsilon}) \\
(\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{\boxdiv}} (\widehat{y}_f, \widehat{y}_r) &= (\widehat{x}_f \mathbin{\widehat{\div}} \widehat{y}_f, \\
&\qquad (\widehat{x}_f \widehat{+} \widehat{x}_r) \widehat{\div} (\widehat{y}_f \widehat{+} \widehat{y}_r) \widehat{-} \widehat{x}_f \widehat{\div} \widehat{y}_f \widehat{+} \widehat{\epsilon})
\end{aligned}$$

To illustrate how abstract values are evaluated, we show the following example:

Example 6: Let us consider $(\widehat{x}_f, \widehat{x}_r)$, $(\widehat{y}_f, \widehat{y}_r)$ as in Example 5. Then, EAI abstract addition $(\widehat{z}_f, \widehat{z}_r) = (\widehat{x}_f, \widehat{x}_r) \mathbin{\widehat{\boxplus}} (\widehat{y}_f, \widehat{y}_r)$ *is evaluated as follows:*

$$\begin{aligned}
\widehat{z}_f &= \widehat{x}_f \mathbin{\widehat{+}} \widehat{y}_f \\
&= ([1,1] \mathbin{\overline{+}} [1,1]\varepsilon_{fx}) \mathbin{\widehat{+}} ([2,2] \mathbin{\overline{+}} [1,1]\varepsilon_{fy}) \\
&= [3,3] \mathbin{\overline{+}} [1,1]\varepsilon_{fx} \mathbin{\overline{+}} [1,1]\varepsilon_{fy}
\end{aligned}$$

We then get the range of $\widehat{z}_f$ as $[1,5]$ by replacing $\varepsilon_{fx}, \varepsilon_{fy}$ with $[-1,1]$.

Similarly, $\widehat{z}_r = [2^{-9}, 2^{-9}]\varepsilon_{rx} \mathbin{\overline{+}} [2^{-9}, 2^{-9}]\varepsilon_{ry}$, and the range of $\widehat{z}_r$ is $[-2^{-8}, 2^{-8}]$.

Therefore the fixed point value of z is bounded by $[1,5]$ and the roundoff error of t is bound by $[-2^{-8}, 2^{-8}]$.

4.3. Abstract comparison operations

Instead of nondeterministic transitions at a conditional branch, the conditional expression can often be evaluated by using abstract environment. This is useful in avoiding unnecessary execution paths. The abstract comparison operations are defined by using FE comparisons as follows:

Definition 12: Replacing (x_f, x_r), (y_f, y_r), in the definition of $\tilde{x}$ in FE comparisons (Definition 3) with

- $(\overline{x}_f, \overline{x}_r)$, $(\overline{y}_f, \overline{y}_r)$, we obtain **CI abstract comparison operations** $\overline{\boxed{\triangle}} = \{\overline{\leqslant},\ \overline{=}\}$,
- $(\ddot{x}_f, \ddot{x}_r)$, $(\ddot{y}_f, \ddot{y}_r)$, we obtain **AI abstract comparison operations** $\ddot{\boxed{\triangle}} = \{\ddot{\leqslant},\ \ddot{=}\}$, and
- $(\widehat{x}_f, \widehat{x}_r)$, $(\widehat{y}_f, \widehat{y}_r)$, we obtain **EAI abstract comparison operations** $\widehat{\boxed{\triangle}} = \{\widehat{\leqslant},\ \widehat{=}\}$.

For example, let $(\widehat{x}_f, \widehat{x}_r)$, $(\widehat{y}_f, \widehat{y}_r) \in \widehat{\Phi}$, let $\overline{x}$ be range of $(\widehat{x}_f \mathbin{\widehat{+}} \widehat{x}_r)$, and let $\overline{y}$ be the range $(\widehat{y}_f \mathbin{\widehat{+}} \widehat{y}_r)$.
$(\widehat{x}_f, \widehat{x}_r) \leqslant (\widehat{y}_f, \widehat{y}_r)$ is evaluated as follows:

$$(\widehat{x}_f, \widehat{x}_r) \mathrel{\widehat{\leqslant}} (\widehat{y}_f, \widehat{y}_r) = \begin{cases} \textbf{true} & \text{if } \forall u \in \overline{x}\ \forall v \in \overline{y}.u \leqslant v \\ \textbf{false} & \text{if } \forall u \in \overline{x}\ \forall v \in \overline{y}.u > v \\ \textbf{unknown} & \text{otherwise} \end{cases}$$

The following example illustrates how to evaluate EAI abstract comparison $\widehat{\leqslant}$.

Example 7: Use $(\widehat{x}_f, \widehat{x}_r)$, and $(\widehat{y}_f, \widehat{y}_r)$ as in Example 5. $(\widehat{x}_f, \widehat{x}_r) \mathrel{\widehat{\leqslant}} (\widehat{y}_f, \widehat{y}_r)$ is evaluated as follows:

- $\widehat{x} = \widehat{x}_f \mathbin{\widehat{+}} \widehat{x}_r$
  $= [1,1] \mathbin{\overline{+}} \varepsilon_{fx} \mathbin{\overline{+}} [2^{-9}, 2^{-9}]\varepsilon_{rx}$
  The range of $\widehat{x}$ is $\overline{x} = [-2^{-9}, 2 + 2^{-9}]$.
- $\widehat{y} = \widehat{y}_f \mathbin{\widehat{+}} \widehat{y}_r$
  $= [2,2] \mathbin{\overline{+}} \varepsilon_{fy} \mathbin{\overline{+}} [2^{-9}, 2^{-9}]\varepsilon_{ry}$
  The range of $\widehat{y}$ is $\overline{y} = [1 + -2^{-9}, 3 + 2^{-9}]$.

Since $\overline{x} \cap \overline{y} = [2 + 2^{-9}, 3 + 2^{-9}]$, we can conclude that $(\widehat{x}_f, \widehat{x}_r) \mathrel{\widehat{\leqslant}} (\widehat{y}_f, \widehat{y}_r)$ is **unknown**.

4.4. Meet operation

At the meet of two paths in a program, we need to combine the results that are generated from these paths. The result of the meet must bind all input abstract values. We first consider how to compute the union of two ranges:

Definition 13: The unions of ranges are:

- *CI:* $[x_l, x_h] \overline{\cup} [y_l, y_h] = [min(x_l, y_l), max(x_h, y_h)]$.
- *AI:* $(u_0 + \sum_{i=1}^n u_i\varepsilon_i) \mathbin{\ddot{\cup}} (v_0 + \sum_{i=1}^n v_i\varepsilon_i) = (\frac{u_0+v_0}{2} + \frac{|u_0-v_0|}{2}\varepsilon_{n+1} + \sum_{i=1}^n t_i\varepsilon_i)$ where $\varepsilon_{n+1} \in [-1,1]$ is a new noise symbol and, for each i,

$$t_i = \begin{cases} u_i & \text{if } |u_i| > |v_i|, \\ v_i & \text{otherwise.} \end{cases}$$

- *EAI:* $(\overline{u}_0 \mathbin{\overline{+}} \sum_{i=1}^n \overline{u}_i\varepsilon_i) \mathbin{\hat{\cup}} (\overline{v}_0 \mathbin{\overline{+}} \sum_{i=1}^n \overline{v}_i\varepsilon_i) = (\overline{u}_0 \mathbin{\overline{\cup}} \overline{v}_0) \mathbin{\overline{+}} \sum_{i=1}^n (\overline{u}_i \mathbin{\overline{\cup}} \overline{v}_i)\varepsilon_i$.

Then, the result of meet operation is a pair of the union of fixed point ranges and the union of roundoff error ranges.

Definition 14: The meets in abstract values are:

- *CI meet:* $(\overline{x}_f, \overline{x}_r) \mathbin{\overline{\sqcup}} (\overline{y}_f, \overline{y}_r) = (\overline{x}_f \mathbin{\overline{\cup}} \overline{y}_f, \overline{x}_r \mathbin{\overline{\cup}} \overline{y}_r)$
- *AI meet:* $(\ddot{x}_f, \ddot{x}_r) \mathbin{\ddot{\sqcup}} (\ddot{y}_f, \ddot{y}_r) = (\ddot{x}_f \mathbin{\ddot{\cup}} \ddot{y}_f,\ \ddot{x}_r \mathbin{\ddot{\cup}} \ddot{y}_r)$
- *EAI meet:* $(\widehat{x}_f, \widehat{z}_r) \mathbin{\widehat{\sqcup}} (\widehat{y}_f, \widehat{y}_r) = (\widehat{x}_f \mathbin{\widehat{\cup}} \widehat{y}_f,\ \widehat{x}_r \mathbin{\widehat{\cup}} \widehat{y}_r)$

$\sqcup \in \{\overline{\sqcup}, \ddot{\sqcup}, \widehat{\sqcup}\}$ is extended to $\Phi_\perp \in \{\overline{\Phi}_\perp, \ddot{\Phi}_\perp, \widehat{\Phi}_\perp\}$ by $\perp \sqcup (x_f, x_r) = (x_f, x_r) \sqcup \perp = (x_f, x_r)$.

5. ORE analysis as weighted model checking

5.1. Weighted model checking

It has been suggested that the connections between program analysis and model checking are intimate [12]. That is, a program is first encoded into a model (*transition system*) by abstraction, and the program analysis problem then becomes model checking problem on the generated model.

Weighted model checking computes dataflow (or, an update of environments) by associating a *weight* to each transition in the model. It was originally proposed as weighted pushdown model checking [11]. For our purpose, it is enough to restrict the underlying pushdown system to a finite state transition system.

Definition 15: A **transition system** $\mathcal{P}$ is a triplet (P, Δ, s_0) in which

- P is a finite set of states,
- $\Delta \subseteq P \times P$ is a set of transitions, and
- $s_0 \in P$ is the initial state.

In weighted model checking, the weight domain must satisfy the conditions of the idempotent semiring.

Definition 16: A **idempotent semiring** is a quintuple $(D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, where $\mathbf{0}, \mathbf{1} \in D$ and $\oplus$, $\otimes$ are binary operators on D such that, for $a, b, c \in D$,

- $(D, \oplus)$ is a commutative monoid with the unit **0**,
- $(D, \otimes)$ is a monoid with the unit **1**,
- $\otimes$ distributes over $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$,
- $\oplus$ is idempotent, i.e., $a \oplus a = a$, and
- $\overline{0}$ is the zero element of $\otimes$, i.e., $a \otimes \mathbf{0} = \mathbf{0} \otimes a = \mathbf{0}$.

For program analysis, each element of a bounded idempotent semiring is regarded as follows:

- **0** stands for interruption of dataflow,
- **1** stands for the identity function (i.e., no state update),
- $\otimes$ is the composition of two successive dataflows, and
- $\oplus$ merges two dataflows at the meet of two transition sequences.

The *weighted transition system* is then defined as *a transition system "plus" a weight domain.*

Definition 17: A **weighted transition system** is a triplet $\mathcal{W} = (\mathcal{P}, S, f)$, where $\mathcal{P} = (P, \Delta, s_0)$ is a transition system, $\mathcal{S} = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a bounded idempotent semiring and $f : \Delta \to D$ is a map that assigns a weight to each transition.

Denote Δ^* be a set of all **sequences of transitions**. Let $\sigma = [r_1, \ldots, r_k] \in \Delta^*$. We define $v(\sigma) =_\Delta f(r_1) \otimes \ldots \otimes f(r_k)$. If σ is a transition sequence from a state c to a state c', we denote $c \Rightarrow^\sigma c'$. The set of all such sequences is denoted by $paths(c, c')$, i.e.,

$$paths(c, c') = \{\sigma \mid c \Rightarrow^\sigma c'\}$$

Weighted model checking finds the weight summary of $paths(c, c')$, which is the summation $\oplus_{\sigma \in paths(c,c')} v(\sigma)$.

If a cycle in a weighted model exists, $paths(c, c')$ becomes infinite. For termination of a weighted model checking, we need an idempotent semiring must be *bounded*.

Definition 18: An idempotent semiring is **bounded** if there are no infinite descending chains wrt $\sqsubseteq$, where $a \sqsubseteq b$ if and only if $a \oplus b = a$.

5.2. Weight domain for ORE problem

For an ORE problem, we abstract a concrete environment as an abstract environment by using range representations.

Definition 19: Let Var be the set of all variables of the program. An **abstract environment** at a program location is the set of functions $AbsEnv = \{Var \to \Phi_\perp^k\}$, where $k = |Var|$ and $\Phi_\perp \in \{\overline{\Phi}_\perp, \ddot{\Phi}_\perp, \hat{\Phi}_\perp\}$. We define the *zero environment* $e_\mathbf{0} \in AbsEnv$ by $e_\mathbf{0}(x) = \perp$ for $x \in Var$. Let $e, e' \in AbsEnv$, and **environment meet operation** is defined below:

$$e \sqcup e' = \lambda x.e(x) \sqcup e'(x)$$

where $\sqcup \in \{\overline{\sqcup}, \ddot{\sqcup}, \hat{\sqcup}\}$.

Weight design. The standard definition of a weight domain has the base set of weights $D = AbsEnv \to AbsEnv$. We then theoretically define the weight domain for D as follows:

Definition 20: The weight domain (bounded idempotent semiring) $\mathcal{S} = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ with

$$\begin{aligned}
&D = AbsEnv \to AbsEnv,\\
&\mathbf{1} = \lambda x.x,\\
&\mathbf{0} = \lambda x.e_\mathbf{0},\\
&w_1 \oplus w_2 = \begin{cases} \lambda x.w_1(x) \sqcup w_2(x) & \text{if } w_1, w_2 \neq \mathbf{0} \\ w_1 & \text{if } w_2 = \mathbf{0} \\ w_2 & \text{if } w_1 = \mathbf{0} \end{cases}\\
&w_1 \otimes w_2 = \begin{cases} w_2 \cdot w_1 & \text{if } w_1, w_2 \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise} \end{cases}
\end{aligned}$$

where $\sqcup \in \{\overline{\sqcup}, \ddot{\sqcup}, \hat{\sqcup}\}$.

However, this does not satisfy the descending chain condition (boundedness), since intervals are infinitely many (thus the abstract domain is infinite). To cope with this problem, we:

- restrict the models to be acyclic,
- fix an initial abstract environment I, and
- generate weight on-the-fly.

In the context of our ORE analysis, the intuition behind the first two is,

- a target program has bounded loops only; thus after unfolding loops, abstraction produces an acyclic transition system, and
- the result of ORE analysis depends heavily on the input value; we will set a possible range of inputs at the program entry in advance.

On-the-fly weight generation. We first introduce the augmented weight domain to associate an input abstract environment to each weight. "_" means any input.

Definition 21: The *augmented weight domain* $\mathcal{S}^+ = (D^+, \oplus, \otimes, \mathbf{0}^+, \mathbf{1}^+)$ consists of $D^+ = \{(W, w) \mid W \in \mathit{AbsEnv}, w \in D\}$, $\mathbf{0}^+ = (\_, \mathbf{0})$, $\mathbf{1}^+ = (\_, \mathbf{1})$, and

$$w_1^+ \oplus w_2^+ = \begin{cases} (W_1, w_1 \oplus w_2) & \text{if } W_1 = W_2 \\ \mathbf{0}^+ & \text{otherwise} \end{cases}$$

$$w_1^+ \otimes w_2^+ = \begin{cases} (W_2, w_1 \otimes w_2) & \text{if } W_1 = w_2(W_2) \\ \mathbf{0}^+ & \text{otherwise} \end{cases}$$

for $w_1^+ = (W_1, w_1), w_2^+ = (W_2, w_2) \in D^+$.

Now we are ready to define the on-the-fly weight domain $\mathcal{S}^+_{\mathcal{P},I}$ for a transition system $\mathcal{P}$ and $I \in \mathit{AbsEnv}$. The intuition is, starting from the initial abstract environment I, only reachable instances of weights are computed in on-the-fly manner.

Definition 22: For a transition system $\mathcal{P}$ and $I \in \mathit{AbsEnv}$, the weight domain $\mathcal{S}^+_{\mathcal{P},I} = (D^+_{\mathcal{P},I}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a sub semiring of $\mathcal{S}^\times$ with $D^+_{\mathcal{P},I} \subseteq D^+$. $D^+_{\mathcal{P},I}$ is given by

$$\left\{ (W, w) \;\middle|\; \begin{array}{l} \exists \sigma, \sigma' \in \Delta^* \; \exists c, c' \in P.\; s_0 \Rightarrow^{\sigma} c \Rightarrow^{\sigma'} c' \\ \wedge W = v(\sigma)(I) \wedge w = v(\sigma') \end{array} \right\}$$

In implementation, we will identify $D^+ \subseteq \mathit{AbsEnv} \times D$ with $D^+ \subseteq \mathit{AbsEnv} \times \mathit{AbsEnv}$ by

$$(W, w) \equiv (W, w(W))$$

for $W \in \mathit{AbsEnv}, w \in D$.

5.3. Weighted transition system for ORE problem

The inputs of our analysis are a subclass of C programs that have bounded loops only. In preprocessing phase, C programs are transformed into three address codes. Next, analysis is performed on these three address codes. Basically, the instructions of three address codes include:

- **Assignment:** "$x = y \circ z$" with $\circ \in \{+, -, *, /\}$.
- **Conditional instruction:** "if $x \circ y$ then s" where s is an instruction and $\circ \in \{<, <=, >, >=, =, != \}$. If the condition $(x \circ y)$ is false, s is not visited; otherwise, s is visited.
- **Control instruction:** "return *loc*", "goto *loc*", "break", "continue". Control moves to the specified location, and the values of variables do not change.
- **While Loop:** "while $x \circ y$ { *body* }" with $\circ \in \{<, <=, >, >=, =, != \}$. *body* is repeated as long as the condition $(x \circ y)$ holds. Inside *body*, "break" will exit from the loop.

In preprocessing phase, the bounded while loops are unfolded as a sequence of conditional instructions. Thus, the generated transition system is *acyclic*.

instruction	weight
"$x = y \circ z$" (assignment)	$(W_i, \{x_o = y_i \boxdot z_i,\ v_o = v_i \vert v \in Var \backslash \{x\}\})$ where $\boxdot$ is the corresponding abstract arithmetic operation of $\circ$
"if $x \circ y$ then s" (Conditional instruction)	$\mathbf{0}^+$ if $x_i \boxdot y_i =$ **false**; $\mathbf{1}^+$ otherwise, where $\boxdot$ is the corresponding abstract comparison of $\circ$
Control instructions	$\mathbf{1}^+$

Table 1. Weight function of ORE analysis

```
     main {
st1:  rst = x; i = 0;
st2:  while (i <= n) {
st3:  if (!(i<=n)){
st4:  break;
      }
st5:  __cil_tmp3 =(float)i;
      rst += __cil_tmp3;
      i++;
st6:  rst /= 10.0f;
      __cil_tmp4 = x * x;
      rst -= __cil_tmp4;
st7:  return (1); st8:  }
```

Figure 2. CIL code for Example 1

The weight function is defined as follows:

Definition 23: For an acyclic transition system $\mathcal{P}$ and $I \in \mathit{AbsEnv}$, the weight function $f_{\mathcal{P},I} : \Delta \to D^+_{\mathcal{P},I}$ is given in Table 1.

As a result, we obtain the weighted transition system:

$$\mathcal{W} = (\mathcal{P}, \mathcal{S}^+_{\mathcal{P},I}, f_{\mathcal{P},I})$$

The following example describes how to create the weighted transition system for the program in Example 1.

Example 8: We use EAI range representation type. The three address codes and control flow graph (CFG) of the C program in Example 1 are shown in Figure 2, and in Figure 3. $st1, ..., st8$ are locations.

The transition system is $\mathcal{P} = (P, \Delta)$, where $P = \{st1, st2..., st8\}$, Δ and f is defined in Table 2.

The initial abstract environment W_{init} at $st1$ is generated from initial range values of variables (given in the topmost comments in Example 1) as:

$$\begin{cases} W_{init}(x) = ([0.5, 0.5] \mp [0.5, 0.5]\varepsilon_{xf},\ [2^{-9}, 2^{-9}]\varepsilon_{xr}) \\ W_{init}(n) = ([100, 100],\ [0, 0]) \\ W_{init}(v) = ([0, 0],\ [0, 0]) \qquad \text{if } v \notin \{x, n\} \end{cases}$$

Then, the resulting weighted transition system is $\mathcal{W} = (\mathcal{P}, \mathcal{S}^+_{\mathcal{P},W_{init}}, f)$.

Since the abstraction is an over-approximation, we conclude soundness of ORE analysis.

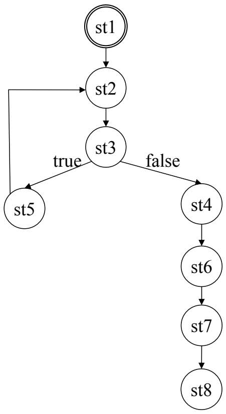


Figure 3. CFG of three address codes in Figure 2

transition	weight
(st1,st2)	$(W_{init},\ \{rst_o = x_{init}, i_o = ([0,0],[0,0]),$ $v_o = v_i \vert\ v \in Var \backslash \{rst, i\}\})$
(st2,st3)	$\mathbf{1}^+$
(st3,st5)	$if\ ((i_i \widehat{<} n_i) = false)\ \mathbf{0}^+\ else\ \mathbf{1}^+$
(st3,st4)	$if\ not((i_i \widehat{<} n_i) = false)\ \mathbf{0}^+\ else\ \mathbf{1}^+$
(st5,st2)	$(W_i,\ \{\_\_cil\_tmp3_o = i_i,$ $rst_o = rst_i\ \widehat{\boxplus}\ \_\_cil\_tmp3_i,$ $i_o = i_i\ \widehat{\boxplus}\ 1,$ $v_o = v_i \vert\ v \in Var \backslash \{\_\_cil\_tmp3, rst, i\}\})$
(st4,st6)	$\mathbf{1}^+$
(st6,st7)	$(W_i,\ \{\_\_cil\_tmp4_o = x_i \widehat{\boxtimes} x_i,$ $rst_o = rst_i\ \widehat{\boxed{\div}}\ 10\ \widehat{\boxminus}\ x_i\ \widehat{\boxtimes}\ x_i,$ $v_o = v_i \vert\ v \in Var \backslash \{\_\_cil\_tmp4, rst\}\})$
(st7,st8)	$\mathbf{1}^+$

Table 2. Weight function for a CIL code in Example 8

Theorem 1: For a C (CIL) program with bounded loops only, ORE analysis is sound.

6. Experiments

6.1. Implementation

We have implemented our analysis framework in a tool *C ANAlyzer* (CANA). CANA uses two libraries as back-end engines: CIL library [2] and WPDS library [3].

- CIL (C Intermediate Language) is a high-level representation along with tools that permit source-to-source transformation of C programs. CIL is used to generate three address codes, information about variables, and CFG of C program.
- WPDS (Weighted Pushdown System) is a library, which provides functions to the sets of forward- or backward- reachable configurations in a weighted pushdown system. Since we exclude procedure calls and unbounded loops at the moment, we adopt WPDS only for weighted finite state (and acyclic) transition systems (i.e., weighted pushdown system with empty stack).

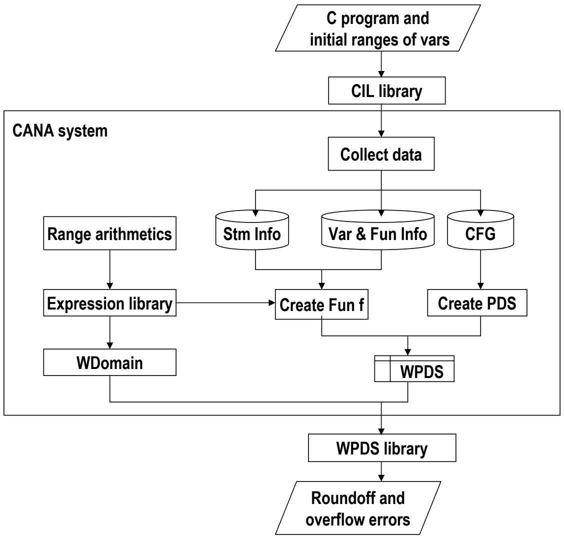


Figure 4. CANA system

The inputs of CANA are subclass of ANSI C programs and initial ranges of variables. The outputs of CANA are roundoff error ranges of variables at each point of the program, and warning about overflow errors (if they occur). CANA has six main modules (Figure 4) as follows:

1) *Collect data* module generates information required for analysis, including: statement information (Stm Info), (2) variable and function information (Var and Func Info), and (3) CFG of C program.
2) *Range arithmetics* module includes three types of range arithmetics: CI arithmetic, AI arithmetic, and EAI arithmetic.
3) *Evaluate exps* module evaluates the abstract values of expressions based on types of range arithmetics.
4) *Create PDS* module generates transition system from control flow graph of C program.
5) *Create Fun f* module assigns a weight to each transition.
6) *WDomain* module includes two operations: ⊗ and ⊕.

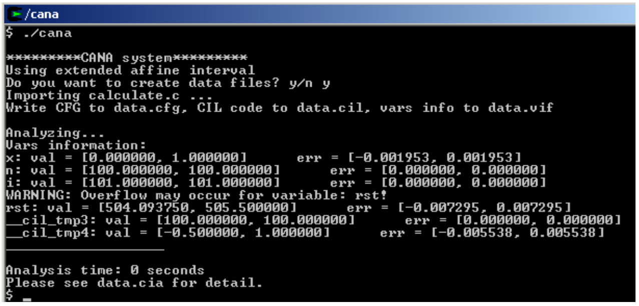
6.2. Experimental Results

We have implemented in CANA three types of range representations: CI, AI, and EAI. CANA can analyze programs that have nested loops 64×64.

Example 9: Figure 5 shows the result of analyzing the C program in Example 1. This shows that the roundoff error of rst lies within $[-0.007295, 0.007295]$. This range satisfies the condition that the roundoff error of rst lie within $[-0.01, 0.01]$.

[2] http://hal.cs.berkeley.edu/cil/

[3] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/



```
$ ./cana

*********CANA system*********
Using extended affine interval
Do you want to create data files? y/n y
Importing calculate.c ...
Write CFG to data.cfg, CIL code to data.cil, vars info to data.vif

Analyzing...
Vars information:
x: val = [0.000000, 1.000000]      err = [-0.001953, 0.001953]
n: val = [100.000000, 100.000000]      err = [0.000000, 0.000000]
i: val = [101.000000, 101.000000]      err = [0.000000, 0.000000]
WARNING: Overflow may occur for variable: rst!
rst: val = [504.093750, 505.500000]      err = [-0.007295, 0.007295]
__cil_tmp3: val = [100.000000, 100.000000]      err = [0.000000, 0.000000]
__cil_tmp4: val = [-0.500000, 1.000000]      err = [-0.005538, 0.005538]
___________________

Analysis time: 0 seconds
Please see data.cia for detail.
$ _
```

Figure 5. The result of CANA for Example 1

An overflow error may occur for variable rst, because the fixed point value of rst lies within $[504.093750, 505.500000]$ where the largest fixed point number which can be represented must be less than $2^7 = 128$.

In order to compare the efficiency of EAI arithmetic to CI, and AI arithmetics, we analyzed source codes of three examples: the first example is a C program that computes a polynomial of degree 5, the second is a program that calculates the sine function, and the last is a tiny fragment, which frequently appears in the mpeg decoder reference algorithm. Figure 6 shows experimental results of analyzing these programs. Figure 6a shows results of analyzing the program that computes $P5(x) = 1-x+3x^2-2x^3+x^4-5x^5$, where the fraction part $fp = 8$. The *true err* row is the width of roundoff error ranges which are found by testing method; the CI, AI, and EAI rows are the approximate widths of roundoff error ranges computed by using CI arithmetic, AI arithmetic, and EAI arithmetic, respectively. Our experiments show that EAI is more precise than CI and is comparable to AI. We get similar results for the program that computes sine of x shown in Figure 6b, and the part of mpeg decoder $pMpeg(exps)$ in Figure 6c. All tests run in less than 2 seconds.

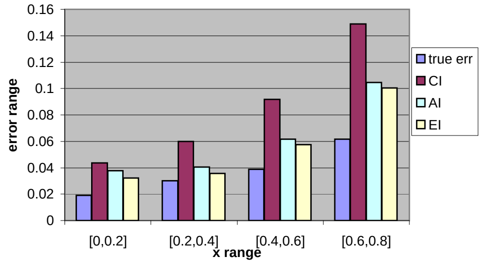
7. Related Work

To our limited knowledge, for ORE problem, there are not many works on abstract interpretation-based static analysis in literature. The works most closely related to ours are papers of Goubault and Putot [4], [5] and Martel [7], [8].

In [4], Goubault and Putot used AI arithmetic to approximate the roundoff error of ANSI C programs. This work focused on floating point numbers, not fixed point numbers.

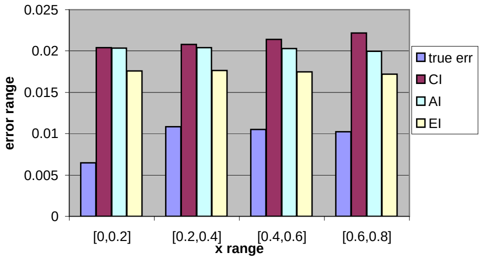
Goubault and Putot [5] introduced an under approximation method of computations in real numbers by using mean value theorem. Similar to our method, their method also does not introduce new noise symbols during computation.

[x]	[0,0.2]	[0.2,0.4]	[0.4,0.6]	[0.6,0.8]
real err	0.01909	0.03	0.03891	0.061588
CI	0.04373	0.0599	0.09172	0.148848
AI	0.0377	0.0406	0.06179	0.104468
EAI	0.03232	0.0356	0.05763	0.100454



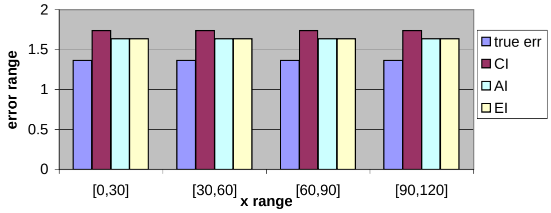
a. The analysis result of P5(x)

[x]	[0,0.2]	[0.2,0.4]	[0.4,0.6]	[0.6,0.8]
real err	0.00647	0.0108	0.01049	0.010206
CI	0.0204	0.0208	0.02139	0.022198
AI	0.02035	0.0204	0.02029	0.019976
EAI	0.01759	0.0176	0.01749	0.017184



b. The analysis result of Sin(x)

[exps]	[0,30]	[30,60]	[60,90]	[90,120]
real err	1.36309	1.3631	1.36309	1.363086
CI	1.73766	1.7377	1.73766	1.737656
AI	1.63453	1.6345	1.63453	1.634531
EAI	1.63453	1.6345	1.63453	1.634531



c. The analysis result of pMpeg(exps)

Figure 6. The experimental results

However, using mean value theorem requires more computations than our method. Also, their method aims to find the under-approximation, instead of over-approximation.

Martel [7], [8] introduced a method to build the roundoff error function of noise symbols by simply computing the addition, multiplication, substraction and division of functions. The procedure will stop whenever the degree of roundoff error function reaches the bound n. Therefore, roundoff error function is an nth degree function of noise symbols, while EAI arithmetic only returns linear function of input noise symbols.

8. Conclusion

In this paper, we analyzed the overflow and roundoff errors of C programs. In conclusion, our contributions are summarized as follows:

- An extended affine interval (EAI) was first proposed to estimate overflow and roundoff errors. EAI has two main advantages over current methods. First, EAI is more precise than CI because EAI can store information sources of uncertainty, whereas CI cannot. Second, EAI forms are more compact than AI forms. This is because EAI arithmetic does not introduce new noise symbols, while AI arithmetic does.
- We proposed an ORE analysis method based on weighted model checking. The range representations (i.e., CI, AI, and EAI) are used to create the set of weights. Next, the C program is modeled by weighted transition system (finite transition system + weight domain), where weight domain is generated in an on-the-fly manner. Finally, the ORE problem was reduced to checking reachability properties for the weighted transition system.
- A static analysis tool CANA for overflow and roundoff error analysis of subclass C programs was implemented. Although our experiments were performed on small examples, the result is encouraging that EAI is more precise than CI and is comparable to AI.

For future work, we plan to consider the following issues:

- Design a widening operator to analyze programs that have unbounded loops,
- Improve CANA to analyze C programs in practice, e.g., the endoder/decoder reference algorithms, and
- Combine our proposed technique with the simulation technique to improve precision and efficiency of ORE analysis.

Acknowledgements

This research is partially supported by STARC (Semiconductor Technology Academic Research Center). The authors would like to thank anonymous reviewers, Prof. Hiroyuki Seki, Dr. Nao Hirokawa, Dr. Xin Li, and Ms. Mary Ann Mooradian for their valuable comments.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Intl. Conf. POPL77*, pp. 238-252, Los Angeles, 1977.

[2] L. Giraud, J. Langou, M. Rozlonk, J. v. d. Eshof, Rounding error analysis of the classical Gram-Schmidt orthogonalization process, *Numerische Mathematik*, v.101 n.1, pp.87-100, July 2005

[3] D. Gopan. Numeric program analysis techniques with applications to array analysis and library summarization. PhD Thesis, University of Wisconsin-Madison, 2007

[4] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS'06*, pp. 18-34, LNCS 4134, Springer-Verlag 2006.

[5] E. Goubault and S. Putot. Under-approximations of computations in real numbers based on generalized affine arithmetic. In *Proc. Intl. Conf. SAS'07*, pp. 137-152, LNCS 4634, Springer-Verlag 2007.

[6] C. F. Fang, T. Chen, and R. Rutenbar. Floating-point error analysis based on affine arithmetic. In *Proc. 2003 International Conf. on Acoustic, Speech and Signal Processing*, pp.561-564, 2003.

[7] M. Martel. Propagation of Roundoff Errors in Finite Precision Computations: A Semantics Approach. In *11th European Symposium on Programming*, pp. 194-208, Springer-Verlag 2002.

[8] M. Martel. Semantics of roundoff error propagation in finite precision calculations. In *Higher-Order and Symbolic Computation*, v19(1),pp.7-30, Springer Netherlands 2006.

[9] R. E. Moore. Interval Analysis. *Prentice-Hall*, 1966.

[10] G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, In *Proc. Intl. Conf. on Compiler Construction (CC'02)*, pp. 213-228, LNCS 2304, Springer-Verlag 2002.

[11] T. Reps, S. Schwoon, S. Jha and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming 58*, pp. 206-263, No. 1-2, October 2005.

[12] D. A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Proc. Intl. Conf. SAS'98*, pp. 351380, LNCS 1503, Springe-Verlag 1998.

[13] J. Stolfi. Self-Validated Numerical Methods and Applications. Ph. D. Dissertation, Computer Science Department, Stanford University, 1997.

[14] J. Stolfi and L.H. de Figueiredo. An introduction to affine arithmetic. *TEMA Tend. Mat. Apl. Comput.*, No.4, Vol.3, pp. 297-312, 2003.