Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors*

Antoine Miné

DI-École Normale Supérieure de Paris, France,
mine@di.ens.fr

Abstract. We present a new idea to adapt relational abstract domains to the analysis of IEEE 754-compliant floating-point numbers in order to statically detect, through Abstract Interpretation-based static analyses, potential floating-point run-time exceptions such as overflows or invalid operations. In order to take the non-linearity of rounding into account, expressions are modeled as linear forms with interval coefficients. We show how to extend already existing numerical abstract domains, such as the octagon abstract domain, to efficiently abstract transfer functions based on interval linear forms. We discuss specific fixpoint stabilization techniques and give some experimental results.

1 Introduction

It is a well-established fact, since the failure of the Ariane 5 launcher in 1996, that run-time errors in critical embedded software can cause great financial—and human—losses. Nowadays, embedded software is becoming more and more complex. One particular trend is to abandon fixed-point arithmetics in favor of floating-point computations. Unfortunately, floating-point models are quite complex and features such as rounding and special numbers (infinities, *NaN*, etc.) are not always understood by programmers. This has already led to catastrophic behaviors, such as the Patriot missile story told in [16].

Much work is concerned about the *precision* of the computations, that is to say, characterizing the amount and cause of drift between a computation on perfect reals and the corresponding floating-point implementation. Ours is *not*. We seek to prove that an exceptional behavior (such as division by zero or overflow) will not occur in any execution of the analyzed program. While this is a simpler problem, our goal is to scale up to programs of hundreds of thousands of lines with full data coverage and very few (even none) false alarms.

Our framework is that of Abstract Interpretation, a generic framework for designing sound static analyses that already features many instances [6,7]. We adapt existing relational numerical abstract domains (generally designed for the analysis of integer arithmetics) to cope with floating-point arithmetics. The need for such domains appeared during the successful design of a commissioned special-purpose prototype analyzer for a critical embedded avionics

* This work was partially supported by the ASTRÉE RNTL project.

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 3–17, 2004.

system. Interval analysis, used in a first prototype [3], proved too coarse because error-freeness of the analyzed code depends on tests that are inherently poorly abstracted in non-relational abstract domains. We also had to design special-purpose widenings and narrowings to compensate for the pervasive rounding errors, not only in the analyzed program, but also introduced by our efficient abstractions. These techniques were implemented in our second prototype whose overall design was presented in [4]. The present paper focuses on improvements and novel unpublished ideas; it is also more generic.

2 Related Work

Abstract Domains. A key component in Abstract-Interpretation-based analyses is the abstract domain which is a computer-representable class of program invariants together with some operators to manipulate them: transfer functions for guards and assignments, a control-flow join operator, and fixpoint acceleration operators (such as widenings $\triangledown$ and narrowings $\triangle$) aiming at the correct and efficient analysis of loops. One of the simplest yet useful abstract domain is the widespread interval domain [6]. Relational domains, which are more precise, include Cousot and Halbwachs's polyhedron domain [8] (corresponding to invariants of the form $\sum c_i v_i \leq c$), Miné's octagon domain [14] ($\pm v_i \pm v_j \leq c$), and Simon's two variables per inequality domain [15] ($\alpha v_i + \beta v_j \leq c$). Even though the underlying algorithms for these relational domains allow them to abstract sets of reals as well as sets of integers, their efficient implementation—in a maybe approximate but sound way—using floating-point numbers remains a challenge. Moreover, these relational domains do not support abstracting floating-point expressions, but only expressions on perfect integers, rationals, or reals.

Floating-Point Analyses. Much work on floating-point is dedicated to the analysis of the precision of the computations and the origins of the rounding errors. The CESTAC method [17] is widely used, but also much debated as it is based on a probabilistic model of error distribution and thus cannot give sound answers. An interval-based Abstract Interpretation for error terms is proposed in [1]. Some authors [11,13] go one step further by allowing error terms to be related in relational, even non-linear, domains. Unfortunately, this extra precision does not help when analyzing programs whose correctness also depends upon relations between variables and not only error terms (such as programs with inequality tests, as in Fig. 3).

Our Work. We first present our IEEE 754-based computation model (Sect. 3) and recall the classical interval analysis adapted to floating-point numbers (Sect. 4). We present, in Sect. 5, an abstraction of floating-point expressions in terms of interval linear forms over the real field and use it to refine the interval domain. Sect. 6 shows how some relational abstract domains can be efficiently adapted to work on these linear forms. Sect. 7 presents adapted widening and narrowing techniques. Finally, some experimental results are shown in Sect. 8.

3 IEEE 754-Based Floating-Point Model

We present in this section the concrete floating-point arithmetics model that we wish to analyze and which is based on the widespread IEEE 754-1985 [5] norm.

3.1 IEEE 754 Floating-Point Numbers

The binary representation of a IEEE 754 number is composed of three fields:

- a 1-bit sign s;
- an exponent $e - \mathbf{bias}$, represented by a biased $\mathbf{e}$-bit unsigned integer e;
- a fraction $f = .b_1 \dots b_{\mathbf{p}}$, represented by a $\mathbf{p}$-bit unsigned integer.

The values $\mathbf{e}$, $\mathbf{bias}$, and $\mathbf{p}$ are format-specific. We will denote by $\mathbf{F}$ the set of all available formats and by $\mathbf{f} = \mathbf{32}$ the 32-bit *single* format ($\mathbf{e} = 8$, $\mathbf{bias} = 127$, and $\mathbf{p} = 23$). Floating-point numbers belong to one of the following categories:

- *normalized* numbers $(-1)^s \times 2^{2-\mathbf{bias}} \times 1.f$, when $1 \leq e \leq 2^{\mathbf{e}} - 2$;
- *denormalized* numbers $(-1)^s \times 2^{1-\mathbf{bias}} \times 0.f$, when $e = 0$ and $f \neq 0$;
- $+0$ or -0 (depending on s), when $e = 0$ and $f = 0$;
- $+\infty$ or $-\infty$ (depending on s), when $e = 2^{\mathbf{e}} - 1$ and $f = 0$;
- error codes (so-called *NaN*), when $e = 2^{\mathbf{e}} - 1$ and $f \neq 0$.

For each format $\mathbf{f} \in \mathbf{F}$ we define in particular:

- $mf_{\mathbf{f}} = 2^{1-\mathbf{bias}-\mathbf{p}}$ the smallest non-zero positive number;
- $Mf_{\mathbf{f}} = (2 - 2^{-\mathbf{p}})2^{2^{\mathbf{e}}-\mathbf{bias}-2}$, the largest non-infinity number.

The special values $+\infty$ and $-\infty$ may be generated as a result of operations undefined on $\mathbb{R}$ (such as $1/+0$), or when a result's absolute value overflows $Mf_{\mathbf{f}}$. Other undefined operations (such as $+0/+0$) result in a *NaN* (that stands for *Not A Number*). The sign of 0 serves only to distinguish between $1/+0 = +\infty$ and $1/-0 = -\infty$; $+0$ and -0 are indistinguishable in all other contexts (even comparison).

Due to the limited number of digits, the result of a floating-point operation needs to be rounded. IEEE 754 provides four rounding modes: towards 0, towards $+\infty$, towards $-\infty$, and to nearest. Depending on this mode, either the floating-point number directly smaller or directly larger than the exact real result is chosen (possibly $+\infty$ or $-\infty$). Rounding can build infinities from non-infinities operands (this is called *overflow*), and it may return zero when the absolute value of the result is too small (this is called *underflow*). Because of this rounding phase, most algebraic properties of $\mathbb{R}$, such as associativity and distributivity, are lost. However, the opposite of a number is always exactly represented (unlike what happens in two-complement integer arithmetics), and comparison operators are also exact. See [10] for a description of the classical properties and pitfalls of the floating-point arithmetics.

3.2 Custom Floating-Point Computation Model

We focus our analysis on the large class of programs that treat floating-point arithmetics as a practical approximation to the mathematical reals $\mathbb{R}$: roundings and underflows are tolerated, but not overflows, divisions by zero or invalid operations, which are considered run-time errors and halt the program. Our goal is to detect such behaviors. In this context, $+\infty$, $-\infty$, and *NaN*s can never

$$
\begin{aligned}
R_{\mathbf{f},+\infty}(x) &= \begin{cases} \Omega & \text{if } x > Mf_{\mathbf{f}} \\ \min\{y \in \mathbb{F}_{\mathbf{f}} \mid y \geq x\} & \text{otherwise} \end{cases} \\
R_{\mathbf{f},-\infty}(x) &= \begin{cases} \Omega & \text{if } x < -Mf_{\mathbf{f}} \\ \max\{y \in \mathbb{F}_{\mathbf{f}} \mid y \leq x\} & \text{otherwise} \end{cases} \\
R_{\mathbf{f},0}(x) &= \begin{cases} \min\{y \in \mathbb{F}_{\mathbf{f}} \mid y \geq x\} & \text{if } x \leq 0 \\ \max\{y \in \mathbb{F}_{\mathbf{f}} \mid y \leq x\} & \text{if } x \geq 0 \end{cases} \\
R_{\mathbf{f},n}(x) &= \begin{cases} \Omega & \text{if } |x| \geq (2 - 2^{-\mathbf{p}-1})2^{2^{\mathbf{e}}-\mathbf{bias}-2} \\ Mf_{\mathbf{f}} & \text{else if } x \geq Mf_{\mathbf{f}} \\ -Mf_{\mathbf{f}} & \text{else if } x \leq -Mf_{\mathbf{f}} \\ R_{\mathbf{f},-\infty}(x) & \text{else if } |R_{\mathbf{f},-\infty}(x) - x| < |R_{\mathbf{f},+\infty}(x) - x| \\ R_{\mathbf{f},+\infty}(x) & \text{else if } |R_{\mathbf{f},+\infty}(x) - x| < |R_{\mathbf{f},-\infty}(x) - x| \\ R_{\mathbf{f},-\infty}(x) & \text{else if } R_{\mathbf{f},-\infty}(x)\text{'s least significant bit is 0} \\ R_{\mathbf{f},+\infty}(x) & \text{else if } R_{\mathbf{f},+\infty}(x)\text{'s least significant bit is 0} \end{cases}
\end{aligned}
$$

Fig. 1. Rounding functions, extracted from [5].

$$
\begin{array}{lll}
[\![const_{\mathbf{f},\mathbf{r}}(c)]\!]\rho = R_{\mathbf{f},\mathbf{r}}(c) & & \\
[\![v]\!]\rho = \rho(v) & & \\
[\![cast_{\mathbf{f},\mathbf{r}}(e)]\!]\rho = R_{\mathbf{f},\mathbf{r}}([\![e]\!]\rho) & & \text{if } [\![e]\!]\rho \neq \Omega \\
[\![e_1 \odot_{\mathbf{f},\mathbf{r}} e_2]\!]\rho = R_{\mathbf{f},\mathbf{r}}(([\![e_1]\!]\rho) \cdot ([\![e_2]\!]\rho)) & & \text{if } [\![e_1]\!]\rho, [\![e_2]\!]\rho \neq \Omega,\ \cdot \in \{+,-,\times\} \\
[\![e_1 \oslash_{\mathbf{f},\mathbf{r}} e_2]\!]\rho = R_{\mathbf{f},\mathbf{r}}(([\![e_1]\!]\rho)/([\![e_2]\!]\rho)) & & \text{if } [\![e_1]\!]\rho \neq \Omega,\ [\![e_2]\!]\rho \notin \{0, \Omega\} \\
[\![\ominus e]\!]\rho = -([\![e]\!]\rho) & & \text{if } [\![e]\!]\rho \neq \Omega \\
[\![expr_{\mathbf{f}}]\!]\rho = \Omega & & \text{in all other cases}
\end{array}
$$

Fig. 2. Expression concrete semantics, extracted from [5].

be created and, as a consequence, the difference between $+0$ and -0 becomes irrelevant. For every format $\mathbf{f} \in \mathbf{F}$, the set of floating-point numbers will be assimilated to a finite subset of $\mathbb{R}$ denoted by $\mathbb{F}_{\mathbf{f}}$. The grammar of floating-point expressions of format $\mathbf{f}$ includes constants, variables $v \in \mathcal{V}_{\mathbf{f}}$ of format $\mathbf{f}$, casts (conversion from another format), binary and unary arithmetic operators (circled in order to distinguish them from the corresponding operators on reals):

$$
\begin{array}{rcll}
expr_{\mathbf{f}} & :== & const_{\mathbf{f},\mathbf{r}}(c) & c \in \mathbb{R} \\
& | & v & v \in \mathcal{V}_{\mathbf{f}} \\
& | & cast_{\mathbf{f},\mathbf{r}}(expr_{\mathbf{f}'}) & \\
& | & expr_{\mathbf{f}} \odot_{\mathbf{f},\mathbf{r}} expr_{\mathbf{f}} & \odot \in \{\oplus, \ominus, \otimes, \oslash\} \\
& | & \ominus expr_{\mathbf{f}} &
\end{array}
$$

Some constructs are tagged with a floating-point format $\mathbf{f} \in \mathbf{F}$ and a rounding mode $\mathbf{r} \in \{n, 0, +\infty, -\infty\}$ (n representing rounding to nearest). A notable exception is the unary minus $\ominus$ which does not incur rounding and never results in a run-time error as all the $\mathbb{F}_{\mathbf{f}}$s are perfectly symmetric.

An environment $\rho \in \prod_{\mathbf{f} \in \mathbf{F}} (\mathcal{V}_{\mathbf{f}} \to \mathbb{F}_{\mathbf{f}})$ is a function that associates to each variable a floating-point value of the corresponding format. Fig. 2 gives the concrete semantics $[\![expr_{\mathbf{f}}]\!]\rho \in \mathbb{F}_{\mathbf{f}} \cup \{\Omega\}$ of the expression $expr_{\mathbf{f}}$ in the envi-

ronment ρ: it can be a number or the run-time error Ω. This semantics uses solely the regular operators $+,-,\times,/$ on real numbers and the rounding function $R_{\mathbf{f},\mathbf{r}} : \mathbb{R} \to \mathbb{F}_{\mathbf{f}} \cup \{\Omega\}$ defined in Fig. 1. It corresponds exactly to the IEEE 754 norm [5] where the overflow, division by zero, and invalid operation exception traps abort the system with a run-time error.

4 Floating-Point Interval Analysis

Floating-Point Interval Arithmetics. The idea of interval arithmetics is to over-approximate a set of numbers by an interval represented by its lower and upper bounds. For each format $\mathbf{f}$, we will denote by $\mathbb{I}_{\mathbf{f}}$ the set of real intervals with bounds in $\mathbb{F}_{\mathbf{f}}$. As $\mathbb{F}_{\mathbf{f}}$ is totally ordered and bounded, any subset of $\mathbb{F}_{\mathbf{f}}$ can be abstracted by an element of $\mathbb{I}_{\mathbf{f}}$. Moreover, as *all rounding functions* $R_{\mathbf{f},\mathbf{r}}$ *are monotonic*, we can compute the bounds of any expression using pointing-point operations only, ensuring efficient implementation within an analyzer. A first idea is to use the very same format and rounding mode in the abstract as in the concrete, which would give, for instance, the following addition (Ω denoting a run-time error):

$$[a^-; a^+] \oplus^\sharp_{\mathbf{f},\mathbf{r}} [b^-; b^+] = \begin{cases} \Omega & \text{if } a^- \oplus_{\mathbf{f},\mathbf{r}} b^- = \Omega \text{ or } a^+ \oplus_{\mathbf{f},\mathbf{r}} b^+ = \Omega \\ [a^- \oplus_{\mathbf{f},\mathbf{r}} b^-; a^+ \oplus_{\mathbf{f},\mathbf{r}} b^+] & \text{otherwise} \end{cases}$$

A drawback of this semantics is that it requires the analyzer to determine, for each instruction, which rounding mode $\mathbf{r}$ and which format $\mathbf{f}$ are used. This may be difficult as the rounding mode can be changed at run-time by a system call, and compilers are authorized to perform parts of computations using a more precise IEEE format that what is required by the programmer (on Intel x86, all floating-point registers are 80-bit wide and rounding to the user-specified format occurs only when the results are stored into memory). Unfortunately, using in the abstract a floating-point format different from the one used in the concrete computation is not sound.

The following semantics, inspired from the one presented in [11], solves these problems by providing an approximation that is independent from the concrete rounding mode (assuming always the worst: towards $-\infty$ for the lower bound and towards $+\infty$ for the upper bound):

- $const^\sharp_{\mathbf{f}}(c) = [const_{\mathbf{f},-\infty}(c); const_{\mathbf{f},+\infty}(c)]$
- $cast^\sharp_{\mathbf{f}}(c) = [cast_{\mathbf{f},-\infty}(c); cast_{\mathbf{f},+\infty}(c)]$
- $[a^-; a^+] \oplus^\sharp_{\mathbf{f}} [b^-; b^+] = [a^- \oplus_{\mathbf{f},-\infty} b^-; a^+ \oplus_{\mathbf{f},+\infty} b^+]$
- $[a^-; a^+] \ominus^\sharp_{\mathbf{f}} [b^-; b^+] = [a^- \ominus_{\mathbf{f},-\infty} b^+; a^+ \ominus_{\mathbf{f},+\infty} b^-]$
- $[a^-; a^+] \otimes^\sharp_{\mathbf{f}} [b^-; b^+] =$
  $[\min((a^+ \otimes_{\mathbf{f},-\infty} b^+), (a^- \otimes_{\mathbf{f},-\infty} b^+), (a^+ \otimes_{\mathbf{f},-\infty} b^-), (a^- \otimes_{\mathbf{f},-\infty} b^-));$
  $\max((a^+ \otimes_{\mathbf{f},+\infty} b^+), (a^- \otimes_{\mathbf{f},+\infty} b^+), (a^+ \otimes_{\mathbf{f},+\infty} b^-), (a^- \otimes_{\mathbf{f},+\infty} b^-))]$
- $[a^-; a^+] \oslash^\sharp_{\mathbf{f}} [b^-; b^+] =$
  - Ω if $b^- \leq 0 \leq b^+$
  - $[\min((a^+ \oslash_{\mathbf{f},-\infty} b^+), (a^- \oslash_{\mathbf{f},-\infty} b^+), (a^+ \oslash_{\mathbf{f},-\infty} b^-), (a^- \oslash_{\mathbf{f},-\infty} b^-));$

```
for (n=0;n¡N;n++) {
  // fetch X in [−128; 128] and D in [0; 16]
  S = Y;    R = X ⊖_{32,n} S;    Y = X;
  if (R ≤ ⊖D) Y = S ⊖_{32,n} D;
  if (R ≥ D) Y = S ⊕_{32,n} D;
}
```

Fig. 3. Simple rate limiter function with input X, output Y, and maximal rate variation D.

$$\max((a^+ \oslash_{\mathbf{f},+\infty} b^+), (a^- \oslash_{\mathbf{f},+\infty} b^+), (a^+ \oslash_{\mathbf{f},+\infty} b^-), (a^- \oslash_{\mathbf{f},+\infty} b^-))]$$

otherwise

- $\ominus^\sharp[a^-; a^+] = [\ominus a^+; \ominus a^-]$
- return Ω if one interval bound evaluates to Ω

This semantics frees the analyzer from the job of statically determining the rounding mode of the expressions and allows the analyzer to use, in the abstract, less precise formats that those used in the concrete (however, using a more precise format in the abstract remains unsound).

Floating-Point Interval Analysis. Interval analysis is a non-relational Abstract Interpretation-based analysis where, at each program point and for each variable, the set of its possible values during all executions reaching this point is over-approximated by an interval. An abstract environment $\rho^\sharp$ is a function mapping each variable $v \in \mathcal{V}_\mathbf{f}$ to an element of $\mathbb{I}_\mathbf{f}$. The abstract value $[\![expr_\mathbf{f}]\!]^\sharp \rho^\sharp \in \mathbb{I}_\mathbf{f} \cup \{\Omega\}$ of an expression $expr_\mathbf{f}$ in an abstract environment $\rho^\sharp$ can be derived by induction using the interval operators defined in the preceding paragraph.

An assignment $v \leftarrow expr_\mathbf{f}$ performed in an environment $\rho^\sharp$ returns $\rho^\sharp$ where v's value has been replaced by $[\![expr_\mathbf{f}]\!]^\sharp \rho^\sharp$ if it does not evaluate to Ω, and otherwise by $\mathbb{F}_\mathbf{f}$ (the *top* value) and reports an error. Most tests can only be abstracted by ignoring them (which is sound). Even though for simple tests such as, for instance, $X \leq Y \oplus_{\mathbf{f},\mathbf{r}} c$, the interval domain is able to refine the bounds for X and Y, it cannot remember the relationship between these variables. Consider the more complete example of Fig. 4. It is a rate limiter that given random input flows X and D, bounded respectively by $[-128; 128]$ and $[0; 16]$, computes an output flow Y that tries to follow X while having a change rate limited by D. Due to the imprecise abstraction of tests, the interval domain will bound Y by $[-128 - 16n; 128 + 16n]$ after n loop iterations while in fact it is bounded by $[-128; 128]$ independently from n. If N is too big, the interval analysis will conclude that the limiter may overflow while it is in fact always perfectly safe.

5 Linearization of Floating-Point Expressions

Unlike the interval domain, relational abstract domains rely on algebraic properties of operators, such as associativity and distributivity, that are not true in the floating-point world. Our solution is to approximate floating-point expressions by *linear expressions* in the *real* field with *interval coefficients* and free variables in $\mathcal{V} = \cup_{\mathbf{f} \in \mathbf{F}} \mathcal{V}_\mathbf{f}$. Let $i + \sum_{v \in \mathcal{V}} i_v v$ be such a linear form; it can be viewed as a

function from $\mathbb{R}^{\mathcal{V}}$ to the set of real intervals. For the sake of efficiency, interval coefficient bounds will be represented by floating-point numbers in a format $\mathbf{fa}$ that is efficient on the analyzer's platform: $i, i_v \in \mathbb{I}_{\mathbf{fa}}$. Because, for all $\mathbf{f}$ and $\cdot \in \{+, -, \times, /\}$, $\odot^\sharp_{\mathbf{f}}$ is a valid over-approximation of the corresponding real interval arithmetics operation, we can define the following sound operators $\boxplus^\sharp$, $\boxminus^\sharp$, $\boxtimes^\sharp$, $\boxslash^\sharp$ on linear forms:

- $(i + \sum_{v\in\mathcal{V}} i_v v) \boxplus^\sharp (i' + \sum_{v\in\mathcal{V}} i'_v v) = (i \oplus^\sharp_{\mathbf{fa}} i') + \sum_{v\in\mathcal{V}} (i_v \oplus^\sharp_{\mathbf{fa}} i'_v)v$
- $(i + \sum_{v\in\mathcal{V}} i_v v) \boxminus^\sharp (i' + \sum_{v\in\mathcal{V}} i'_v v) = (i \ominus^\sharp_{\mathbf{fa}} i') + \sum_{v\in\mathcal{V}} (i_v \ominus^\sharp_{\mathbf{fa}} i'_v)v$
- $i \boxtimes^\sharp (i' + \sum_{v\in\mathcal{V}} i'_v v) = (i \otimes^\sharp_{\mathbf{fa}} i') + \sum_{v\in\mathcal{V}} (i \otimes^\sharp_{\mathbf{fa}} i'_v)v$
- $(i + \sum_{v\in\mathcal{V}} i_v v) \boxslash^\sharp i' = (i \oslash^\sharp_{\mathbf{fa}} i') + \sum_{v\in\mathcal{V}} (i_v \oslash^\sharp_{\mathbf{fa}} i')v$

Given an expression $expr_{\mathbf{f}}$ and an interval abstract environment $\rho^\sharp$ as in Sect. 4, we construct the interval linear form $(\!|\, expr_{\mathbf{f}} \,|\!)\rho^\sharp$ on $\mathcal{V}$ as follows:

- $(\!|\, const_{\mathbf{f},\mathbf{r}}(c) \,|\!)\rho^\sharp = [const_{\mathbf{f},-\infty}(c); const_{\mathbf{f},+\infty}(c)]$
- $(\!|\, v_{\mathbf{f}} \,|\!)\rho^\sharp = [1;1]v_{\mathbf{f}}$
- $(\!|\, cast_{\mathbf{f},\mathbf{r}}(e) \,|\!)\rho^\sharp = (\!|\, e \,|\!)\rho^\sharp \boxplus^\sharp \varepsilon_{\mathbf{f}}((\!|\, e \,|\!)\rho^\sharp) \boxplus^\sharp mf_{\mathbf{f}}[-1;1]$
- $(\!|\, e_1 \oplus_{\mathbf{f},\mathbf{r}} e_2 \,|\!)\rho^\sharp =$
  $(\!|\, e_1 \,|\!)\rho^\sharp \boxplus^\sharp (\!|\, e_2 \,|\!)\rho^\sharp \boxplus^\sharp \varepsilon_{\mathbf{f}}((\!|\, e_1 \,|\!)\rho^\sharp) \boxplus^\sharp \varepsilon_{\mathbf{f}}((\!|\, e_2 \,|\!)\rho^\sharp) \boxplus^\sharp mf_{\mathbf{f}}[-1;1]$
- $(\!|\, e_1 \ominus_{\mathbf{f},\mathbf{r}} e_2 \,|\!)\rho^\sharp =$
  $(\!|\, e_1 \,|\!)\rho^\sharp \boxminus^\sharp (\!|\, e_2 \,|\!)\rho^\sharp \boxplus^\sharp \varepsilon_{\mathbf{f}}((\!|\, e_1 \,|\!)\rho^\sharp) \boxplus^\sharp \varepsilon_{\mathbf{f}}((\!|\, e_2 \,|\!)\rho^\sharp) \boxplus^\sharp mf_{\mathbf{f}}[-1;1]$
- $(\!|\, [a;b] \otimes_{\mathbf{f},\mathbf{r}} e_2 \,|\!)\rho^\sharp =$
  $([a;b] \boxtimes^\sharp (\!|\, e_2 \,|\!)\rho^\sharp) \boxplus^\sharp ([a;b] \boxtimes^\sharp \varepsilon_{\mathbf{f}}((\!|\, e_2 \,|\!)\rho^\sharp)) \boxplus^\sharp mf_{\mathbf{f}}[-1;1]$
- $(\!|\, e_1 \otimes_{\mathbf{f},\mathbf{r}} [a;b] \,|\!)\rho^\sharp = (\!|\, [a;b] \otimes_{\mathbf{f},\mathbf{r}} e_1 \,|\!)\rho^\sharp$
- $(\!|\, e_1 \otimes_{\mathbf{f},\mathbf{r}} e_2 \,|\!)\rho^\sharp = (\!|\, \iota((\!|\, e_1 \,|\!)\rho^\sharp)\rho^\sharp \otimes_{\mathbf{f},\mathbf{r}} e_2 \,|\!)\rho^\sharp$
- $(\!|\, e_1 \oslash_{\mathbf{f},\mathbf{r}} [a;b] \,|\!)\rho^\sharp =$
  $((\!|\, e_1 \,|\!)\rho^\sharp \boxslash^\sharp [a;b]) \boxplus^\sharp (\varepsilon_{\mathbf{f}}((\!|\, e_1 \,|\!)\rho^\sharp) \boxslash^\sharp [a;b]) \boxplus^\sharp mf_{\mathbf{f}}[-1;1]$
- $(\!|\, e_1 \oslash_{\mathbf{f},\mathbf{r}} e_2 \,|\!)\rho^\sharp = (\!|\, e_1 \oslash_{\mathbf{f},\mathbf{r}} \iota((\!|\, e_2 \,|\!)\rho^\sharp)\rho^\sharp \,|\!)$

where the "error" $\varepsilon_{\mathbf{f}}(l)$ of the linear form l is the following linear form:

$$\varepsilon_{\mathbf{f}}\left([a;b] + \sum_{v\in\mathcal{V}} [a_v;b_v]v\right) = (\max(|a|,|b|) \otimes^\sharp_{\mathbf{fa}} [-2^{-\mathbf{p}}; 2^{-\mathbf{p}}]) + \sum_{v\in\mathcal{V}} (\max(|a_v|,|b_v|) \otimes^\sharp_{\mathbf{fa}} [-2^{-\mathbf{p}}; 2^{-\mathbf{p}}])v$$

($\mathbf{p}$ is the fraction size in bits for the format $\mathbf{f}$, see Sect. 3.1)

and the "intervalization" $\iota(l)\rho^\sharp$ function over-approximates the range of the linear form l in the abstract environment $\rho^\sharp$ as the following interval of $\mathbb{I}_{\mathbf{fa}}$:

$$\iota\left(i + \sum_{v\in\mathcal{V}} i_v v\right)\rho^\sharp = i \oplus^\sharp_{\mathbf{fa}} \left(\bigoplus^\sharp_{v\in\mathcal{V}} i_v \otimes^\sharp_{\mathbf{fa}} \rho^\sharp(v)\right)$$

(any summation order for $\oplus^\sharp_{\mathbf{fa}}$ is sound)

Note that this semantics is very different from the one proposed by Goubault in [11] and subsequently used in [13]. In [11], each operation introduces a new variable representing an error term, and there is no need for interval coefficients.

About ι. Dividing a linear form by another linear form which is not reduced to an interval does not yield a linear form. In this case, the ι operator is used to over-approximate the divisor by a single interval before performing the division. The same holds when multiplying two linear forms not reduced to an interval, but we can choose to apply ι to either argument. For the sake of simplicity, we chose here to "intervalize" the left argument. Moreover, any non-linear operator (such as, e.g., square root or sine) could be dealt with by performing the corresponding operator on intervals after "intervalizing" its argument(s).

About $\varepsilon_{\mathbf{f}}$. To account for rounding errors, an upper bound of $|R_{\mathbf{f},\mathbf{r}}(x \cdot y) - (x \cdot y)|$ (where $\cdot \in \{+, -, \times, /\}$) is included in $(\!|\, expr_{\mathbf{f}} \,|\!)\rho^\sharp$. It is the sum an error relative to the arguments x and y, expressed using $\varepsilon_{\mathbf{f}}$, and an absolute error $mf_{\mathbf{f}}$ due to a possible underflow. Unlike what happened with interval arithmetics, correct error computation does not require the abstract operators to use floating-point formats that are no more precise than the concrete ones: the choice of **fa** is completely free.

About Ω. It is quite possible that, during the computation of $(\!|\, expr_{\mathbf{f}} \,|\!)\rho^\sharp$, a floating-point run-time error Ω occurs. In that case, we will say that the linearization "failed". It does not mean that the program has a run-time error, but only that we cannot compute a linearized expression and must revert to the classical interval arithmetics.

Main Result. When we evaluate in $\mathbb{R}$ the linear form $(\!|\, expr_{\mathbf{f}} \,|\!)\rho^\sharp$ in a concrete environment ρ included in $\rho^\sharp$ we get a real interval that over-approximates the concrete value of the expression:

Theorem 1.
If $[\![expr_{\mathbf{f}}]\!]^\sharp\rho^\sharp \neq \Omega$ and the linearization does not fail and $\forall v \in \mathcal{V},\ \rho(v) \in \rho^\sharp(v)$, then $[\![expr_{\mathbf{f}}]\!]\rho \in (\!|\, expr_{\mathbf{f}} \,|\!)\rho^\sharp\ (\rho)$.

Linear Form Propagation. As the linearization manipulates expressions symbolically, it is able to perform simplifications when the same variable appears several times. For instance $Z \leftarrow X \ominus_{\mathbf{32},n} (0.25 \otimes_{\mathbf{32},n} X)$ will be interpreted as $Z \leftarrow [0.749\cdots; 0.750\cdots]X + 2.35\cdots 10^{-38}[-1; 1]$. Unfortunately, no simplification can be done if the expression is broken into several statements, such as in $Y \leftarrow 0.25 \otimes_{\mathbf{32},n} X; Z \leftarrow X \ominus_{\mathbf{32},n} Y$. Our solution is to remember, in an extra environment $\rho_l^\sharp$, the linear form assigned to each variable and use this information while linearizing: we set $(\!|\, v_{\mathbf{f}} \,|\!)(\rho^\sharp, \rho_l^\sharp) = \rho_l^\sharp(v)$ instead of $[-1; 1]v_{\mathbf{f}}$. Care must be taken, when a variable v is modified, to discard all occurrences of v in $\rho_l^\sharp$. Effects of tests on $\rho_l^\sharp$ are ignored. Our partial order on linear forms is flat, so, at control-flow joins, only variables that are associated with the same linear form in both environments are kept; moreover, we do not need any widening. This technique is reminiscent of Kildall's constant propagation [12].

Applications. A first application of linearization is to improve the precision of the interval analysis. We simply replace $[\![expr_{\mathbf{f}}]\!]^\sharp\rho^\sharp$ by $[\![expr_{\mathbf{f}}]\!]^\sharp\rho^\sharp \cap \iota((\!|\, expr_{\mathbf{f}} \,|\!)\rho^\sharp)\rho^\sharp$ whenever the hypotheses of Thm. 1 hold.

While improving the assignment transfer function (trough expression simplification), this is not sufficient to treat tests precisely. For this, we need relational domains. Fortunately, Thm. 1 also means that if we have a relational domain that manipulates sets of points with real coordinates $\mathbb{R}^{\mathcal{V}}$ and that is able to perform assignments and tests of linear expressions with interval coefficients, we can use it to perform relational analyses on floating-point variables. Consider, for instance, the following algorithm to handle an assignment $v \leftarrow expr_{\mathbf{f}}$ in such a relational domain (the procedure would be equivalent for tests):

- If $[\![expr_{\mathbf{f}}]\!]^\sharp \rho^\sharp = \Omega$, then we report a run-time error and apply the transfer function for $v \leftarrow \mathbb{F}_{\mathbf{f}}$.
- Else, if the linearization of $expr_{\mathbf{f}}$ fails, then we do not report an error but apply the transfer function for $v \leftarrow [\![expr_{\mathbf{f}}]\!]^\sharp \rho^\sharp$.
- Otherwise, we do not report an error but apply the transfer function for $v \leftarrow (\!| expr_{\mathbf{f}} |\!) \rho^\sharp$.

Remark how we use the interval arithmetics to perform the actual detection of run-time errors and as a fallback when the linearization cannot be used.

6 Adapting Relational Abstract Domains

We first present in details the adaptation of the octagon abstract domain [14] to use floating-point arithmetics and interval linear forms, which was implemented in our second prototype analyzer [4]. We then present in less details some ideas to adapt other domains.

6.1 The Octagon Abstract Domain

The octagon abstract domain [14] can manipulate sets of constraints of the form $\pm x \pm y \leq c$, $x, y \in \mathcal{V}$, $c \in \mathbb{E}$ where $\mathbb{E}$ can be $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$. An abstract element $\mathbf{o}$ is represented by a half-square constraint matrix of size $|\mathcal{V}|$. Each element at line i, column j with $i \leq j$ contains four constraints: $v_i + v_j \leq c$, $v_i - v_j \leq c$, $-v_i + v_j \leq c$, and $-v_i - v_j \leq c$, with $c \in \overline{\mathbb{E}} = \mathbb{E} \cup \{+\infty\}$. Remark that diagonal elements represent interval constraints as $2v_i \leq c$ and $-2v_i \leq c$. In the following, we will use notations such as $\max_{\mathbf{o}}(v_i + v_j)$ to access the upper bound, in $\overline{\mathbb{E}}$, of constraints embedded in the octagon $\mathbf{o}$.

Because constraints in the matrix can be combined to obtain implied constraints that may not be in the matrix (e.g., from $x - y \leq c$ and $y + z \leq d$, we can deduce $x + z \leq c + d$), two matrices can represent the same set of points. We introduced in [14] a Floyd-Warshall-based closure operator that provides a normal form by combining and propagating, in $\mathcal{O}(|\mathcal{V}|^3)$ time, all constraints. The optimality of the abstract operators requires to work on closed matrices.

Floating-Point Octagons. In order to represent and manipulate efficiently constraints on real numbers, we choose to use floating-point matrices: $\mathbb{F}_{\mathbf{fa}}$ replaces $\mathbb{E}$ (where **fa** is, as before, an efficient floating-point format chosen by the analyzer implementation). As the algorithms presented in [14] make solely use of the $+$ and $\leq$ operators on $\overline{\mathbb{E}}$, it is sufficient to replace $+$ by $\oplus_{\mathbf{fa},+\infty}$ and map Ω to $+\infty$ in these algorithms to provide a sound approximation of all the transfer

functions and operators on reals using only $\overline{\mathbb{F}}_{\mathbf{fa}} = \mathbb{F}_{\mathbf{fa}} \cup \{+\infty\}$. As all the nice properties of $\mathbb{E}$ are no longer true in $\mathbb{F}_{\mathbf{fa}}$, the closure is no longer a normal form. Even though working on closed matrices will no longer guaranty the optimality of the transfer functions, it still greatly improves their precision.

Assignments. Given an assignment of the form $v_k \leftarrow l$, where l is a interval linear form, on an octagon $\mathbf{o}$, the resulting set is no always an octagon. We can choose between several levels of approximation. Optimality could be achieved at great cost by performing the assignment in a polyhedron domain and then computing the smallest enclosing octagon. We propose, as a less precise but much faster alternative ($\mathcal{O}(|\mathcal{V}|$ time cost), to replace all constraints concerning the variable v_k by the following ones:

$$\begin{array}{lll}
v_k + v_i & \leq u(\mathbf{o}, l \boxplus^\sharp v_i) & \forall i \neq k \\
v_k - v_i & \leq u(\mathbf{o}, l \boxminus^\sharp v_i) & \forall i \neq k \\
-v_k + v_i & \leq u(\mathbf{o}, v_i \boxminus^\sharp l) & \forall i \neq k \\
-v_k - v_i & \leq u(\mathbf{o}, \boxminus^\sharp(l \boxplus^\sharp v_i)) & \forall i \neq k \\
v_k & \leq u(\mathbf{o}, l) & \\
-v_k & \leq u(\mathbf{o}, \boxminus^\sharp l) &
\end{array}$$

where the upper bound of a linear form l on an octagon $\mathbf{o}$ is approximated by $u(\mathbf{o}, l) \in \overline{\mathbb{F}}_{\mathbf{fa}}$ as follows:

$$u\left(\mathbf{o},\ [a^-; a^+] + \sum_{v \in \mathcal{V}} [a_v^-; a_v^+] v\right) = a^+ \oplus_{\mathbf{fa},+\infty} \left(\bigoplus_{\mathbf{fa},+\infty}^{v \in \mathcal{V}} \begin{array}{r} \max(\max_{\mathbf{o}}(v) \otimes_{\mathbf{fa},+\infty} a_v^+,\ \ominus(\max_{\mathbf{o}}(-v) \otimes_{\mathbf{fa},-\infty} a_v^+), \\ \max_{\mathbf{o}}(v) \otimes_{\mathbf{fa},+\infty} a_v^-,\ \ominus(\max_{\mathbf{o}}(-v) \otimes_{\mathbf{fa},-\infty} a_v^-)) \end{array}\right)$$

(any summation order for $\oplus_{\mathbf{fa},+\infty}$ is sound)

and $\oplus_{\mathbf{fa},+\infty}$ and $\otimes_{\mathbf{fa},+\infty}$ are extended to $\overline{\mathbb{F}}_{\mathbf{fa}}$ as follows:

$$\begin{array}{l}
+\infty \oplus_{\mathbf{fa},+\infty} x = x \oplus_{\mathbf{fa},+\infty} +\infty = +\infty \\
+\infty \otimes_{\mathbf{fa},+\infty} x = x \otimes_{\mathbf{fa},+\infty} +\infty = \begin{cases} 0 & \text{if } x = 0 \\ +\infty & \text{otherwise} \end{cases}
\end{array}$$

Example 1. Consider the assignment $X = Y \oplus_{\mathbf{32},n} Z$ with $Y, Z \in [0; 1]$. It is linearized as $X = [1 - 2^{-23}; 1 + 2^{-23}](Y + Z) + mf_{\mathbf{32}}[-1; 1]$, so our abstract transfer function will infer relational constraints such as $X - Y \leq 1 + 2^{-22} + mf_{\mathbf{32}}$.

Tests. Given a test of the form $l_1 \leq l_2$, where l_1 and l_2 are linear forms, for all variable $v_i \neq v_j$, appearing in l_1 or l_2, the constraints in the octagon $\mathbf{o}$ can be tightened by adding the following extra constraints:

$$\begin{array}{ll}
v_j - v_i & \leq u(\mathbf{o},\ l_2 \boxminus^\sharp l_1 \boxminus^\sharp v_i \boxplus^\sharp v_j) \\
v_j + v_i & \leq u(\mathbf{o},\ l_2 \boxminus^\sharp l_1 \boxplus^\sharp v_i \boxplus^\sharp v_j) \\
-v_j - v_i & \leq u(\mathbf{o},\ l_2 \boxminus^\sharp l_1 \boxminus^\sharp v_i \boxminus^\sharp v_j) \\
-v_j + v_i & \leq u(\mathbf{o},\ l_2 \boxminus^\sharp l_1 \boxplus^\sharp v_i \boxminus^\sharp v_j) \\
v_i & \leq u(\mathbf{o},\ l_2 \boxminus^\sharp l_1 \boxplus^\sharp v_i)
\end{array}$$

$$-v_i \leq u(\mathbf{o},\ l_2 \boxminus^\sharp l_1 \boxminus^\sharp v_i)$$

Example 2. Consider the test $Y \oplus_{\mathbf{32},n} Z \leq 1$ with $Y, Z \in [0;1]$. It is linearized as $[1-2^{-23}; 1+2^{-23}](Y+Z) + mf_{\mathbf{32}}[-1;1] \leq [1;1]$. Our abstract transfer function will be able to infer the constraint: $Y + Z \leq 1 + 2^{-22} + mf_{\mathbf{32}}$.

Example 3. The optimal analysis of the rate limiter function of Fig. 3 would require representing interval linear invariants on *three* variables. Nevertheless, the octagon domain with our approximated transfer functions can prove that the output Y is bounded by $[-136;136]$ independently from n (the optimal bound being $[-128;128]$), which is sufficient to prove that Y does not overflow.

Reduction with Intervals. The interval environment $\rho^\sharp$ is important as we use it to perform run-time error checking and to compute the linear form associated to an expression. So, we suppose that transfer functions are performed in parallel in the interval domain and in the octagon domain, and then, information from the octagon result $\mathbf{o}$ is used to refine the interval result $\rho^\sharp$ as follows: for each variable $v \in \mathcal{V}$, the upper bound of $\rho^\sharp(v)$ is replaced by $\min(\max \rho^\sharp(v), \max_{\mathbf{o}}(v))$ and the same is done for its lower bound.

6.2 Polyhedron Domain

The polyhedron domain is much more precise than the octagon domain as it allows manipulating sets of invariants of the form $\sum_v c_v v \leq c$, but it is also much more costly. Implementations, such as the New Polka or the Parma Polyhedra libraries [2], are targeted at representing sets of points with integer or rational coordinates. They internally use rational coefficients and, as the coefficients usually become fairly large, arbitrary precision integer libraries.

Representing Reals. These implementations could be used as-is to abstract sets of points with real coordinates, but the rational coefficients may get out of control, as well as the time cost. Unlike what happened for octagons, it is not so easy to adapt the algorithms to floating-point coefficients while retaining soundness as they are much much more complex. We are not aware, at the time of writing, of any such floating-point implementation.

Assignments and Tests. Assignments of the form $v \leftarrow l$ and tests of the form $l \leq 0$ where $l = [a^-; a^+] + \sum_{v \in \mathcal{V}} [a_v^-; a_v^+] v$ seem difficult to abstract in general. However, the case where all coefficients in l are scalar except maybe the constant one is much easier. To cope with the general case, an idea (yet untested) is to use the following transformation that abstracts l into an over-approximated linear form l' where $\forall v,\ a_v^- = a_v^+$ by transforming all relative errors into absolute ones:

$$l' = \left([a^-; a^+] \oplus^\sharp_{\mathbf{fa}} \bigoplus^\sharp_{v \in \mathcal{V}, \mathbf{fa}} (a_v^+ \ominus_{\mathbf{fa},+\infty} a_v^-) \otimes^\sharp_{\mathbf{fa}} [0.5;0.5] \otimes^\sharp_{\mathbf{fa}} \rho^\sharp(v) \right) + \sum_{v \in \mathcal{V}} ((a_v^- \oplus_{\mathbf{fa},+\infty} a_v^+) \otimes^\sharp_{\mathbf{fa}} [0.5;0.5]) v$$

(any summation order for $\oplus^\sharp_{\mathbf{fa}}$ is sound)

6.3 Two Variables per Linear Inequalities Domain

Simon's domain [15] can manipulate constraints of the form $\alpha v_i + \beta v_j \leq c$, $\alpha, \beta, c \in \mathbb{Q}$. An abstract invariant is represented using a planar convex polyhedron for each pair of variables. As for octagons, most computations are done point-wise on variable pairs and a closure provides the normal form by propagating and combining constraints. Because the underlying algorithms are simpler than for generic polyhedra, adapting this domain to handle floating-point computations efficiently may prove easier while greatly improving the precision over octagons. This still remains an open issue.

6.4 Ellipsoid and Digital Filter Domains

During the design of our prototype analyzer [4], we encountered code for computing recursive sequences such as $X_i = ((\alpha \otimes_{\mathbf{32},n} X_{i-1}) \oplus_{\mathbf{32},n} (\beta \otimes_{\mathbf{32},n} X_{i-2})) \oplus_{\mathbf{32},n} \gamma$ (1), or $X_i = (\alpha \otimes_{\mathbf{32},n} X_{i-1}) \oplus_{\mathbf{32},n} (Y_i \ominus_{\mathbf{32},n} Y_{i-1})$ (2). In order to find precise bounds for the variable X, one has to consider invariants out of the scope of classical relational abstract domains. Case (1) can be solved by using the ellipsoid abstract domain of [4] that can represent non-linear real invariants of the form $aX_i^2 + bX_{i-1}^2 + cX_iX_{i-1} \leq d$, while case (2) is precisely analyzed using Feret's filter domains [9] by inferring temporal invariants of the form $|X_i| \leq a \max_{j \leq i} |Y_j| + b$. It is not our purpose here to present these new abstract domains but we stress the fact that such domains, as the ones discussed in the preceding paragraphs, are naturally designed to work with perfect reals, but used to analyze imperfect floating-point computations.

A solution is, as before, to design these domains to analyze interval linear assignments and tests on reals, and feed them with the result of the linearization of floating-point expressions defined in Sect. 5. This solution has been successfully applied (see [9] and Sect. 8).

7 Convergence Acceleration

In the Abstract Interpretation framework, loop invariants are described as fixpoints and are over-approximated by iterating, in the abstract, the body transfer function $F^\sharp$ until a post-fixpoint is reached.

Widening. The widening $\triangledown$ is a convergence acceleration operator introduced in [6] in order to reduce the number of abstract iterations: $\lim_i (F^\sharp)^i$ is replaced by $\lim_i E_i^\sharp$ where $E_{i+1}^\sharp = E_i^\sharp \triangledown F^\sharp(E_i^\sharp)$. A straightforward widening on intervals and octagons is to simply discard unstable constraints. However, this strategy is too aggressive and fails to discover sequences that are stable after a certain bound, such as, e.g., $X = (\alpha \otimes_{\mathbf{32},n} X) \oplus_{\mathbf{32},n} \beta$. To give these computations a chance to stabilize, we use a staged widening that tries a user-supplied set of bounds in increasing order. As we do not know in advance which bounds will be stable, we use, as set $\mathbb{T}$ of thresholds, a simple exponential ramp: $\mathbb{T} = \{\pm 2^i\} \cap \mathbb{F}_{\mathbf{fa}}$. Given two octagons $\mathbf{o}$ and $\mathbf{o}'$, the widening with thresholds $\mathbf{o} \triangledown \mathbf{o}'$ is obtained by setting, for each binary unit expression $\pm v_i \pm v_j$:

$$\max_{\mathbf{o} \triangledown \mathbf{o}'}(C) = \begin{cases} \max_{\mathbf{o}}(C) & \text{if } \max_{\mathbf{o}'}(C) \leq \max_{\mathbf{o}}(C) \\ \min\{t \in \mathbb{T} \cup \{+\infty\} \mid t \geq \max_{\mathbf{o}'}(C)\} & \text{otherwise} \end{cases}$$

Decreasing Iterations. We now suppose that we have iterated the widening with thresholds up to an abstract post-fixpoint $X^\sharp$: $F^\sharp(X^\sharp) \sqsubseteq X^\sharp$. The bound of a stable variable is generally over-approximated by the threshold immediately above. One solution to improve such a bound is to perform some decreasing iterations $X^\sharp_{i+1} = X^\sharp_i \sqcap F(X^\sharp_i)$ from $X^\sharp_0 = X^\sharp$. We can stop whenever we wish, the result will always be, by construction, an abstraction of the concrete fixpoint; however, it may no longer be a post-fixpoint for $F^\sharp$. It is desirable for invariants to be abstract post-fixpoint so that the analyzer can check them independently from the way they were generated instead of relying solely on the maybe buggy fixpoint engine.

Iteration Perturbation. Careful examination of the iterates on our benchmarks showed that the reason we do not get an abstract post-fixpoint is that the *abstract* computations are done in floating-point which incurs a somewhat non-deterministic extra rounding. There exists, between $F^\sharp$'s definitive pre-fixpoints and $F^\sharp$'s definitive post-fixpoints, a chaotic region. To ensure that the $X^\sharp_i$ stay above this region, we replace the intersection $\sqcap$ used in the decreasing iterations by the following narrowing $\triangle$: $\mathbf{o} \triangle \mathbf{o}' = \epsilon(\mathbf{o} \sqcap \mathbf{o}')$ where $\epsilon(\mathbf{o})$ returns an octagon where the bound of each unstable constraint is enlarged by $\epsilon \times d$, where d is the maximum of all non $+\infty$ constraint bounds in $\mathbf{o}$. Moreover, replacing $\mathbf{o} \triangledown \mathbf{o}'$ by $\epsilon(\mathbf{o} \triangledown \mathbf{o}')$ allows the analyzer to skip above $F^\sharp$'s chaotic regions and effectively reduces the required number of increasing iterations, and so, the analysis time.

Theoretically, a good ϵ can be estimated by the relative amount of rounding errors performed in the abstract computation of one loop iteration, and so, is a function of the complexity of the analyzed loop body, the floating-point format **fa** used in the analyzer and the implementation of the abstract domains. We chose to fix ϵ experimentally by enlarging a small value until the analyzer reported it found an abstract post-fixpoint for our program. Then, as we improved our abstract domains and modified the analyzed program, we seldom had to adjust this ϵ value.

8 Experimental Results

We now show how the presented abstract domains perform in practice. Our only real-life example is the critical embedded avionics software of [4]. It is a $132,000$ lines reactive C program (75 KLoc after preprocessing) containing approximately $10,000$ global variables, $5,000$ of which are floating-point variables, single precision. The program consists mostly of one very large loop executed $3.6 \cdot 10^6$ times. Because relating several thousands variables in a relational domain is too costly, we use the "packing" technique described in [4] to statically determine sets of variables that should be related together and we end up with approximately $2,400$ octagons of size 2 to 42 instead of one octagon of size $10,000$.

Fig. 4 shows how the choice of the abstract domains influence the precision and the cost of the analysis presented in [4] on our 2.8 GHz Intel Xeon. Together with the computation time, we also give the number of abstract executions of the big loop needed to find an invariant; thanks to our widenings and narrowings, it is much much less than the concrete number of iterations. All cases use the

	domains			time	nb. of iterations	memory	nb. of alarms
	linearize	octagons	filters				
(1)	×	×	×	1623 s	150	115 MB	922
(2)	√	×	×	4001 s	176	119 MB	825
(3)	√	√	×	3227 s	69	175 MB	639
(4)	√	×	√	8939 s	211	207 MB	363
(5)	√	√	√	4541 s	72	263 MB	6

Fig. 4. Experimental results.

interval domain with the symbolic simplification automatically provided by the linearization, except (1) that uses plain interval analysis. Other lines show the influence of the octagon (Sect. 6.1) and the specialized digital filter domains ([9] and Sect. 6.4): when both are activated, we only get six potential run-time errors for a reasonable time and memory cost. This is a sufficiently small number of alarms to allow manual inspection, and we discovered they could be eliminated without altering the functionality of the application by changing only three lines of code. Remark that as we add more complex domains, the time cost per iteration grows but the number of iterations needed to find an invariant decreases so that a better precision may reduce the overall time cost.

9 Conclusion

We presented, in this paper, an adaptation of the octagon abstract domain in order to analyze programs containing IEEE 754-compliant floating-point operations. Our methodology is somewhat generic and we proposed some ideas to adapt other relational numerical abstract domains as well. The adapted octagon domain was implemented in our prototype static analyzer for run-time error checking of critical C code [4] and tested on a real-life embedded avionic application. Practical results show that the proposed method scales up well and does greatly improve the precision of the analysis when compared to the classical interval abstract domain while maintaining a reasonable cost. To our knowledge, this is the first time relational numerical domains are used to represent relations between floating-point variables.

Acknowledgments. We would like to thank all the members of the "magic" team: Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, David Monniaux, Xavier Rival, as well as the anonymous referees.

References

1. Y. Aït Ameur, G. Bel, F. Boniol, S. Pairault, and V. Wiels. Robustness analysis of avionics embedded systems. In *LCTES'03*, pages 123–132. ACM Press, 2003.
2. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *SAS'02*, volume 2477 of *LNCS*, pages 213–229. Springer, 2002.

3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS, pages 85–108. Springer, 2002.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM PLDI'03*, volume 548030, pages 196–207. ACM Press, 2003.
5. IEEE Computer Society. IEEE standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std 745-1985, 1985.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*, pages 238–252. ACM Press, 1977.
7. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL'78*, pages 84–97. ACM Press, 1978.
9. J. Feret. Static analysis of digital filters. In *ESOP'04*. LNCS, 2004.
10. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.
11. É. Goubault. Static analyses of floating-point operations. In *SAS'01*, volume 2126 of *LNCS*, pages 234–259. Springer, 2001.
12. G. Kildall. A unified approach to global program optimization. In *POPL'73*, pages 194–206. ACM Press, 1973.
13. M. Martel. Static analysis of the numerical stability of loops. In *SAS'02*, volume 2477 of *LNCS*, pages 133–150. Springer, 2002.
14. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, 2001.
15. A. Simon, A. King, and J. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'02*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.
16. R. Skeel. Roundoff error and the Patriot missile. *SIAM News*, 25(4):11, July 1992.
17. J. Vignes. A survey of the CESTAC method. In J-C. Bajard, editor, *Proc. of Real Numbers and Computer Conference*, 1996.