The pitfalls of verifying floating-point computations

David Monniaux
CNRS / Laboratoire d'informatique de l'École normale supérieure
http://www.di.ens.fr/~monniaux

Abstract

Current critical systems commonly use a lot of floating-point computations, and thus the testing or static analysis of programs containing floating-point operators has become a priority. However, correctly defining the semantics of common implementations of floating-point is tricky, because semantics may change with many factors beyond source-code level, such as choices made by compilers. We here give concrete examples of problems that can appear and solutions to implement in analysis software.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification—formal methods, validation, assertion checkers; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification, assertions, invariants; G.1.0 [*Mathematics of Computing*]: Numerical analysis—Computer arithmetic, Interval arithmetic; G.4 [*Mathematics of Computing*]: Mathematical software—Certification and testing, Algorithm design and analysis

General Terms Experimentation, Reliability, Standardisation, Verification

Keywords Abstract interpretation, Static analysis, Program testing, Verification, Floating point, Embedded software, Safety-Critical Software, x87, IA32, AMD64, PowerPC, FPU, Rounding, IEEE-754

1. Introduction

In the past, critical applications often used fixed-point computations. However, with the wide availability of processors with hardware floating-point units, many current critical applications (say, for controlling automotive or aerospace systems) use floating-point operations. Such applications have to undergo stringent testing or validation. In this paper, we show how the particularities of floating-point implementations can hinder testing methodologies, and have to be cared for in static analysis.

It has been known for a long time that it was erroneous to compute with floating-point numbers and operations as though they were on the real field. There exist entire treatises discussing the topic of stability in numerical algorithms from the point of view of the applied mathematician: whether or not some algorithm, when implemented with floating-point, will give "good" approximations of the real result; we will not discuss such issues in this paper. The purpose of this paper is to show the kind of difficulties that floating-point computations pose for static analysis and program testing methods: both for defining the semantics of the programs to be analysed, and for defining and implementing the analysis techniques.

For the sake of a better understanding, in Sect. 2, we recall the bases of IEEE-754 arithmetics. For a wider perspective on issues related to floating-point computations, see the papers by William Kahan.

A naive approach to floating-point issues is that since all current commonplace platforms claim to support IEEE-754, there should not be problems for analysing or simulating the computations of one platform on another platform. In fact, there are subtle differences between hardware or software floating-point units (FPUs); and compilers tend not to implement exact IEEE-754 single- or double-precision semantics [VCV97]. Differences between hardware platforms are one reason why, for instance, implementors of Java found it difficult to implement consistent floating-point across various systems, since the semantics of Java programs was supposed to be the same whatever the platform.[1] A well-known issue is the 80-bit internal floating-point registers on the Intel platform. [Sun01, Appendix D] In Section 3, we shall expand on such issues and show, for instance, how low-level issues such as register allocation [AG97, chapter 11] and the insertion of logging instructions with no "apparent" computational effects can change the final results of computations. In Section 4.1 we shall discuss issues pertaining to the PowerPC architecture.

An important factor throughout the discussion is that it is not the hardware platform that matters in itself, but its combination with the software context, including the compiler, libraries, and possible runtime environment. Compatibility has to be appreciated at the level of the application writer — whether code written using types mapped to IEEE normalised types will effectively behave as though all atomic floating-point operations (say, $+$, $-$, $\times$, $/$) will

[1] Java's early floating-point model was a strict subset of IEEE-754 [GJS96, §4.2.3]: essentially, strict IEEE-754 single and double-precision arithmetics without the exception traps (overflow, invalid operation…) and without rounding modes other than round-to-nearest. However, strict compatibility with IEEE-754 single- and double-precision operations is difficult to achieve on certain widely used platforms, such as the Intel IA32 (x86); on all but the latest processors, it incurs a performance hit. As a consequence requests were made so that strict compatibility would be relaxed in order to get better performance, particularly for scientific computing applications. The possibility of giving up Java's deterministic, portable semantics was requested by some [KD98], but controversial for others [Pan98]. Finally, the Java language specification was altered [GJSB00, §4.2.3]: runtime computing in extra precision (single-extended and double-extended formats) was allowed for classes and methods not carrying the new `strictfp` modifier [GJSB00, §15.4]. The *Borneo* project introduces full IEEE-754 features.

go according to the standardised definition. [IEC89, IEE85, §1.1] Indeed, the standard recalls [IEC89, IEE85, §1.1]:

> *It is the environment the programmer or user of the system sees that conforms or fails to conform to this standard. Hardware components that require software support to conform shall not be said to conform apart from such software.*

IEEE-754 normalises a few basic operations; however, many programs use functions such as sine, cosine, $\ldots$, which are not specified by this norm and are generally not strictly specified in the system documentation. In Sect. 4.2, we shall explore some difficulties with respect to mathematical libraries. In addition to issues related to certain floating-point implementations, or certain mathematical libraries, there are issues more particularly related to the C programming language, its compilers and libraries. Sect. 4.3 explores such system dependencies. Sect. 4.4 explores issues with input and output of floating-point values.

A commonly held opinion is that whatever the discrepancies, they will be negligible enough and should not have noticeable consequences. In Section 5, we give a complete example of some seemingly innocuous floating-point code fragment based on real-life industrial code. We illustrate how the floating-point "oddities" that we explained in the preceding sections can lead to rare and extremely hard to diagnose runtime errors.

In Section 6 we analyse the consequences of these issues on abstract interpretation-based static analysis and other validation techniques, and show how to obtain sound results. The static analysis techniques that we describe are implemented in the ASTRÉE static analyser [BCC$^+$02, BCC$^+$03, CCF$^+$05].

We shall be particularly interested in programs written in the C programming language, because this language is often favoured for embedded systems. We shall in particular discuss some implications of the most recent norm of that language, "C99" [ISO99].

2. IEEE-754: a reminder

All current general-purpose microprocessors, and many microcontrollers, implement hardware floating-point as a variant of norm ANSI/IEEE-754 [IEE85], later adopted as international standard IEC-60559 [IEC89]. We thus begin by an overview of this standard.

2.1 Numbers

IEEE floating point numbers are of the following kinds:

Zeroes There exist both a $+0$ and a -0. The difference between the two has practical importance only if one extracts the sign bit from the number, or if one divides a nonzero number by a zero (the sign of the zero determines whether $+\infty$ or $-\infty$ is returned).

Infinities Infinities are generated by divisions by zero or by *overflow* (computations of numbers of such a large magnitude that they cannot be represented).

NaNs The special values *Not a Number* (NaN) represent the result of operations that cannot have a meaningful result in terms of an finite number or infinity. Such is for instance the case of $(+\infty) - (+\infty)$, $0/0$ or $\sqrt{-1}$.

Normalised numbers This is the most common kind of nonzero representable reals.

Denormalised numbers These represent some values very close to zero. They pose special issues regarding rounding errors.

Floating point numbers are represented as follows: $x = \pm s.m$ where $1 \leq m < 2$ is the *mantissa*, which has a fixed number p of bits, and $s = 2^e$ the *scaling factor* ($E_{\min} \leq e \leq E_{\max}$ is the *exponent*). The difference introduced by changing the last binary digit of the mantissa is $\pm s.\varepsilon_{\text{last}}$ where $\varepsilon_{\text{last}} = 2^{-(p-1)}$: the *unit in the last place* or *ulp*. For Any nonzero number can be represented uniquely in this way if we impose that the leftmost digit of the mantissa is 1 — this is called a *normalised representation*. Except in the case of numbers of very small magnitude, IEEE-754 always works with normalised representations.

The IEEE-754 *single precision* type, associated to C's `float` type [ISO99, F.2], has $p = 24$, $E_{\min} = -126$, $E_{\max} = +127$. The IEEE-754 *single precision* type, associated to C's `double`, has $p = 53$, $E_{\min} = -1022$, $E_{\max} = +1023$.

We thus obtain a floating-point representation of the form:

$$x = \pm[1.m_1 \ldots m_{p-1}]_2.2^e \tag{1}$$

We note $[vvv]_2$ the representation of a number in terms of binary digits vvv.

2.2 Rounding

Let us note x the result of an operation between two non-NaN, non-infinity floating point numbers in the real field. Each ideal real x is mapped to a floating-point value $r(x)$ by a uniquely defined rounding function; the choice of this function is determined by the *rounding mode*. IEEE-754 mandates four standard rounding modes:

- round-to-$+\infty$: $r(x)$ is the least floating point value greater than or equal to x;
- round-to-$-\infty$: $r(x)$ is the greatest floating point value smaller than or equal to x;
- round-to-0: $r(x)$ is the floating-point value of the same sign as x such that $|r(x)|$ is the greatest floating point value smaller than or equal to $|x|$;
- round-to-nearest: $r(x)$ is the floating-point value closest to x with the usual distance (see below for details); this is the default.

Depending on the rounding mode, the error $\delta = r(x) - x$ committed on normalised floating-point results (see below for denormalised and underflow results) is constrained as follows: in round-to-$+\infty$ mode, $0 \leq \delta < s.\varepsilon_{\text{last}}$; in round-to-$-\infty$ mode, $-s.\varepsilon_{\text{last}} < \delta \leq 0$; in round-to-0 mode, $-s.\varepsilon_{\text{last}} < \delta < s.\varepsilon_{\text{last}}$; in round-to-nearest mode, $-s.\varepsilon_{\text{last}}/2 \leq \delta \leq s.\varepsilon_{\text{last}}/2$.

We can bound the maximal error:

$$|x|.\varepsilon_{\text{last}}/2 < |s|.\varepsilon_{\text{last}} \leq |x|.\varepsilon_{\text{last}} \tag{2}$$

and $|\delta| \leq |x|.\varepsilon_{\text{last}}/2$ in round-to-nearest mode, $|\delta| \leq x.\varepsilon_{\text{last}}$ in other modes. If we do not assume a specific rounding mode, we should take $\varepsilon_{\text{rel}} = \varepsilon_{\text{last}}$. Thus, whatever the rounding mode, for any x such that $r(x)$ does not result in underflow, $|x - r(x)| \leq \varepsilon_{\text{rel}}.|x|$.

Because the exponent of floating-point numbers is bounded ($e_{\min} \leq e \leq e_{\max}$), there exists a minimal positive representable floating-point number ε_{abs} (in IEEE-754 double precision arithmetics, it is $2^{-1074} \approx 5 \times 10^{-324}$). When $|x| < \varepsilon_{\text{abs}}$, depending on rounding conditions, $r(x)$ may be rounded to $\pm\varepsilon_{\text{abs}}$ or to 0. The generation of a zero floating-point result for a nonzero real result is known as *underflow*.

In addition to normalised numbers, IEEE floating-point representations allow for *denormal* numbers.[2] These are numbers very close to zero and are of the form:

$$x = \pm[0.m_1 \ldots m_{p-1}]_2.2^{e_{\min}} \tag{3}$$

[2] Note, however, that certain floating-point units such as the SSE allow disabling the use of denormals for efficiency reasons (see Sect. 3.3). We still can accommodate this case by choosing ε_{abs} to be the least positive *normalised* number, that is, $2^{e_{\min}-(p-1)}$.

It follows that $\varepsilon_{\text{abs}} = 2^{e_{\min}-(p-1)}$. We can bound the maximal error: $|\delta| \le \varepsilon_{\text{abs}}/2$ in round-to-nearest mode, $|\delta| \le \varepsilon_{\text{abs}}$ in other modes. The generation of denormal numbers is also considered by IEEE-754 to be an underflow [IEC89, IEE85, §7.4] — especially since they may incur an "extraordinary" absolute loss of precision, while normalised results incur a relative loss of precision.

2.3 Operations

IEEE-754 normalises 5 operations: addition (which we shall note $\oplus$ in order to distinguish it from the operation over the reals), subtraction ($\ominus$), multiplication ($\otimes$), division ($\oslash$), and also square root.

IEEE-754 specifies *exact rounding* [Gol91, §1.5]: the result of a floating-point operation is the same as if the operation was performed on the reals with the given inputs, then rounded according to the rules in the preceding section. Thus, $x \oplus y$ is defined as $r(x+y)$, with x and y taken as elements of $\mathbb{R} \cup \{-\infty, +\infty\}$; the same applies for the other operators.

From the inequalities derived in 2.1, we obtain

$$|x - r(x)| \le \max(\varepsilon_{\text{rel}}.|x|, \varepsilon_{\text{abs}}) \tag{4}$$

Thus also, *a fortiori*:

$$|x - r(x)| \le \varepsilon_{\text{rel}}.|x| + \varepsilon_{\text{abs}}. \tag{5}$$

It is well-known that floating-point operations are *not associative* (e.g $(10^{20} \oplus 1) \ominus 10^{20} = 0 \ne 1 = (10^{20} \ominus 10^{20}) \oplus 1)$. Many symbolic computation techniques, used in static analysers or in compiler optimisers, assume some good algebraic properties of the arithmetics in order to be sound. In Sect. 6.1, we shall explain how it is possible to make such methods sound with respect to floating-point.

3. IA32, AMD64 and ET64 architectures

The IA32 architecture, originating from Intel, encompasses processors such as the i386, i486 and the various Pentium variants. It is, as of 2005, the most common architecture for micro-computers. The AMD64 and ET64 architectures are 64-bit extensions of IA32. IA32 offers almost complete ascending compatibility from the 8086 processor, first released in 1978; it features a floating-point unit, often nicknamed x87, mostly upwardly compatible with the 8087 co-processor, first released in 1980. However, later, another floating-point unit, known as SSE, was added to the architecture.

3.1 x87 floating-point unit

Processors of the IA32 architecture (Intel 386, 486, Pentium etc. and compatibles) feature a floating-point unit often known as "x87" [Int05, chapter 8].

> *It supports the floating-point, integer, and packed BCD integer data types and the floating-point processing algorithms and exception handling architecture defined in the IEEE Standard 754 for Binary Floating-Point Arithmetic.*

This unit has 80-bit registers internally in "extended double" format (64-bit mantissa and 15-bit exponent), often associated to the `long double` C type; it can read and write data to memory in this 80-bit format or in standard IEEE-754 single and double precision. By default, all operations performed on CPU registers are done with 64-bit precision, but it is possible to reduce precision to 24-bit (same as IEEE single precision) and 53-bit (same as IEEE double precision) mantissas by setting some bits in the unit's control register.[Int05, §8.1.5.2] Note, however, that these precision settings do not affect the range of exponents available, and only affect a limited number of operations (containing all operations specified in IEEE-754).

The most usual way of generating code for the IA32 is to hold temporaries — and, in optimised code, program variables — in the x87 registers. Doing so yields more compact and efficient code than always storing register values into memory and reloading them. However, it is not always possible to do everything inside registers, and compilers then generally store extra temporary values to main memory using the type of the value per the typing rules of the language. This means that *the final result of the computations depend on how the compiler allocates registers*, since temporaries (and possibly variables) will incur or not incur rounding whether or not they are spilt to main memory.

As an example, the following program compiled with `gcc` 4.0.1 [Fre] under Linux will print 10^{308} (`1E308`):

```
  double v = 1E308;
  double x = (v * v) / v;
  printf("%g %d\n", x, x==v);
```

How is that possible? `v * v` done in double precision will overflow, and thus yield $+\infty$, and the final result should be $+\infty$. However, since all computations are performed in extended precision, the computations do not overflow. However, if we use the `-ffloat-store` option, which forces `gcc` to store floating-point variables in memory, we obtain $+\infty$.

The result of computations can actually depend on compilation options or compiler versions, or anything that affects propagation. With the same compiler and system, the following program prints 10^{308} (when compiled in optimised mode (`-O`), while it prints $+\infty$ when compiled in default mode.

```
double foo(double v) {
        double y = v * v;
        return (y / v);
}
main() { printf("%g\n", foo(1E308));}
```

Examination of the assembly code shows that when optimising, the compiler reuses the value of `y` stored in a register, while it saves and reloads `y` to and from main memory in non-optimised mode.

A common optimisation is *inlining* — that is, replacing a call to a function by the expansion of the code of the function at the point of call. For simple functions (such as small arithmetic operations, e.g. $x \mapsto x^2$), this can increase performance significantly, since function calls induce costs (saving registers, passing parameters, performing the call, handling return values). C [ISO99, §6.7.4] and C++ have an `inline` keyword in order to pinpoint functions that should be inlined (however, compilers are free to inline or not to inline such functions; they may also inline other functions when it is safe to do so). However, on x87, whether or not inlining is performed may change the semantics of the code!

Consider what `gcc` 4.0.1 on IA32 does with the following program, depending on whether the optimisation switch `-O` is passed:

```
static inline double f(double x) {
  return x/1E308;
}
double square(double x) { return x*x; }
int main(void) {
  printf("%g\n", f(square(1E308)));
}
```

`gcc` does not inline functions when optimisation is turned off. The `square` function returns a `double`, but the calling convention is to return floating point value into a x87 register — thus in `long double` format. Thus, when `square` is called, it returns approximately 10^{716}, which fits in `long double` but not `double` format. But when `f` is called, the parameter is passed on the stack — thus as a `double`, $+\infty$. The program therefore prints $+\infty$. In comparison, if the program is compiled with optimisation on, `f` is inlined; no parameter passing takes place, thus no conversion to `double` before division, and thus the final result printed is 10^{308}.

It is somewhat common for programmers to add a comparison check to 0 before computing a division, in order to avoid possible division-by-zero exceptions or the generation of infinite results. A first objection to this practise is that, anyway, computing $1/x$ for x very close to zero will generate very large numbers that will result in overflows later. Another objection is that it may actually not work, depending on what the compiler does.

Consider the following source code:[3]

```
void do_nothing(double *x) { }
int main(void) {
  double x = 0x1p-1022, y = 0x1p100, z;
  do_nothing(&y);
  z = x / y;
  if (z != 0) {
    do_nothing(&z);
    assert(z != 0);
  }
}
```

This program exhibits different behaviours depending on various factors, even when one uses the same compiler (`gcc` version 4.0.2 on IA32):

- If it is compiled without optimisation, `x / y` is computed as a `long double` then converted into a IEEE-754 double precision number (0) in order to be saved into memory variable `z`. The `if` statement is thus not taken.
- If it is compiled as a single source code with optimisation, `gcc` performs some kind of global analysis which understands that `do_nothing` does nothing. Then, it does constant propagation, sees that `z` is 0, thus that the `if` statement is not taken, and finally that `main()` performs no side effect. It then effectively compiles `main()` as a "no operation".
- If it is compiled as two source codes (one for each function), `gcc` cannot do constant propagation. The `z != 0` is performed on a nonzero `long double` quantity and thus is taken. However, after the second `do_nothing()` call, `z` is reloaded from main memory as the value 0 (because conversion to double-precision flushed it to 0). As a consequence, the printed result is $+\infty$.
- If, with the same compilation setup, one removes the second `do_nothing()` call, the program detects an assertion failure and aborts. Note that cursory program analysis, optimisation, or naive static analysis may well conclude that the assertion `z != 0` is true throughout the `if` branch.

One should therefore be extra careful with strict comparisons, because these may be performed on the extended precision type.

We are surprised of these discrepancies. After all, the C specification says [ISO99, 5.1.2.3, *program execution*, §12, ex. 4]:

> *Implementations employing wide registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a register or in memory. For example, an implicit spilling of a register is not permitted to alter the value. Also, an explicit store and load is required to round to the precision of the storage type.*

However, this paragraph, being an example, is not normative. [ISO99, foreword, §6].

Let us note, finally, that common debugging practises that, apparently, should not change the computational semantics, may actually alter the result of computations. Adding a logging statement in the middle of a computation may alter the scheduling of registers, for instance by forcing some value to be spilt into main memory and thus undergo additional rounding. As an example, simply inserting a `printf("%g\n", y);` call after the computation of `y` in the above `foo` function forces `y` to be flushed to memory, and thus the final result then becomes $+\infty$ regardless of optimisation.

Also, it is commonplace to disable optimisation when one intends to use a software debugger, because in optimised code, the compiled code corresponding to distinct statements may become fused, variables may not reside in a well-defined location, etc. However, as we have seen, simply disabling or enabling optimisation may change computational results.

3.2 Double rounding

In some circumstances, floating-point results are rounded twice in a row, first to a type A then to a type B. Surprisingly, such *double rounding* can yield different results from direct rounding to the destination type.[4] Such is the case, for instance, of results computed in the `long double` 80-bit type of the x87 floating-point registers, then rounded to the IEEE double precision type for storage in memory. In round-to-0, round-to-$+\infty$ and round-to-$-\infty$ modes, this is not a problem provided that the values representable by type B are a subset of those representable by type A. However, in round-to-nearest mode, there exist some borderline cases where differences are exhibited.

In order to define the round-to-nearest mode, one has to define arbitrarily how to round a real exactly in the middle between the nearest floating-point values. IEEE-754 chooses round-to-even [IEC89, IEE85, §4.1]:[5]

> *In this mode, the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant bit equal to zero shall be delivered.*

This definition makes it possible for double rounding to yield different results than single rounding to the destination type. Consider a floating-point type B where two consecutive values are x_0 and $x_0 + \delta_B$, and another floating-type A containing all values in B and also $x_0 + \delta_B/2$. There exists δ_A such that all reals in the interval $I =]x_0 + \delta_B/2 - \delta_A/2, x_0 + \delta_B/2[$ get rounded to $x_0 + \delta_B/2$ when mapped to A. We shall suppose that the mantissa of x_0 finished by a 1. If $x \in I$, then indirect rounding yields: $x \rightarrow_A x_0 + \delta_B/2 \rightarrow_B x_0 + \delta_B$ and direct rounding yields: $x \rightarrow_B x_0$.

A practical example is with $x_0 = 1 + 2^{-52}$, $\delta = 2^{-52}$ and $r = x_0 + y$ where $y = \delta/2(1 - 2^{-11})$. Both x_0 and y are exactly representable in IEEE-754 double precision (B).

```
double x0 = 0x1.0000000000001p0;
double y = 0x1p-53 * (1. - 0x1p-11);
double z1 = x0 + y;
double z2 = (long double) x0 + (long double) y;
printf("%a %a\n", z1, z2);
```

We see that $z_1 = x_0$ and that $z_2 = x_0 + 2^{-52}$. In order to get true IEEE-754 computations on the `double` type, we execute the single- and double- precision computations on the SSE unit (see Sect. 3.3) of an AMD64 or Pentium 4 processor. Note that if this

[3] C99 introduces hexadecimal floating-point literals in source code. [ISO99, §6.4.4.2] Their syntax is as follows: `0x`*mmmmmm.mmmm*`p`±*ee* where *mmmmmm.mmmm* is a mantissa in hexadecimal, possibly containing a point, and *ee* is n exponent possibly preceded by a sign. They are interpreted as $[mmmmmm.mmmm]_{16} \times 2^{ee}$. See also Sect. 4.4.

[4] This problem has been known for a long time.[FdC00, chapter 6][Gol91, 4.2]

[5] [Gol91, 1.5] explains a rationale for this.

program is executed on an IA32 processor, it is likely that $z_1 = z_2 = x_0 + 2^{-52}$, as seen on Linux / gcc-4.0.1: the computations on `double` will be actually performed in the `long double` type inside the processor, then converted to IEEE double precision.

A similar problem occurs with rounding behaviour near infinities:

> *However, an infinitely precise result with magnitude at least $2^{E_{max}}(2 - 2^{-p})$ shall round to ∞ with no change in sign.*

For IEEE double-precision, $E_{\max} = 1023$ and $p = 53$; let us take x_0 to be the greatest representable real, $M_{\text{double}} = 2^{E_{\max}}(2 - 2^{-(p-1)})$ and $y = 2^{970}(1 - 2^{-11})$. With a similar program as above, $r = x_0 + y$ gets rounded to $z_1 = x_0$ in IEEE double precision, but gets rounded to $2^{E_{\max}}(2-2^{-p})$ in extended precision. As a result, the subsequent conversion into IEEE double precision will yield $+\infty$.

Double rounding can also cause some subtle differences for very small numbers that are rounded into denormal double-precision values if computed in IEEE-754 double precision: if one uses the "double-precision" mode of the x87 FPU, these numbers will be rounded into normalised values inside the FPU register, because of a wider range of negative exponents; then they will be rounded again into double-precision denormals when written to memory. This is known as *double-rounding on underflow* [Sun, §10.4.3.1]. Working around double-rounding on underflow is extremely tedious and incurs significant efficiency penalties (however, the phenomenon is exhibited by $\times$ and $/$, not by $+$ and $-$). [Pan98] A concrete example : taking `x` $=$ `0x1.8000000000001p-1018` ($\approx 5.34018 \times 10^{-307}$) and `y` $=$ `0x1.0000000000001p+56` ($\approx 7.20576 \times 10^{16}$), then `x` $\oslash$ `y` $=$ `0x0.0000000000001p-1022` in IEEE-754 double precision and `x` $\oslash$ `y` $=$ `0x0.0000000000002p-1022` with the x87 in "double precision mode".

3.3 SSE floating-point unit

Intel introduced in the Pentium III processor the SSE floating-point unit [Int05, chapter 10], then the SSE2 extension in the Pentium 4 [Int05, chapter 11]. These extensions to the x86 instruction set contain, respectively, IEEE-compatible single-precision and double-precision instructions.

One can make `gcc` generate code for the SSE subsystem with the `-fpmath=sse` option; since SSE is only available for certain processors, it is also necessary to specify, for instance, `-march=pentium4`. On AMD64, `-fpmath=sse` is the default.

Note the implication: the same program may give different results when compiled on 32-bit and 64-bit "PCs" (or even the same machine, depending on whether one compiles in 32-bit or 64-bit mode) because of the difference in the default floating-point subsystem used.

In addition, the SSE unit offers some non-IEEE-754 compliant modes for better efficiency: with the *flush-to-zero* flag [Int05, §10.2.3.3] on, denormals are not generated and are replaced by zeroes; this is more efficient. As we noted in Sect. 2.1, this does not hamper obtaining good bounds on the errors introduced by floating-point computations; also, we can assume the worst-case situation and suppose that this flag is on when we derive error bounds.

The flush-to-zero flag, however, has another notable consequence: $x \ominus y = 0$ is no longer equivalent to $x = y$. As an example, if $x = 2^{-1022}$ and $y = 1.5 \times 2^{-1022}$, then $y \ominus x = 2^{-1023}$ in normal mode, and $y \ominus x = 0$ in flush-to-zero mode. Analysers should therefore be careful when replacing comparisons by "equivalent" comparisons.

In addition, there exists a *denormals-are-zero* flag [Int05, §10.2.3.4]: if it is on, all denormal operands are considered to be zero, which improves performance. It is still possible to obtain bounds on the errors of floating point computations by assuming that operands are offset by an amount of at most $\pm 2^{e_{\min}-(p-1)}$ before being computed upon. However, techniques based are exact replays of instruction sequences will have to replay the sequence with the same value of the flag.

3.4 Problems and solutions

The problems of running programs written with strict IEEE-754 compliance in mind on the *nearly compatible* x87 floating-point unit have long been recognised. For this reason, `gcc` has a `-ffloat-store` option, flushing floating-point variables to memory. [Fre] Indeed, the `gcc` manual [Fre] says:

> *On 68000 and x86 systems, for instance, you can get paradoxical results if you test the precise values of floating point numbers. For example, you can find that a floating point value which is not a NaN is not equal to itself. This results from the fact that the floating point registers hold a few more bits of precision than fit in a `double` in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them. You can partially avoid this problem by using the `-ffloat-store` option.*

The manual refers to the following option:

> *`-ffloat-store` Do not store floating point variables in registers, and inhibit other options that might change whether a floating point value is taken from a register or memory.*
>
> *This option prevents undesirable excess precision on machines [...] where the floating registers [...] keep more precision than a 'double' is supposed to have. Similarly for the x86 architecture. For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use '`-ffloat-store`' for such programs, after modifying them to store all pertinent intermediate computations into variables.*

Note that this option does not force unnamed temporaries to be flushed to memory, as shown by experiments. To our knowledge, no compiler offers the choice to always flush temporaries to memory, or to flush temporaries to `long double` memory, which would at least remove the worst problem, which is the non-reproducibility of results depending on factors independent of the computation code (register allocation differences caused by compiler options or debugging code, etc.).

Unfortunately, the above precautions do not reconstitute exact IEEE-754 behaviour in round-to-nearest mode, because of the double rounding problem (3.2). In addition, this option is difficult to use: to get IEEE-754 behaviour, the programmer would have to rewrite all program formulae to store temporaries in variables. This does not seem to be reasonable for human-written code, but may be possible with automatically generated code — it is frequent that control/command applications are implemented in a high-level language such as Simulink, the compiled into C.

Another possibility is to force the floating-point unit to limit precision to IEEE-754 standard precisions.[6] This *mostly* solves the double-rounding problem. However, there is no way to constrain the range of the exponents, and thus these modes do not allow exact simulation of IEEE computations when overflows are possible. For instance, the programs of Sec. 3.1, which result in overflows to $+\infty$ if run under strict IEEE-754 compliant arithmetics, do not result in overflows if run with the x87 in double precision mode. Let us note, however, that if a computation never results in overflows when done with IEEE-754 double-precision (resp. single-) arithmetics, it

[6] This is the default setting on FreeBSD 4, presumably in order to achieve closer IEEE-754 compatibility.

can be exactly simulated with the x87 in double-precision (resp. single).[7]

If one wants semantics almost exactly faithful to IEEE-754 in round-to-nearest mode, including with respect to overflow condition, one can use, at the same time, limitation of precision and options and programming style that force operands to be systematically written to memory between floating-point operations. This is incurs some performance loss. Furthermore, there is still a discrepancy due to double rounding of numbers with very small absolute values A simpler solution in current machines is simply to force the compiler to use the SSE unit for computations on IEEE-754 types.

4. Other issues

While the 80-bit internal precision of the x87 may cause significant semantic problems, they are not the only such features. Here, we explore how the PowerPC architecture has some similar problems, and how floating-point libraries are often poorly tested.

4.1 PowerPC architecture

The floating point operations implemented in the PowerPC architecture are compatible with IEEE-754 [Fre01b, §1.2.2.3, §3.2]. However, [Fre01b, §4.2.2] also points out that:

> *The architecture supports the IEEE-754 floating-point standard, but requires software support to conform with that standard.*

The PowerPC architecture features floating-point multiply-add instructions [Fre01b, §4.2.2.2]. These perform $(a, b, c) \mapsto \pm a.b \pm c$ computations in one instruction — with obvious benefits for computations such as matrix computations [CLR90, §26.1], dot products, or Horner's rule for evaluating polynomials [CLR90, §32.1]. Note, however, that they are not semantically equivalent to performing separate addition, multiplication and optional negate IEEE-compatible instructions; in fact, intermediate results are computed with extra precision [Fre01b, D.2]. Whether these instructions are used or not depends on the compiler, optimisation options, and also how the compiler subdivides instructions. For instance, `gcc` 3.3 compiles the following code using the multiply-add instruction if optimisation (`-O`) is turned on, but without it if optimisation is off, yielding different semantics:[8]

```
double dotProduct(double a1, double b1,
                  double a2, double b2) {
        return a1*b1 + a2*b2;
}
```

In addition, the `fpscr` control register has a `NI` bit, which, if on, possibly enables implementation-dependent semantics different from IEEE-754 semantics. [Fre01b, §2.1.4]. For instance, on the MPC750 family, such non-compliant behaviour encompasses flushing denormal results to zero, rounding denormal operands to zero, and treating NaNs differently [Fre01a, §2.2.4]. Similar caveats apply as in Sect. 3.3.

4.2 Mathematical functions

Many operations related to floating-point are not implemented in hardware; most programs using floating-point will thus rely on suitable support libraries. Our purpose, in this section, is not to list comprehensively bugs in current floating-point libraries; it is to illustrate, using examples from common operating systems and runtime environments, the kind of problems that may happen.

4.2.1 Transcendental functions

A first remark is that, though IEEE-754 specifies the behaviour of elementary operations $+$, $-$, $\times$, $/$ and $\sqrt{}$, it does not specify the behaviour of other functions, including the popular trigonometric functions. These are generally supplied by a system library, occasionally by the hardware.

As an example, consider the sine function. On the x87, it is implemented in hardware; on Linux IA32, the GNU libc function `sin()` is just a wrapper around the hardware call, or, depending on compilation options, can be replaced by inline assembly.[9] Intel [Int05, §8.3.10] and AMD [Adv05, §6.5.5] claim that their transcendental instructions (on recent processors) commit errors less than 1 ulp in round-to-nearest mode; however it is to be understood that this is after the operands of the trigonometric functions are reduced modulo 2π, which is done currently using a 66-bit approximation for π. [Int05, §8.3.8] However, the AMD-K5 used a 512-bit value.[10]

One obtains different results for `sin(0x1969869861.p+0)` on PCs running Linux. The Intel Pentium 4, AMD Athlon64 and AMD Opteron processors, and GNU libc running in 32-bit mode on IA32 or AMD64 all yield `-0x1.95b011554d4b5p-1`, However, Mathematica, Sun Sparc under Solaris and Linux, and GNU libc on AMD64 (in 64-bit mode) yield `-0x1.95b0115490ca6p-1`.

A more striking example of discrepancies is $\sin(p)$ where $p = 14885392687$. This value was chosen so that $\sin(p)$ is close to 0, in order to demonstrate the impact of imprecise reduction modulo 2π.[11] Both the Pentium 4 x87 and Mathematica yield a result $\sin(p) \approx 1.671 \times 10^{-10}$. However, GNU libc on AMD64 yields $\sin(p) \approx 1.4798 \times 10^{-10}$, about 11.5% off!

Note, also, that different processors within the same architecture can implement the same transcendental functions with different accuracies. We already noted the difference between the AMD-K5 and the K6 and following processors with respect to angular reduction. Intel also notes that the algorithms' precision was improved between the 80387 / i486DX processors and the Pentium processors. [Int97, §7.5.10]

> *With the Intel486 processor and Intel 387 math coprocessor, the worst-case, transcendental function error is typically 3 or 3.5 ulps, but is sometimes as large as 4.5 ulps.*

There thus may be floating-point discrepancies between the current Intel embedded processors (based on i386 / i387) and the current Pentium workstation processors.

To summarise, one should not expect consistent behaviour on transcendental functions across libraries, processor manufacturers or models.

[7] Let us consider the round-to-nearest case. If $|r| \leq M_{\text{double}}$, then the x87 in double-precision mode rounds exactly like IEEE-754 double-precision arithmetics. If $M_{\text{double}} < |r| < 2^{E_{\max}}(2 - 2^{-p})$, then, according to the round-to-nearest rules (including "round-to-even" for $r = 2^{E_{\max}}(2 - 2^{-p})$), r is rounded to $\pm M_{\text{double}}$ on the x87, which is correct with respect to IEEE-754. If $|r| \leq 2^{E_{\max}}(2-2^{-p})$, then rounding r results in an overflow. The cases for the other rounding modes are simpler.

[8] `gcc` has an option `-mno-fused-madd` to turn off the use of this instruction.

[9] The latter behaviour is triggered by option `-ffast-math`. The documentation for this function says *it can result in incorrect output for programs which depend on an exact implementation of IEEE or ISO rules/specifications for math functions*.

[10] Personal communication.

[11] Let us now consider a rational approximation of π, i.e. integers p and q such that $p/q \approx \pi$ (such an approximation can be obtained by a continued fraction development of π [Wei]). $\sin(p) \approx \sin(q\pi) = 0$. If p', the result of reduction modulo 2π of p, is imprecise by a margin of ϵ ($\exists k\ p' - p = \epsilon + 2k\pi$), then $\sin(p') - \sin(p) \approx \epsilon$ ($\sin(x) \sim x$ close to 0). Such inputs are thus good candidates to illustrate possible lack of precision in the algorithm for reduction modulo 2π.

4.2.2 Non-standard rounding modes

We also have found that floating-point libraries are often poorly tested in "uncommon" usage conditions, such as rounding modes different from "round-to-nearest". This is especially of interest for implementors of static analysers (Sect. 6.1), since some form of interval arithmetic will almost certainly be used.

FreeBSD 4.4 provides the `fpsetround()` function to set the rounding mode of the processor. However, the `printf()` standard printing function of the C library does not work properly if the processor is set in round-to-$+\infty$ mode: when one attempts to print very large values (such as 10^{308}), one can get garbage output (such as a semicolon in a location where a digit should be, e.g. `:e+307`).

On GNU libc 2.2.93 on IA32 processors, the `fesetround()` function only changed the rounding mode of the x87 FPU, while the `gcc` compiler also offered the possibility of compiling for SSE.

On GNU libc 2.3.3 on AMD64, computing x^y using the `pow()` function in round-to-$+\infty$ mode can result in a segmentation violation for certain values of x and y, e.g. $x =$ `0x1.3d027p+6` and $y =$ `0x1.555p-2`. As for the `exp()` exponential function, it gives a result close to 2^{502} on input 1, and a negative result on input `0x1.75p+0`. The problems were corrected in version 2.3.5.

4.3 Compiler issues

The C standard, by default, allows the compiler some substantial leeway in the way that floating-point expressions may be evaluated. While outright simplifications based on operator associativity are not permitted, since these can be very unsound on floating-point types [ISO99, §5.1.2.3 #13], the compiler is for instance permitted to replace a complex expression by a simpler one, for instance using compound operators (e.g. the fused multiply-and-add in Sect. 4.1) [ISO99, 6.5]:

> *A floating expression may be contracted, that is, evaluated as though it were an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method. [...] This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented.*

While some desirable properties of contracted expressions [ISO99, §F.6] are requested, but no precise behaviour is made compulsory.

Because of the inconveniences that discrepancies can create, the standard also mandates a special directive, `#PRAGMA STDC FP_CONTRACT` [ISO99, §7.12.2], for controlling whether or not such contractions can be performed. Unfortunately, while many compiler will contract expressions if they can, few compilers implement this pragma. As of 2005, `gcc` (v4.0) ignores the pragma with a warning, and Microsoft's Visual C++ handles it as a recent addition.

We have explained how, on certain processors such as the x87 (Sect. 3.1), it was possible to change the precision of results by setting special flags — while no access to such flags is mandated by the C norm, the possibility of various precision modes is acknowledged by the norm [ISO99, F.7.2]. Furthermore, IEEE-754 mandates the availability of various rounding modes (Sect. 2.1); in addition, some processors offer further flags that change the behaviour of floating-point computations.

All changes of modes are done through library functions (or inline assembly) executed at runtime; at the same time, the C compiler may do some computations at compile time, when these modes are not reflected.

> *During translation the IEC 60559 default modes are in effect: The rounding direction mode is rounding to nearest. The rounding precision mode (if supported) is set so that results are not shortened. Trapping or stopping (if supported) is disabled on all floating-point exceptions. [...] The implementation should produce a diagnostic message for each translation-time floating-point exception, other than inexact; the implementation should then proceed with the translation of the program.*

In addition, compilers may test or change the floating-point status or operating modes using library functions, or even inline assembly. If the compiler performs code reorganisations, then some results may end up being computed before the applicable rounding modes are set. For this reason, the C norm introduces `#pragma STDC FENV_ACCESS ON/OFF` [ISO99, §7.6.1]:

> *The `FENV_ACCESS` pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes. [...] If part of a program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the `FENV_ACCESS` pragma off, the behaviour is undefined. The default state (on or off) for the pragma is implementation-defined. [...] The purpose of the `FENV_ACCESS` pragma is to allow certain optimisations that could subvert flag tests and mode changes (e.g., global common subexpression elimination, code motion, and constant folding). In general, if the state of `FENV_ACCESS` is off, the translator can assume that default modes are in effect and the flags are not tested.*

Another effect of this pragma is to change how much the compiler can evaluate at compile time regarding constant initialisations `ON`. [ISO99, F.7.4, F.7.5]. If it is set to `OFF`, the compiler can evaluate floating-point constants at compile time, whereas if they had been evaluated at runtime, they would have resulted in different values (because of different rounding modes) or floating-point exception. If it is set to `ON`, the compiler may do so only for static constants — which are generally all evaluated at compile time and stored as a block of constants in the binary code of the program.

Unfortunately, as per the preceding pragma, most compilers do not recognise this pragma. There may, though, be some compilation options that have some of the same effect. The norm discusses which optimisations may or may not be applied [ISO99, F.8].

4.4 Input/output issues

Another possible compilation issue is how compilers interpret constants in source code. The C norm states:

> *For decimal floating constants, and also for hexadecimal floating constants when `FLT_RADIX`[12] is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants when `FLT_RADIX` is a power of 2, the result is correctly rounded.*

This means that two compilers on the same platform may well interpret the same floating-point decimal literal in the source code as different floating-point value. (And that is even if both compilers follow C99 closely, which few current compilers do.) Similar limitations apply to the behaviour of the C library when converting from decimal representations to floating-point variables [ISO99, §7.20.1.3].

There exist few guarantees as to the precision of results printed in decimal in the C norm [ISO99, §7.19.6.1, F.5]. IEEE-754, however, mandates some guarantees [IEC89, IEE85, §5.6], such that printing and reading back the values should yield the same numbers, withing certain bounds. However, we have seen that the standard C libraries of certain systems are somewhat unreliable; thus, one may prefer not to trust them on accuracy issues. Printing out

[12] `FLT_RADIX` is the radix of floating-point computations, thus 2 on IEEE-754 systems. There currently exist few systems with other radixes.

exact values and reading them is important for replaying exact test cases. Again, we stress that reading and printing floating-point numbers accurately is a non-trivial issue if the printing base (here, 10) is not a power of the computation base (binary in the case of IEEE-754) [Cli90, SW90].

In order to alleviate this, we suggest the use of hexadecimal floating-point constants, which are interpreted exactly; unfortunately, many older compilers do not support these; also, in order to print floating-point values as hexadecimal easily, `printf` and associated functions have to support `%a` and `%A`, which is not yet the case of all current C libraries.

5. Example

Arguably, many of the examples we gave in the preceding sections, though correct, are somewhat contrived: they discuss small discrepancies, often happening with very large or very small inputs. In this section, we give a complete and realistic example of semantic problems related to differences between floating-point implementations (even, dependent on compilation options!). It consists of two parts:

1. an algorithm for computing a modulo (such as mapping an angle into $[-180, 180]$ degrees), inspired by an algorithm found in an embedded system;
2. possible implementations of tabulated functions.

The composition of the two give seemingly innocuous implementations of angular periodic functions... which crash for certain specific values of the inputs on certain platforms.

Given x, m and M ($m < M$), we want to compute r such that $r - x$ is an integer multiple of $M - m$ and $m \leq r \leq M$. The following algorithm, adapted from code found in a real-life critical system, is correct if implemented over the real numbers:

```
double modulo(double x, double mini, double maxi) {
  double delta = maxi-mini;
  double decl = x-mini;
  return x - floor(decl/delta)*delta;
}

int main() {
  double m = 180.;
  double r = modulo(nextafter(m, 0.), -m, m);
}
```

We recall that `floor(`x`)` is the greatest integer less than x, and that `nextafter(`a`, `b`)` is the next representable `double` value from a in the direction of b.

The above program, compiled by `gcc` 3.3 with optimization for the x87 target, yields an output $r \simeq 179.9999999999999997158$. In such a mode, variable `q` is cached in a extended precision register; `q` $= 1 - \epsilon$ with $\epsilon \simeq 7.893.10^{-17}$. However, if the program compiled without optimization, `decl` and `q` are saved to, then reloaded from, main memory as a double precision variable; in the process, `q` is rounded to 1. The program then returns $r \simeq -180.00000000000002842$ outside the specified bounds.

Simply rewriting the code as follows makes the problem disappear (because the compiler holds the value in a register):

```
double modulo(double x, double mini, double maxi) {
  double delta = maxi-mini;
  return x - floor((x-mini)/delta)*delta;
}
```

This is especially troubling since that code and the above look semantically equivalent at first sight.

If we simply add a logging statement (`printf()`) after the computation of `decl`, even if optimisation is turned on, then the return value is also outside the specified bounds. This is due to the forced flushing of the floating-point registers into main memory with less precision.

Interestingly enough, the ASTRÉE static analyser gives a lower bound slightly lower than -180. on such code, and thus signals to the user some suspicious behaviour. This is because, as described in 6.1, it estimates error bounds according to IEEE-754 worst-case behaviour. In comparison, simply performing unit testing on a IA32 PC will not discover the problem.

Now, one could argue that the odds of landing exactly on `nextafter(180., 0.)` are very rare. Assuming an embedded control routine executed at 100 kHz, 24 hours a day, and a uniform distribution of values in the $[-180, 180]$ interval, such a value should happen once every 4000 years or so on a single unit.[13] However, in the case of a mass-produced system (an automobile model, for instance), this argument does not hold. If hundreds of thousands of systems featuring a buggy component are manufactured every year, there will be real failures happening when the systems are deployed — and they will be very difficult to recreate and diagnose. If the system is implemented in single precision, the odds are considerably higher with the same probabilistic assumptions: a single 100 Hz system would break down twice a day.

Now, it seems that a slight error such as this should be of no consequence: the return value is only extremely slightly outside the bounds. However, let us consider a typical application of computing modulos: computing some kind of periodic function, for instance depending on an angle. Such a function is likely to contain trigonometric operators and other operations costly and complex to compute; they are often evaluated by looking them up a table.

One implementation sometimes found is to have some big array of constants, perhaps computed using some external application, and lookup the array at various offsets:

```
val = bigTable[r + 180 + FUNCTION_F_OFFSET];
```

This means an implicit truncation of `r+180+FUNCTION_F_OFFSET` to 0; if r is a little below -180, then this truncation will evaluate to `FUNCTION_F_OFFSET`-1. The table lookup can then yield whatever value is at this point in memory, possibly totally out of the expected range.

Another example is a table lookup with interpolation:

```
double periodicFunction(double r) {
  double biased = r+180;
  double low = floor(biased);
  double delta = biased-low;
  int index = (int) low;
  return ( table[index]*(1-delta)
         + table[index+1]*delta );
}
```

If r is slightly below 0, the value return will depend on `table[-1]`, that is, whatever is in memory before `table`. This can result in a segmentation fault (access to a unallocated memory location), or, more annoyingly, in reading whatever is at that location, including special values such as *NaN* or $\pm\infty$, or even simply very large values. With `table[-1]` $= 10^{308}$ and `table[0]` $= 0$, the above program outputs approximately 2.8×10^{294} — a value probably too large for many algorithms to follow.

Such kinds of problems is that they are extremely difficult to reproduce if they are found in practise — they may result in program crashes or nondeterministic behaviour for very rare input values. Furthermore, they are not likely to be elicited by random

[13] For 180, the unit at the last position is $\delta = 2^{-45}$; all reals in $]180 - 3/2\delta, 180 - 1/2\delta[$ are rounded to $180 - \delta$, thus the probability of rounding a random real in $[-180, 180]$ to this number is $360/\delta$.

testing.[14] Finally, the problem may disappear if the program is tested with another compiler, compilation options, or execution platform.

6. Implications for program verification

Program analysis, verification, testing and proving techniques all rely on having a somewhat well-defined concrete execution model. Thus, the issues discussed in the preceding sections may have a negative impact on such techniques; if handled improperly, these techniques may fail to discover real-life problems in compiled code. We shall distinguish the case of analysers based on abstract numerical domains [Min04b] and the case of testing or, more generally, analysis techniques based on exact replays of executions.

6.1 Static analysers based on abstract interpretation

Let us recall that static analysis consists in deriving results about possible program executions without actually running the program. In this section, we are only interested in *sound* analysis methods, that is, methods that will never claim that some property is verified whereas some possible program executions do not verify the property. We focus on methods based on *abstract interpretation*, a generic framework for giving an *over-approximation* of the set of possible program executions. [CC92] An *abstract domain* is a set of possible symbolic constraints with which to analyse programs (for instance, the *octagon abstract domain* [Min01] represents constraints of the form $\pm x \pm y \leq C$ where x and y are program variables and C is a number).

6.1.1 General issues

A common objection to the analysis of programs containing floating-point operations is that their behaviour is impossible to analyse soundly, or requires a human expert. On the one hand, it is indeed our experience that fine behaviour of floating-point programs (such as conditions on the least significant bits) is difficult to check; however, most common programs do not rely on such fine behaviour in order to work (and arguably they should not, in general, since such reliance is bad for portability). Also, certain numerical algorithms, particularly those with stability conditions such as IIR filters, need specific abstract domains to be properly analysed. [Fer04].

A naive approach to the concrete semantics of programs running on "IEEE-754-compatible" platforms is to consider that a `+`, `-`, `*` or `/` sign in the source code, between operands of type `float` (resp. `double`), corresponds to a IEEE-754 $\oplus$, $\ominus$, $\otimes$, $\oslash$ operation, with single-precision (resp. double-precision) operands and result. As we have seen, this does not hold in many common cases, especially on the x87 (Sect. 3.1).

A second approach is to analyse assembly or object code, and take the exact processor-specified semantics as the concrete semantics for each operation. This is likely to be more precise, but also quite cumbersome. In addition, as we have seen, some "advanced" functions in floating-point units may have been defined differently in successive generations of processors, so we cannot rule out discrepancies.

A third approach is to encompass all possible semantics of the source code into the analysis. Static analysis methods based on abstract interpretations are well-suited for absorbing such "implementation-defined" behaviours while still staying sound.

The abstractions discussed below aim at providing a sound over-approximation of all possible concrete semantics. However, they do not point where precision is lost in the concrete computation, which is a more ambitious tasks requiring specific abstractions [Gou01, Mar02b, Mar02a]. In this case, one has to care whether the semantics of introduced error fits reality, including the quirks of the various implementations.

6.1.2 Intervals

The most basic domain for the analysis of floating-point programs is the interval domain [BCC$^+$02, Min04a, Min04b]: to each quantity x in the program, attach an interval $[m_x, M_x]$ such that in any concrete execution, $x \in [m_x, M_x]$. Such intervals $[m_x, M_x]$ can be computed efficiently in a static analyser for software running on a IEEE-754 machine if one runs the analyser on a IEEE-754 machine, by doing interval arithmetics on the same data types as used in the concrete system, or smaller ones.

Let us note floating-point addition (with given types, rounding and precision) as $\oplus$. Fixing x, $y \mapsto x \oplus y$ is monotonic. It is therefore tempting, when implementing interval arithmetics, to approximate $[a, b] \oplus [a', b']$ by $[a \oplus a', b \oplus b']$. Unfortunately, this is not advisable unless one is really sure that computations in the program to be analysed are really done with the intended precision (no extended precision temporary values, as common on x87 3.1), do not suffer from double rounding effects (see 3.2), and do not use compound operations instead of atomic elementary arithmetic operations. It is much safer to compute upper bounds in round-to-$+\infty$ mode, and lower bounds in round-to-$-\infty$ mode.[15] This is what we do in ASTRÉE [CCF$^+$05]. Note, however, that on some systems, rounding-modes other than round-to-nearest are poorly tested and may fail to work properly (see Sect. 4.2.2); the implementors of analysers should therefore pay extra caution in that respect.

Another area where one should exercise caution is strict comparisons. A simple solution for abstracting the outcome for `x` of `x < y`, where `x` $\in [m_x, M_x]$ and `y` $\in [m_y, M_y]$, is to take $M'_x = \min(M_x, m_y)$, as if one abstracted `x <= y`. Unfortunately, this is not sufficient to deal with programs that use special floating-point values as markers (say, 0 is a special value meaning "end of table"). One can thus take $M'_x = \min(M_x, \text{pred}(m_y))$ where $\text{pred}(x)$ is the largest floating-point number less than x *in the exact floating-point type used* (which may be very difficult for optimised code on the x87, because of the unpredictable scheduling of conversions from `long double`).

6.1.3 Relational domains

The interval domain, while simple to implement, suffers from not keeping relations between variables. However, relational abstract domains tend to be designed for ideal data structures (ordered rings, etc.); it is thus necessary to bridge the gap between these ideal abstract structures and the concrete execution. One successful avenue in that domain is to consider that the "real" execution and the floating-point execution differ by a small error, which we can bound [Min04a, Min04b]. However, in doing so, one must be careful not to be too optimistic!

Recall that in straight IEEE-754 round-to-nearest mode, it is possible to bound the error as $|x - r(x)| \leq \frac{1}{2}\max(\varepsilon_{\text{abs}}, \varepsilon_{\text{last}}.|x|)$ (see 2.3). It is thus tempting to take this bound to analyse floating-point programs running in round-to-nearest mode (by far the most common case). However, double rounding, as incurred sometimes in default compilation modes on x87, can result in slightly large errors (see 3.2). The other rounding modes are bounded as follows

[14] The example described here actually demonstrates the importance of *non-random* testing: that is, trying values that "look like they might cause problems"; that is, typically, values at or close to discontinuities in the mathematical function implemented, special values, etc.

[15] Changing the rounding mode may entail significant efficiency penalties. A common trick is to set the processor in round-to-$+\infty$ mode permanently, and replace $x \oplus_{-\infty} y$ by $-((-x) \oplus_{+\infty} (-y))$ and so on.

$|x - r(x)| \leq \max(\varepsilon_{\text{abs}}, \varepsilon_{\text{last}}.|x|) \leq \varepsilon_{\text{abs}} + \varepsilon_{\text{last}}.|x|$; this bound is thus safe for any mode. This is the bound we use in ASTRÉE.

Other kinds of relational domains propagate pure constraints or guards. Intuitively, if some arithmetic condition is true on some variables x and y, and neither x nor y are suppressed, then the condition continues to hold later. However, we have seen in Sect. 3 that one should beware of "hidden" rounding operations, which may render certain propagated identities invalid. In addition, we have also seen that certain equivalences such as $x \ominus y = 0 \iff x = y$ are not necessarily valid, depending on whether certain modes such as flush-to-zero are active.[16]

6.2 Testing

In the case of embedded system development, the development platform (typically, a PC running some variant of IA32 or AMD64 processors) is not the same as the target platform (typically, a microcontroller). Often, the target platform is much slower, thus making extensive unit testing time-consuming; or there may be a limited number of them for the development group. As a consequence, it is tempting to test or debug numerical programs on the development platform. We have shown that this can be a risky approach, even if both the development platform and the target platform claim to be "IEEE-754 compatible".

In some cases, even if the testing and the target platform are identical, the final result may depend on the vagaries of compilation. Even inserting "monitoring" instructions can affect the final result, because these can change register allocation, which can change the results of computations on the x87. This is especially an issue since it is common to have "debugging-only" statements excluded from the final version loaded in the system. One should be particularly cautious on platforms, such as IA32 processors with the x87 floating-point unit, where the results of computations can depend on register scheduling. In the case of the x87, it is possible to limit the discrepancies (but not totally eradicate them) by setting the floating-point unit to 53-bit mantissa precision, if one computes solely with IEEE-754 double precision numbers.

Static analysis techniques where concrete traces are replayed inside the analysis face similar problems. One has to have an exact semantic model of the program to be analysed as it runs on the target platform.

7. Conclusion

Despite the claims of conformance to standards, common software/hardware platforms may exhibit subtle differences in floating-point computations. These differences pose special problems for unit testing or debugging, unless one uses exactly the same object code as the one executed in the target environment — which is difficult, since, on some platforms, merely adding logging or debugging statements can cause the final results to change, even though the added statements should not change the result according to the "naive" semantics of the source code.

When one relies on exact predictability and reproducibility of computations, a norm such as IEEE-754 is of paramount importance. However, in practise, many processors and compilers advertised as "IEEE-754 compatible" do *not* implement strict IEEE-754 semantics — or at least do not implement them by default. Compiler writers should try to offer compilation options that minimise, or even suppress, variations in program outputs caused by the insertion of logging statements or other issues related to register scheduling or optimisation.

These differences, however, are simpler to handle conservatively for abstract-interpretation-based static analysers, because they can be folded into the abstraction inherent in those approaches.

Acknowledgments

We thank Patrick Cousot, Jérôme Feret, David Madore and Antoine Miné for their helpful comments and suggestions.

References

[Adv05] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, 3.10 edition, March 2005.

[AG97] Andrew Appel and Maia Ginsburg. *Modern compiler implementation in C*. Cambridge University Press, revised and expanded edition, December 1997.

[BCC+02] 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. In *The Essence of Computation: Complexity, Analysis, Transformation*, number 2566 in LNCS, pages 85–108. Springer, 2002.

[BCC+03] 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 *PLDI*, pages 196–207. ACM, 2003.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 2-3(13):103–179, 1992.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP*, number 3444 in LNCS, pages 21–30, 2005.

[Cli90] William D. Clinger. How to read floating point numbers accurately. In *PLDI*, pages 92–101. ACM, 1990.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[FdC00] Samuel A. Figueroa del Cid. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. PhD thesis, New York University, 2000.

[Fer04] Jérôme Feret. Static analysis of digital filters. In *ESOP*, number 2986 in LNCS. Springer, 2004.

[Fre] Free Software Foundation. *The GNU compiler collection*.

[Fre01a] Freescale Semiconductor, Inc. *MPC750 RISC Microprocessor Family User's Manual*, December 2001. MPC750UM/D.

[Fre01b] Freescale Semiconductor, Inc. *Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture*, December 2001. MPCFPE32B/AD.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison Wesley, 1st edition, 1996.

[GJSB00] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[Gou01] Éric Goubault. Static analyses of floating-foint operations. In *SAS*, volume 2126 of *LNCS*. Springer, 2001.

[IEC89] IEC. *International standard – binary floating-point arithmetic for microprocessor systems*, 2nd edition, 1989. IEC-60559.

[IEE85] IEEE. *IEEE standard for Binary floating-point arithmetic for microprocessor systems*, 1985. ANSI/IEEE Std 754-1985.

[16] There is still the possibility of considering that $x \ominus y = 0 \implies |x-y| < 2^{E_{\min}}$.

[Int97] Intel Corp. *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 1997. order number 243190.

[Int05] Intel Corp. *IA-32 Intel Architecture Software Developers Manual Volume 1: Basic Architecture*, September 2005. order number 253665-017.

[ISO99] ISO/IEC. *International standard – Programming languages – C*, 1999. standard 9899:1999.

[KD98] William Kahan and Joseph D. Darcy. How Java's floating-point hurts everyone everywhere. Available online, March 1998.

[Mar02a] Matthieu Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *ESOP*, number 2305 in LNCS. Springer, 2002.

[Mar02b] Matthieu Martel. Static analysis of the numerical stability of loops. In *SAS*, number 2477 in LNCS. Springer, 2002.

[Min01] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, pages 310–319. IEEE CS Press, October 2001.

[Min04a] Antoine Miné. *Domaines numriques abstraits faiblement relationnels*. PhD thesis, École polytechnique, 2004.

[Min04b] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[Pan98] Java Grande Forum Panel. Java grande forum report: Making java work for high-end computing, November 1998.

[Sun] Sun Microsystems. *KVM Porting Guide*.

[Sun01] Sun Microsystems. *Numerical Computation Guide*, 2001.

[SW90] Guy L. Steele Jr. and Jon L. White. How to print floating-point numbers accurately. In *PLDI*, pages 112–126. ACM, 1990.

[VCV97] D. Verschaeren, A. Cuyt, and B. Verdonk. On the need for predictable floating-point arithmetic in the programming languages Fortran 90 and C/C++. *SIGPLAN Not.*, 32(3):57–64, 1997.

[Wei] Eric W. Weisstein. Continued fraction. From MathWorld.