An efficient rounding boundary test for pow(x,y) in double precision

Christoph Lauter, Vincent Lefèvre

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

An efficient rounding boundary test for `pow(x,y)` *in double precision*

Christoph Quirin Lauter,
Vincent Lefèvre

September 4, 2007

Research Report N° RR2007-36

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

An efficient rounding boundary test for `pow(x,y)` in double precision

Christoph Quirin Lauter, Vincent Lefèvre

September 4, 2007

Abstract

The correct rounding of the function pow : $(x, y) \mapsto x^y$ is currently based on Ziv's iterative approximation process. In order to ensure its termination, cases when x^y falls on a rounding boundary must be filtered out. Such rounding boundaries are floating-point numbers and midpoints between two consecutive floating-point numbers.

Detecting rounding boundaries for pow is a difficult problem. Previous approaches use repeated square root extraction followed by repeated square and multiply. This article presents a new rounding boundary test for pow in double precision which resumes to a few comparisons with pre-computed constants. These constants are deduced from worst cases for the Table Maker's Dilemma, searched over a small subset of the input domain. This is a novel use of such worst-case bounds.

The resulting algorithm has been designed for a fast-on-average correctly rounded implementation of pow, considering the scarcity of rounding boundary cases. It does not stall average computations for rounding boundary detection. The article includes its correction proof and experimental results.

Keywords: floating-point arithmetic, correct rounding, power function

Résumé

L'arrondi correct de la fonction pow : $(x, y) \mapsto x^y$ se base actuellement sur le processus d'approximation itératif de Ziv. Afin de garantir la terminaison de ce processus, les cas où x^y tombe sur une frontière d'arrondi doivent être filtrés. Les frontières d'arrondi sont formées par les nombres flottants et les milieux entre deux nombres flottants consécutifs.

La détection de frontières d'arrondi de pow est un problème difficile. Les approches antérieures utilisent des extractions de racines carrées répétées suivies d'une boucle de mises au carré et de multiplications. Cet article présente un nouveau test de frontière d'arrondi pour pow en double précision qui se contente de quelques comparaisons avec des constantes pré-calculées. Ces constantes ont été déduites des pires cas du Dilemme du fabricant des tables, recherchés sur un petit sous-ensemble du domaine d'entrée. Ceci est une utilisation nouvelle de bornes de pires cas.

L'algorithme présenté a été développé pour une implémentation de pow correctement arrondie, rapide en moyenne, en considérant la rareté des cas sur les frontières d'arrondi. Il ne retarde pas les calculs de cas moyens pour la détection de cas sur les frontières d'arrondi. Cet article inclut la preuve de correction de l'algorithme ainsi que des résultats expérimentaux.

Mots-clés: arithmétique flottante, arrondi correct, fonction power

1 Introduction

Correct rounding of elementary functions $f : \mathbb{R} \to \mathbb{R}$ such as exp, sin, $\log_2$, extends bit-by-bit portability of the IEEE 754 standard for binary floating point arithmetic [1]. Correct rounding means returning a floating-point result rounded as if infinite intermediate precision were used to evaluate f. Its importance, its impact and, in particular, its feasibility have been shown in the last years [17, 14, 7, 4, 5, 11].

Correctly rounding implementations [3, 7, 17, 16] offering high performance currently rely on two main approaches, that can be used together. In the first technique due to Ziv, an iterative process increases the accuracy of an approximation to the given function. Eventually, it can be ensured that rounding the approximation is equivalent to rounding the infinitely exact value [17, 14]. In the second approach, the worst-case accuracy in Ziv's iterations is pre-calculated for some target precision such as IEEE 754 double precision [11, 15], so that not only average but also worst-case performance can be brought to a high level [4, 5].

In order to guarantee the termination of Ziv's iterative process, it is necessary to detect cases when the image $f(x)$ of a function lies exactly on a so-called rounding boundary [17, 14, 13]. A rounding boundary is a point where the rounded value abruptly changes: values less than the boundary are rounded downward and values greater than it are rounded upward. Special rules, like the ties-to-even rule of the IEEE 754 standard, apply if the value lies exactly on the boundary.

For common elementary, transcendental functions, such as for example exp, sin, $\log_2$, this detection is easy: rounding boundaries are rational numbers, whereas the images of these functions on rational values are transcendental except for a few well-known arguments [14, 8]. Thus, only few values remain to be filtered. For instance, $\exp(x)$ is rational only for $x = 0$ and $\log_2(x)$ is rational only for integer powers of 2.

For the function pow : $(x, y) \mapsto x^y$ the situation is different. First, the function is bivariate. This currently makes worst-case accuracy computation unfeasible for double precision in reasonable time. Current techniques are opposed to a combinational explosion [15, 10]. Hence, the correct rounding of pow is based only on Ziv's iterative approach [13, 16, 17]. Second, the images $\text{pow}(x, y) = x^y$ are not floating-point numbers of some precision for some floating-point numbers x and y, and potentially fall on rounding boundaries. Consider for example $1296^{0.75} = 216$. Nevertheless, the rounding boundary cases (x, y) – in a given precision – form a complicated set. For example, x^y is rational for x rational and y integer or, more difficult, for x a repeatedly perfect square and y the reciprocal of the corresponding integer power of 2. However, not all rationals are rounding boundaries. The detection of these rounding boundary cases of pow requires a particular algorithm.

This paper addresses this rounding boundary detection problem for pow in binary floating-point arithmetic as specified by the IEEE 754 standard. The problem is considered for IEEE 754 double precision and all rounding modes defined by the standard: the default mode round-to-nearest-ties-to-even and the three directed rounding modes [1].

The problem is not new: different detection algorithms have already been proposed in the MPFR library [7, 13], in Sun's `libmcr` [16] and in IBM's (Ziv's) `libultim` that is now integrated in the GNU C Library (`glibc`)[1] [17]. In all these three previous approaches, the detection algorithm performs relatively expensive computations at run-time. Complex tests ensure that these computations are all error-free. A novelty in our approach is that these computations are replaced with simple tests with constants and approximations that are already available in an implementation of pow. Typically we replace a repeated square root extraction and testing process followed with a square and multiply loop by eight comparisons with pre-computed constants.

A second novelty in our work is how these constants can be pre-computed. Surely, worst-case bounds for the correct rounding cannot be computed for the function pow on its whole definition domain in double precision in reasonable time [15, 10]. Nevertheless, worst-case bounds can be computed for a subset of the domain [9]. This subset principally contains arguments (x, n) of integer powers x^n of a double precision number x for small integers n and pre-images $(x, 2^{-F} \cdot n)$

[1] available at `http://www.gnu.org/software/libc/`

of small 2^F-th roots of a double precision number x raised to some very small integer power: $\left(x^{2^{-F}}\right)^n$. We observe and prove that all cases when x^y falls on a rounding boundary must lie in such a small subset of the double precision numbers. Further, we show that we can detect rounding boundary cases using approximations better than the worst-case: if an approximation to x^y is provably twice as near to a rounding boundary than an inexact case can ever be, the true value of x^y is exactly on the rounding boundary.

Purpose of the design of our algorithm has not been just a new application of worst-case search results. The algorithm has been designed as a basic brick for a fast correctly rounded implementation of pow. We show that in random input, rounding boundary cases are rare. This particularly holds for the default rounding mode. Our algorithm allows one to get high average performance by not stalling not-rounding-boundary-cases on the critical path and to pitch on the rounding boundary cases essentially for free after the second iteration in Ziv's process.

This paper is organized as follows: In Section 2, we analyze previous correct rounding implementations of pow. After fixing notations (Section 2.1), we give an overview on how Ziv's iteration technique works and why its termination is conditioned by detecting rounding boundary cases (Section 2.2). We analyze then the general, previously proposed techniques for rounding boundary detection for pow (Section 2.3) before sketching the algorithms in Sun's `libmcr`, Ziv's `libultim` and MPFR (Section 2.4). We expose our approach in Section 3. We show here analyses of the number of rounding boundary cases (Section 3.1). We expose our technique using worst-cases (Section 3.2) and show what algorithm can be used for computing these worst cases (Section 3.3). Finally we present our algorithm (Section 3.4). We give a sketch of the correctness proof of the algorithm in Section 4. Before concluding in Section 6, we show performance results in Section 5.

2 Correct rounding implementations of pow

2.1 Notations

Throughout this paper we work with binary floating point numbers. In our formalization [2] a floating point number $2^E \cdot m$ consists of an exponent E and a significand m, both signed integers. We make abstraction from special data like infinities or Not-a-Numbers (NaNs). We denote the set of floating-point numbers of precision k with unbounded exponent range as:

$$\mathbb{F}_k = \left\{2^E \cdot m \mid E \in \mathbb{Z},\ m \in \mathbb{Z},\ 2^{k-1} \leq |m| \leq 2^k - 1\right\} \cup \{0\} \ .$$

IEEE 754 double precision floating-point numbers have a bounded exponent range [1]. They correspond to the set

$$\begin{aligned} \mathbb{D} &= \left\{2^E \cdot m \mid E \in \mathbb{Z},\ -1074 \leq E \leq 971,\ 2^{52} \leq |m| \leq 2^{53} - 1\right\} \\ &\cup \left\{2^{-1074} \cdot m \mid m \in \mathbb{Z},\ |m| \leq 2^{52} - 1\right\} \ . \end{aligned}$$

Floating-point numbers in $\mathbb{D}$ for which m varies between 2^{52} and $2^{53}-1$ are called normal numbers. The remaining numbers are called subnormal numbers. Remark that $\mathbb{D} \subseteq \mathbb{F}_{53}$ [2].

We consider the IEEE 754 rounding modes round-down, round-up, round-to-zero and the default mode round-to-nearest-ties-to-even [1]. We denote the operation of rounding as a function $\circ_k : \mathbb{R} \to \mathbb{F}_k$. We use the symbol $\diamond_k : \mathbb{R} \to \mathbb{F}_k$ if a distinction between two rounding modes is necessary.

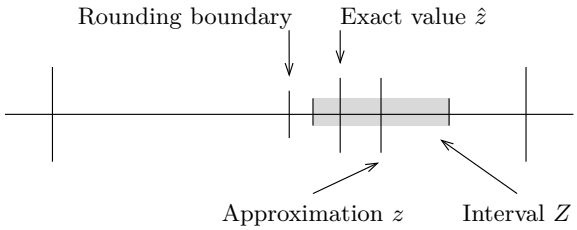
Rounding functions are discontinuous. The discontinuity points of a rounding function are the rounding boundaries of the corresponding rounding mode. For the modes $\circ_k$ round-down, round-up and round-towards-zero, the set of the rounding boundaries is the set $\mathbb{F}_k$ [11]. We refer to such a case as an exact case. For the rounding round-to-nearest mode $\circ_k$, the set of the rounding boundaries is formed by the midpoints of two consecutive floating-point numbers in $\mathbb{F}_k$ [11]. We refer to such a case as a midpoint case.

The midpoints of numbers in $\mathbb{F}_k$ are numbers in $\mathbb{F}_{k+1}$ with odd significand. Since $\mathbb{F}_k \subset \mathbb{F}_{k+1}$, the rounding boundaries of all considered rounding-modes $\circ_k$ lie in $\mathbb{F}_{k+1}$. Testing whether $\mathrm{pow}(x, y)$ is a rounding boundary case in double precision hence means computing the predicate

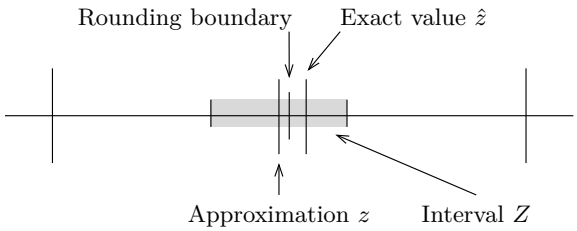
$$RB(x, y) = (x^y \in \mathbb{F}_{54}) \ .$$

2.2 The Table Maker's Dilemma and Ziv's correct rounding technique

The correct rounding of a non-rational function f is subject to the Table Maker's Dilemma [14]. The unknown, exact value $\hat{z} = f(x)$ of the function is approximated by z with an error δ. It is only known that $\hat{z}$ lies in an interval $Z = [z - \delta; z + \delta]$ around the approximation z. If no rounding boundary lies in Z, all values $\tilde{z} \in Z$ round to the same value $\circ(\tilde{z})$. Hence, rounding the approximation z gives to correct rounding: $\circ(\hat{z}) = \circ(z)$ (see Figure 1(a)). On the other hand, if a rounding boundary lies in Z, there is a doubt: some values $\tilde{z} \in Z$ round up, other values $\tilde{z}$ round down (see Figure 1(b)) [14].



(a) Easy to round case



(b) Hard to round case

Figure 1: Table Maker's dilemma

Ziv's correct rounding technique [17] iteratively decreases the approximation error δ when the rounding cannot be decided. If the exact value $\hat{z}$ is not a rounding boundary, there is some non-zero distance between $\hat{z}$ and the nearest boundary (see Figure 1(a)). With decreasing δ, the width of $Z = [z - \delta; z + \delta]$ eventually becomes less than this distance and correct rounding becomes possible. In the case when $\hat{z}$ lies on a rounding boundary, the iteration does not terminate. With a non-zero approximation error δ, there will be decreasingly smaller intervals Z around the rounding boundary $\hat{z}$. Nevertheless, they will never reach $\hat{z}$. The iteration repeatedly misinterprets a rounding boundary case as a hard-to-round case (i.e. a non-rounding-boundary case whose image is very close to a rounding boundary). Hence, rounding boundary cases must be filtered out [17].

High performance is obtained in the average case with this technique. Values $\hat{z} = f(x)$ can be considered as randomly distributed around rounding boundaries [17, 14, 4, 6]. On average, the first fast approximation step suffices for correct rounding. Nevertheless, the worst-case timing of Ziv's iteration remains unknown. In order to ensure this worst-case timing, the number of iteration must be statically bounded. This is possible if the smallest, worst-case distance between a value $\hat{z} = f(x)$ and the nearest rounding boundary can be computed. This is currently feasible for most univariate elementary functions in double precision. For the bivariate $\mathrm{pow} : (x, y) \mapsto x^y$ function, such worst-case researches are currently unfeasible in reasonable time for the whole double precision definition range [10, 15, 9]. In consequence, Ziv's iteration process must be used

with an a-priori unknown number of iterations. This implies the necessity of detecting rounding boundaries after some number of iterations in the process or before it (see Figure 1(b)).

Using approximations for some of the computations needed for rounding boundary detection is nevertheless possible. Detecting rounding boundary cases in double precision means computing the predicate $RB(x, y) = (x^y \in \mathbb{F}_{54})$. Let $\circ_{54}$ be rounding-to-nearest to 54 bits of precision. Since $\circ_{54}(x^y) \in \mathbb{F}_{54}$ holds, rounding boundary detection means testing whether $x^y = \circ_{54}(x^y)$. The rounding boundary detection reduces thus to some sort of equality test. Here, the following observation is important: Let $\diamond_{53}$ be a rounding to 53 bits in any rounding mode. The rounding $\circ_{54}(x^y)$ is never confronted to the Table Maker's Dilemma when the rounding $\diamond_{53}(x^y)$ is confronted to the Table Maker's Dilemma: since $\diamond_{53}(x^y)$ is confronted to the Table Maker's Dilemma, x^y lies near or exactly on a rounding boundary, which is a floating-point number in $\mathbb{F}_{54}$. In consequence, the value x^y is far from the middle of two consecutive floating-point numbers in $\mathbb{F}_{54}$ and can easily rounded to the nearest in $\mathbb{F}_{54}$ (see Figure 2).

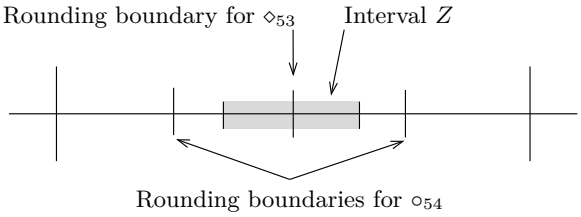


Figure 2: Using an approximation in the detection

A rounding boundary detection test launched after a first step in Ziv's iteration may thus use $\circ_{54}(x^y + \delta) = \circ_{54}(x^y)$ for the test whether $x^y = \circ_{54}(x^y)$. Further, once a rounding boundary case has been detected, i.e. $x^y = \circ_{54}(x^y)$, the correctly rounded value $\diamond_{53}(x^y)$ may be deduced from the approximation as $\diamond_{53}(\circ_{54}(x^y + \delta))$. Although the report [13] alludes this technique, it does not appear to be used in any known previous implementation.

2.3 General techniques for rounding boundary cases of pow

All previous rounding boundary tests for the function pow : $(x, y) \mapsto x^y$ use some basic properties of the arguments $x, y \in \mathbb{F}_{53}$ and the potential rounding boundary $x^y \in \mathbb{F}_{54}$. A variant of the corresponding algorithm with a proof sketch is already outlined in [13]. These basic properties imply some branching scheme. Typically the sign of some values and their exponents are tested. We reuse this preliminary branching scheme in our algorithm. Further, the basic properties yield bounds on particular values. In previous approaches, these bounds mainly guarantee the termination of an iteration. Our algorithm explicitly tests against these bounds. They further determine a domain, on which constants must be pre-computed for our algorithm. In Section 4 we therefore extend the proof sketch given in [13], in particular by proving such bounds.

Let us now consider these basic properties of rounding boundary cases x^y. Let z be a rounding boundary near x^y, i.e. $z = \circ_{54}(x^y)$. Remark that previous approaches do not explicitly compute z but merely suppose it to exist. The numbers x, y and z are floating-point numbers. Without lack of generality, x can be supposed to be positive. The numbers x, y and z can hence be written

$$\begin{aligned} x &= 2^E \cdot m \\ y &= 2^F \cdot n \\ z &= 2^G \cdot k \end{aligned}$$

where $E, F, G \in \mathbb{Z}$, $m, k \in 2\mathbb{N} + 1$ and $n \in 2\mathbb{Z} + 1$.

Testing a rounding boundary case, i.e. computing $RB(x, y) = (x^y = \circ_{54}(x^y))$ means thus checking if

$$2^{2^F \cdot E \cdot n} \cdot m^{2^F \cdot n} = 2^G \cdot k\,.$$

Since m is odd (see Section 4), this is equivalent to testing the two conditions

$$m^{2^F \cdot n} = k \quad (1)$$
$$2^F \cdot E \cdot n = G\,. \quad (2)$$

A distinction must now be made depending on the sign of n. If n is negative, $m^{2^F \cdot n} = k$ can be written

$$\left(m^{-n}\right)^{2^F} = \frac{1}{k}\,.$$

Since m, n, k and F are integers and $m, -n, k > 0$, this implies with the second condition that x must be an integer power of 2 and that $E \cdot y$ must be an integer G (see Section 4 for more details). This can easily be tested for in IEEE 754 double precision. Conversely, if these conditions are satisfied, then x^y is a integer power of two, i.e. one knows that x^y is an exact case (or an underflow or overflow).

On the other hand, if n is positive, the situation is more complicated. There are two alternatives depending on the sign of F. If F is negative, $m^{2^F \cdot n} = k$ can be written

$$\left(\sqrt[2^{-F}]{m}\right)^n = k\,.$$

Since n is odd and 2^{-F} is even (see Section 4), this reduces to testing whether there exists an integer $j \in \mathbb{N}$ such that

$$\sqrt[2^{-F}]{m} = j \quad (3)$$
$$j^n = k\,. \quad (4)$$

The test whether $\sqrt[2^{-F}]{m} \in \mathbb{N}$ is performed by previous approaches as repeated square root extraction and testing: since m is odd,

$$\sqrt[2^{-F}]{m} \in \mathbb{N} \Rightarrow \sqrt[2^{-F-1}]{\sqrt{m}} \in \mathbb{N} \Rightarrow \sqrt[2^{-F-1}]{m'} \in \mathbb{N} \wedge m' = \sqrt{m} \in \mathbb{N}\,.$$

The condition $\sqrt{m} \in \mathbb{N}$ is mainly tested by taking the floating-point square root of m and checking whether it is exact, i.e. produced without round-off. The termination of this iteration is ensured by a bound on m: m has at most 53 significant bits. Since upon each exact square root extraction the number of significant bits in m is halved at each step, no more than 5 iterations are possible: $53 < 2^{5+1}$ (see Section 4). While the bound $F \geq -5$ is still important in our algorithm, the square root extraction loop is no longer needed.

If F is positive or zero, $m^{2^F \cdot n} = k$ can be written

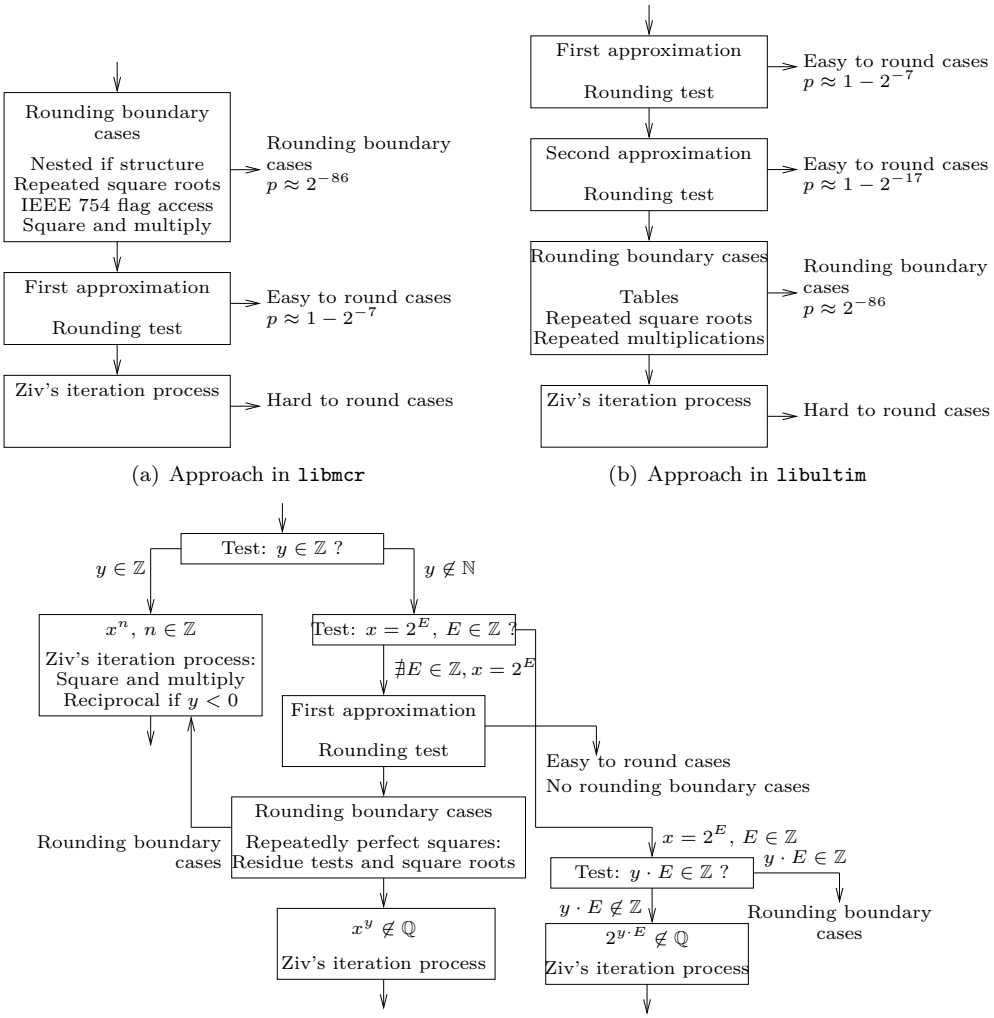
$$m^t = k$$

where $t = 2^F \cdot n = y \in \mathbb{N}$. In consequence, independently of the sign of F, the algorithm must test either $m^t = k$ or $j^n = k$ with $m, n, j, t, k \in \mathbb{N}$. Let $u^s = k$ be the test to be performed. Since k is bounded above by $2^{54} - 1$, the upper bound on s depends on u. Previous approaches store these bounds in a table or branching structure. Eventually, u^s is computed by a multiply loop. In our approach, we only use a weaker bound: $s < 35$ because $u \geq 3$ (see Section 4).

2.4 Previous correctly rounded implementations of pow

All previous correctly rounded implementations of pow, in Sun's `libmcr`, in Ziv's `libultim` and in MPFR, use a combination of Ziv's correct rounding technique and the general techniques for detecting rounding boundary cases. Their control flows are illustrated in Figure 3. The illustrations have been obtained by analysis of the source code because documentation is only partially available [17, 16, 7, 13].

The implementation in Sun's `libmcr` uses the most conservative approach to rounding boundary filtering (see Figure 3(a)). Before Ziv's iteration process is started, all rounding boundary



(a) Approach in `libmcr`

(b) Approach in `libultim`

(c) Approach in MPFR

Figure 3: Previous approaches

cases are filtered out. Arguments run through nested branches before possibly being tested in the repeated square root extraction loop. The exactness of each square root extraction is tested by clearing and checking the IEEE 754 inexact flag. If a case is determined to be a rounding boundary case, the result of pow is computed by a square and multiply loop. Midpoint cases get correctly rounded by the last multiplication [16]. If a case does not lie on a rounding boundary, Ziv's iteration process is launched without reuse of the intermediate results produced in the rounding boundary test.

Branches and, in particular, IEEE 754 flag access are expensive operations on current processors because of pipeline stalls. In the `libmcr` approach, some of the rounding boundary detection branches get executed independently whether the case is a rounding boundary case or not. Even if average, not-rounding-boundary cases do not run through every stage of the nested testing structure, the critical path gets delayed.

Ziv's approach in `libultim` accounts for higher average performance on the critical path. Rounding boundary detection is performed only after two iterations of Ziv's correctly rounding process (see Figure 3(b)). As probability arguments [17, 4, 6] show, average cases are thus returned faster. In consequence, the rounding boundary test is executed on average for fewer inputs. Its probabilistic relative cost decreases.

Although the approach in `libultim` allows for higher average performance than the approach in `libmcr`, it is still not optimal: the rounding boundary detection is delayed until after the second approximation step in Ziv's iteration. However, the approximation is used neither for faster rejection of not-rounding-boundary cases nor for faster computation of the value of rounding boundary cases. Although an approximation to x^y is available and could yield to $\diamond_{53}(x^y)$ by $\diamond_{53}(\circ_{54}(x^y))$ as explained above, Ziv's implementation recomputes rounding boundary cases x^y by repeated multiplication. This approach delays thus rounding boundary cases more than necessary.

The implementation of pow in MPFR modifies the approach of pow in `libultim` [7, 13]. Some special cases, i.e. $y \in \mathbb{Z}$, $x = 2^E$, $E \in \mathbb{Z}$ and $x = 2^E$, $y \cdot E \in \mathbb{Z}$, are filtered out before starting a Ziv iteration. Rounding boundary handling is simple for these special cases. For the remaining cases, an approach similar to the one in `libultim` is used: the first step of Ziv's iteration is executed. This allows for fast average performance for not-rounding-boundary cases. Rounding boundary test is then performed by repeated testing of perfect squares. MPFR relies here on a test implemented in the GNU Multi Precision Library (GMP)[2]. Once a rounding boundary case is detected, the value of x^y is computed by the integer power function, using a square and multiply process.

In the comparisons of the different approaches, the following important point must be accounted for. The libraries `libmcr` and `libultim` are targeted to double precision [17, 16]. The MPFR library supports multiple-precision computations [7, 13]. Algorithms for multiple-precision should have the best known asymptotic complexity in the average case. In contrast, algorithms for a given fixed precision, like double precision, may be optimized not in terms of asymptotic complexity but in terms of latency in cycles. In this article, we aim at speeding up an implementation of pow in double precision. The MPFR implementation cannot meaningfully be compared to the double precision implementations in every aspect but may be source of inspiration.

3 New approach for pow

3.1 Number of rounding boundary cases

In the continuation of the successive improvements in Ziv's `libultim` and MPFR, we want to increase average performance on both not-rounding-boundary cases and rounding-boundary cases. For average case analysis and improvement, information on the probabilities of the different types of inputs is necessary [17, 4]. The case counts given in this section have been obtained mainly on the base of the properties of rounding boundary cases presented in Section 2.3. The cases have been counted using ad-hoc, quick-and-dirty algorithms that are outside the scope of this article.

In double precision, the function pow : $(x, y) \mapsto x^y$, $x, y \in \mathbb{D}$ has about 2^{112} regular arguments. We call regular arguments inputs (x, y) for which x^y is a real number that can be rounded into double precision without under- or overflow or complete loss of precision. Typically, irregular arguments produce NaN or infinities on output or are rounded to 0 or 1. The count can be verified by considering all exponents of x and computing bounds on y for each exponent.

Let us now consider the number of rounding boundary cases separating them into exact and midpoint cases. Some rounding boundary cases are trivial. Typically, the cases $y = 1$ or $y = 2$ are handled by ad-hoc filtering: $x^1 = x$ and $x^2 = x \cdot x$. We do not consider these two trivial cases. The number of exact cases is slightly greater than the number of mid-point cases: there are roughly 2^{27} non-trivial exact cases and roughly 2^{25} midpoint cases. The conditions for an input (x, y) to be a rounding boundary case detailed in Section 2.3 are slightly weaker for exact than for midpoint cases. Round-to-nearest is the IEEE 754 default mode [1]. We therefore concentrate on midpoint cases, that are associated to that mode.

There are $37500822 \approx 2^{25.5}$ midpoint cases x^y, $y \neq 2$, in double precision. When additional excluding $y = 3$, the count drops to $19066760 \approx 2^{24.2}$ cases. Further excluding even $y = 4$, $18596893 \approx 2^{24.1}$ cases remain. Remark that still 18431732 of these cases are formed by the case $y = \frac{3}{2}$. In contrast, excluding $y = \frac{3}{2}$ is difficult, because it would currently imply usage of a

[2] available at `http://gmplib.org/`

square root extraction which is an expensive operation on current pipelined processors. Only 2330 midpoint cases x^y, $y \neq 2$, round to subnormal numbers $\circ_{53}(x^y)$. In all subnormal cases, x and y are normal numbers and y is different from $3, 4$ or $\frac{3}{2}$.

For uniformly distributed regular arguments (x, y), the probability for an argument to be a midpoint case with $y \neq 2$ is approximately $p_{\text{midpoint}} = \frac{2^{25}}{2^{112}} = 2^{-87}$. By probabilistic arguments [17, 4, 6], this can be related to the probability of a not-rounding-boundary, hard-to-round case. It would be a case for which an accuracy corresponding to $53+1+87 = 141$ bits is necessary in Ziv's iteration in order to ensure correct rounding. In Ziv's `libultim`, the rounding boundary detection seems thus to be executed still too early after the second approximation step giving about 80 significant bits [4, 17]. However, remark that this argument is based on a uniformity hypothesis on the inputs that might not be satisfied.

Executing the rounding boundary detection after a later Ziv's iteration step has negative impact on the performance on rounding boundary cases. One might think of an application using pow on a set of inputs that are all rounding boundary cases. Here the following observation can be made: On uniformly distributed arguments (x, y) that are non-trivial midpoint rounding boundary cases, the probability for y being $y = 3$ or $y = 4$ is $p_{\text{simple}} = \frac{37500822-18596893}{37500822} \approx 50.4\%$. We propose thus to filter not only $y = 2$ but also $y = 3$ and $y = 4$ before starting Ziv's iteration process. The detection of the remaining rounding boundary cases can then be performed after an iteration step with an accuracy of about 120 valid bits. The handling of arguments $y = 2$, $y = 3$ and $y = 4$ can be done essentially for free in current pipelined processors. The impact of late rounding boundary detection on rounding boundary case performance drops to the half. Section 5 gives the measured performance of this approach.

3.2 Using worst-case bounds for rounding boundary detection

The impact of rounding boundary detection on the whole performance of a correctly rounded implementation of pow can be decreased by simplifying the detection algorithm. Let us now see how the repeated square root extraction and square and multiply process can be avoided at all.

Suppose that worst-case information is available. Let us show how rounding boundary cases can be discerned from not-rounding-boundary cases merely using an approximation, that is needed anyway in Ziv's correct rounding iteration process. On all regular double precision inputs $(x, y) \in \mathbb{D}^2$, that are not rounding boundary cases, Ziv's iteration will be able to provide a correctly rounded result in any rounding mode after approximating $\hat{z} = x^y$ by $z = x^y \cdot (1 + \varepsilon)$ with a relative error ε not greater than some bound $\overline{\varepsilon}$, i.e. $|\varepsilon| \leq \overline{\varepsilon}$, where $\overline{\varepsilon}$ is the worst-case of the Table Maker's Dilemma [14, 10, 15]. Hence, whatever interval between two consecutive floating-point numbers is considered, not-rounding-boundary cases x^y may fall in some range between the floating-point numbers and their midpoint. However, they cannot be nearer to them than $x^y \cdot \overline{\varepsilon}$. Around floating-point numbers and their midpoints there are gaps in which no numbers x^y, $x, y \in \mathbb{D}$, can fall. See Figure 4 for an illustration. Let $\hat{z} = x^y$ now be a rounding boundary case, for example a midpoint case. Let z be an approximation to $\hat{z}$ with a relative error ε less than half the worst-case error $\overline{\varepsilon}$, i.e. $z = x^y \cdot (1 + \varepsilon)$, with $|\varepsilon| \leq \frac{1}{2} \cdot \overline{\varepsilon}$. The approximation z will then lie in the gap – where no not-rounding-boundary cases may lie because of the worst-case bound. More precisely, the approximation interval Z around an approximation z to a rounding boundary case will not intersect with any approximation interval corresponding to a not-rounding-boundary case.

Rounding boundary detection can thus be performed as follows: after approximating x^y with an accuracy slightly higher than the worst-case, the algorithm simply checks whether the approximation falls in the gap or not. This test is exactly the same as the test for checking whether an approximation can be correctly rounded or not in Ziv's iteration [17, 3, 4]. It must be performed anyway in an approach using Ziv's iteration. If worst-case information were available for the function pow on its whole double precision definition range, rounding boundary detection would be essentially for free.

It is currently unfeasible to compute worst-cases for pow in double precision [15, 10]. Nevertheless, our rounding boundary detection approach still works. We define a subset $\mathbb{S}$ of the double precision numbers $\mathbb{D}^2$. This subset reflects the bounds on rounding boundary cases presented in

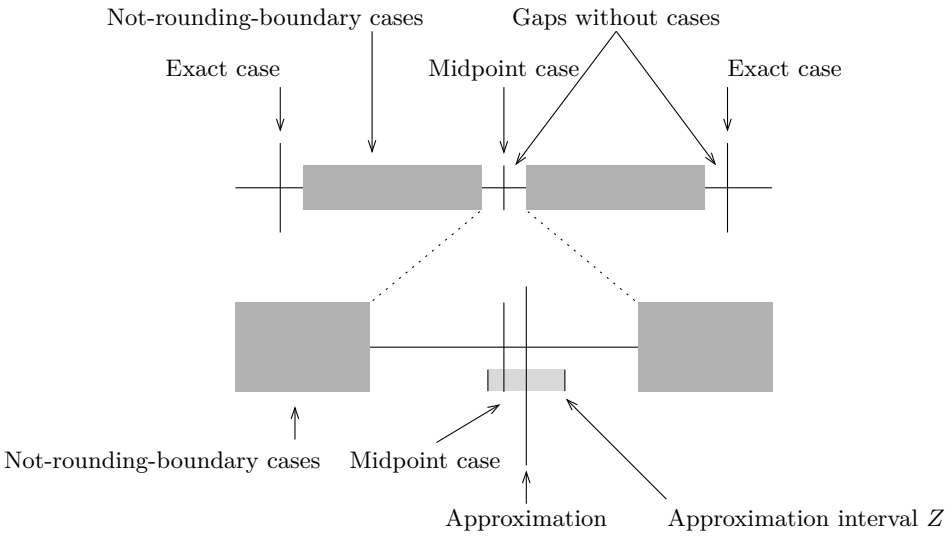


Figure 4: Using worst-case information for rounding boundary cases

Section 2.3.

$$\begin{aligned}\mathbb{S} &= \left\{(x,y)\in\mathbb{D}^2 \mid y\in\mathbb{N},\ 2\le y\le 35\right\}\\ &\cup \left\{(m,2^F n)\in\mathbb{D}^2 \mid F\in\mathbb{Z},\ -5\le F<0,\ n\in 2\mathbb{N}+1,\ 3\le n\le 35,\ m\in 2\mathbb{N}+1\right\}.\end{aligned}$$

As shown in Section 4, all rounding boundary cases of pow in double precision lie in $\mathbb{S}$. The rounding boundary test can thus refute rounding boundary cases immediately if some $(x,y)\in\mathbb{D}$ does not lie in $\mathbb{S}$. Further, it is possible to compute the worst-cases of x^y for inputs $(x,y)\in\mathbb{S}$ [10, 15]. For the first part of $\mathbb{S}$, where $y\in\mathbb{N}$, worst-case information for the integer powers of a double precision number can be reused [9]. The second part has about $2^{58.4}$ elements which has been a feasible number of inputs; Section 3.3 describes how the worst case has been found. For inputs in the subset, $(x,y)\in\mathbb{S}$, rounding boundary detection can thus be performed using worst-case information.

The following worst case for correct rounding to double precision has been found in the subset $\mathbb{S}$: for $x = 1988580363009869$, $y = 2^{-4}\cdot 5$, x^y reads in binary

$$x^y = 1110101111001110.0101001100001100010111001011000110001\ 1\ \underbrace{0\ldots0}_{60\text{ zeros}}111\ldots.$$

The worst-case accuracy $\overline{\varepsilon}$ is thus $\overline{\varepsilon} = \left|\frac{\circ_{54}(x^y)-x^y}{x^y}\right| \ge 2^{-114}$.

Let us see now how these worst cases have been computed.

3.3 Searching for the worst case of m^α

If we write $\alpha = 2^F n$, the problem of finding the worst case of the second part of $\mathbb{S}$ is reduced to searching for the worst case of m^α with $m\in\mathbb{D}\cap(2\mathbb{N}+1)$ for each of the 85 values of α.

The algorithms to search for the worst cases, described in [11, 15, 10], can be used only if the tested function can be approximated by small-degree polynomials on large enough intervals (with an error small enough to filter out most input numbers with these algorithms). However, if we consider $h_\alpha(m) = m^\alpha$ for the first values of m, the function h_α cannot be approximated by a small-degree polynomial. Fortunately, as m goes larger, the fast algorithms quickly start to be applicable and become more and more efficient, so that the whole search is possible in a reasonable time. Still, this makes the split into small intervals more complex, as shown below.

We sought to reuse existing code as much as possible, not only the implementation of the core algorithms [10], but the whole toolchain, which includes the split into small intervals, automatic error analysis (with guaranteed error bounds), generation of efficient code and the parallelization. So, we had to reduce the problem to the search of the worst cases of a univariate function with both input and output in some fixed precision.

The input number m takes 2^{52} values: 1, 3, 5, ..., $2^{53}-1$, so one can write $x = 1 + k \cdot 2^{-52}$, with $k = \frac{m-1}{2}$. As k takes the values 0, 1, 2, ..., $2^{52}-1$, the corresponding set of the values of x is $\mathbb{D} \cap [1, 2)$.

Since $x = 1 + (m-1)/2^{53}$, one has $m = 1 + (x-1) \cdot 2^{53}$, thus the tested function is $f_\alpha(x) = \left(1 + (x-1) \cdot 2^{53}\right)^\alpha$.

The input interval $[1, 2)$ was split in the following way (probably not optimal, but this choice was sufficient):

1. $[1 + 2^{-8}, 2)$ split into 8160 intervals of width 2^{-13};
2. $[1 + 2^{-15}, 1 + 2^{-8})$ split into 1016 intervals of width 2^{-18};
3. $[1 + 2^{-22}, 1 + 2^{-15})$ split into 254 intervals of width 2^{-23};
4. $[1 + 2^{-29}, 1 + 2^{-22})$ split into 254 intervals of width 2^{-30};
5. $[1 + 2^{-36}, 1 + 2^{-29})$ split into 254 intervals of width 2^{-37};
6. $[1 + 2^{-43}, 1 + 2^{-36})$ split into 254 intervals of width 2^{-44};
7. $[1 + 2^{-50}, 1 + 2^{-43})$ split into 254 intervals of width 2^{-51};
8. 4 values of $[1, 1 + 2^{-50})$.

The search ran for 25 days on a small network of machines. The SLZ algorithm [15] was not used because it is less interesting for the double precision, but also because the current implementation is not part of the mentioned toolchain.

3.4 Detection algorithm

Our rounding boundary algorithm **detectRoundingBoundaryCase** combines all previous elements for computing the predicate $RB(x, y) = (x^y \in \mathbb{F}_{54})$. It is illustrated in Algorithm 1.

The algorithm takes x, y and an approximation $z = x^y \cdot (1 + \varepsilon)$ in input. The accuracy ε of this approximation must be slightly better than the worst-case of the function pow in the subset $\mathbb{S}$. Typically, we choose $|\varepsilon| \leq 2^{-117}$. The algorithm starts with rounding z to the nearest floating-point number in $\mathbb{F}_{54}$, $2^G \cdot k = \circ_{54}(z)$. As explained (see Section 2.2), this rounding is not subject to the Table Maker's Dilemma if the rounding $\diamond_{53}(z)$ into $\mathbb{D}$ is. The algorithm performs then a rounding test, i.e. it checks whether z is near or on a rounding boundary. If the condition of this test, $\left|2^G \cdot k - z\right| \geq 2^{-116}$, is fulfilled, x^y is far from a rounding boundary. It does not fall in the gap around rounding boundaries. Hence, it cannot be a rounding boundary; the algorithm returns *false*. The rounding $\diamond_{53}(z)$ of the approximation z already yields the correctly rounded result $\diamond_{53}(x^y)$.

After this first test, the algorithm checks whether x is an integer power of 2, i.e. $x = 2^E$. In this case, x^y can be a rounding boundary case only if $E \cdot y \in \mathbb{Z}$. The algorithm responds appropriately. See Section 4 for the correctness proof in this particular case. If x is not an integer power of 2, the algorithm checks whether (x, y) lies in $\mathbb{S}$ or not. In the case where (x, y) lies in $\mathbb{S}$ but y is not integer, i.e. it decomposes into $y = 2^F \cdot n$, where $n \in 2\mathbb{N} + 1$ with negative F, the algorithm further checks whether not only $m^{2^F \cdot n} = k$ is satisfied but also $E \cdot y = G$, respectively whether $k \in 2\mathbb{Z} + 1$. This check is necessary in this case because the test $x^y = 2^G \cdot k$ must be decomposed into $E \cdot y = G$ and $m^{2^F \cdot n} = k$ (see Section 2.3). If one of the conditions, i.e. $(x, y) \in \mathbb{S}$ or $E \cdot y = G$, is not satisfied, the case cannot be a rounding boundary case by the gap argument. The algorithm

responds immediately *false* in this case. Otherwise the case must be a rounding boundary case; the algorithm returns *true*.

Input: $x \in \mathbb{D}, x > 0, y \in \mathbb{D}, y \neq 0, y \neq 1$
z such that $z = x^y \cdot (1 + \varepsilon)$ with $|\varepsilon| \leq 2^{-117}$.
Output: a predicate $RB(x, y) = (x^y \in \mathbb{F}_{54})$

```
1  Let 2^G · k = ∘_54(z) such that G ∈ ℤ, k ∈ ℕ;
2  if |2^G · k − z| ≥ 2^−116 · z then return false;
3  if ∃E ∈ ℤ, x = 2^E then
4  |  if E · y ∈ ℤ then return true else return false;
5  else
6  |  if y < 0 ∨ y > 35 then return false;
7  |  Let F ∈ ℤ, n ∈ 2ℕ + 1 such that 2^F · n = y;
8  |  if n > 35 ∨ F < −5 then return false;
9  |  if F < 0 then
10 |  |  Let E ∈ ℤ, m ∈ 2ℕ + 1 such that 2^E · m = x;
11 |  |  if E · y ∉ ℤ then return false;
12 |  |  if 2^(G−E·y) · k ∉ 2ℤ + 1 then return false;
13 |  end
14 |  return true;
15 end
```

Algorithm 1: **detectRoundingBoundaryCase**

Our rounding boundary detection algorithm decomposes its inputs x and y into $x = 2^E \cdot m$ and $y = 2^F \cdot n$. This operation seems to require an expensive loop for counting the trailing zeros in the significand of the numbers. In fact, previous approaches, for instance Sun's `libmcr` and Ziv's `libultim`, use such a loop. Actually, techniques are known[3] for performing such a decomposition using only some logical operations or a small table [12].

3.5 Correct rounding algorithm

Based on our analysis of the probability of rounding boundary cases presented in Section 3.1 and using our rounding boundary detection algorithm, we propose the approach for correctly rounding pow illustrated in Figure 5.

In our approach, the algorithm starts with filtering simple cases such as $y = 2$ for any x and $y = 3$ or $y = 3$ for x on not more than 21 bits. This filter can be performed while filtering irregular arguments of the function. On pipelined processors, the evaluation of x^y in the general case can even already be started. The cost of the filter is thus hidden inside the critical path. The results of the special cases x^2, x^3 and x^4 can be computed in an ad-hoc way. Since about half of the rounding boundary cases are these special cases, average performance on pure rounding boundary input will get speeded up.

The algorithm continues then with two iterations in Ziv's correct rounding loop. The first step will approximate x^y with an accuracy equivalent to about 60 valid bits [3, 4, 5]. In order to meet the requirements of our rounding boundary detection algorithm, the second step must then approximate x^y with an accuracy equivalent to at least 117 valid bits. The rounding boundary test then filters the remaining rounding boundary cases before further iterations are launched if needed. In our approach, the rounding boundary test with its mere eight comparisons becomes thus a negligible part of the whole algorithm for a correctly rounded function pow.

[3] see http://graphics.stanford.edu/~seander/bithacks.html

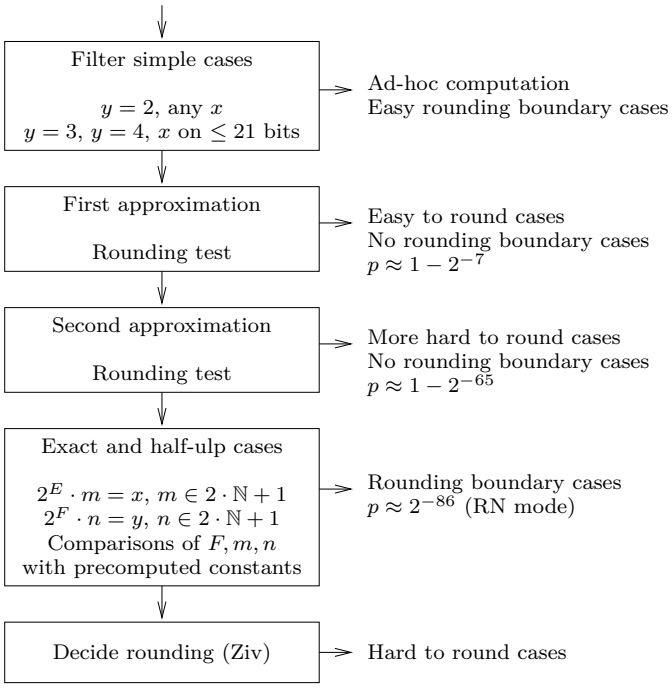


Figure 5: New approach to a correctly rounded pow

4 Correctness proof

Claiming correct rounding properties and – more important – claiming termination of Ziv's iteration is only worthwhile if a complete proof is given. Let us thus prove the correction of our rounding boundary detection Algorithm 1 **detectRoundingBoundaryCase**. We show a series of lemmas following the argumentation scheme in Section 2.3. This proof extends concepts sketched in [13].

Theorem 4.1
Let

$$\begin{aligned} \mathbb{S} &= \left\{(x,y) \in \mathbb{D}^2 \mid y \in \mathbb{N},\ 2 \leq y \leq 35\right\} \\ &\cup \left\{(m, 2^F n) \in \mathbb{D}^2 \mid F \in \mathbb{Z},\ -5 \leq F < 0,\ n \in 2\mathbb{N}+1,\ 3 \leq n \leq 35,\ m \in 2\mathbb{N}+1\right\}. \end{aligned}$$

Let $\circ_{54}$ *be the rounding-to-nearest to* 54 *bits. Thus, the following holds:*

$$\forall (x,y) \in \mathbb{S}, x^y \in \mathbb{F}_{54} \vee \left| \frac{\circ_{54}(x^y) - x^y}{x^y} \right| \geq 2^{-114}.$$

Proof (sketch):
The bound has been obtained using our worst-case search algorithm [10, 9]. Proving the correctness of this algorithm is behind the scope of this paper. ■

The test whether $2^{E \cdot 2^F \cdot n} \cdot m^{2^F \cdot n} = 2^G \cdot k$ can be decomposed into two separate tests:

Lemma 4.1
Assume that $E, F, G \in \mathbb{Z}$, $m, n, k \in 2\mathbb{N}+1$. *Thus, the following holds*

$$2^{E \cdot 2^F \cdot n} \cdot m^{2^F \cdot n} = 2^G \cdot k \quad \Leftrightarrow \quad \begin{array}{l} E \cdot 2^F \cdot n = G \\ \wedge \ \ m^{2^F \cdot n} = k. \end{array}$$

Proof (sketch):
Consider the fact that $m^{2^F \cdot n}$ and k or m^n and $k^{2^{-F}}$ are odd integers and $E \cdot 2^F \cdot n - G \neq 0$ or $E \cdot n - G \cdot 2^{-F} \neq 0$. Thus, the equations $2^{E \cdot 2^F \cdot n - G} \cdot m^{2^F \cdot n} = k$ or $2^{E \cdot n - G \cdot 2^{-F}} \cdot m^n = k^{2^{-F}}$ yield to contradictions because their left-hand sides are even integers and their right-hand sides are odd. ■

The algorithms check several bounding conditions on the inputs. These bounds can be shown as follows.

Lemma 4.2
Let $m \in 2\mathbb{N}+1$ be bounded by $3 \leq m \leq 2^{53}-1$. Let $n \in 2\mathbb{N}+1$ be bounded by $1 \leq n \leq 2^{53}-1$. Let $k \in 2\mathbb{N}+1$ be bounded by $1 \leq k \leq 2^{54}-1$. Let $F \in \mathbb{Z}$ be an integer. Assume that $m^{2^F \cdot n} = k$. Thus, $2^F \cdot n$ is bounded by $2^F \cdot n \leq 35$ and F is bounded by $-5 \leq F \leq 5$.

Proof:
In the first place let us show the upper bounds. Since $k \leq 2^{54}-1$, we know that $m^{2^F \cdot n} \leq 2^{54}$ and $2^F \cdot n \cdot \log_2(m) \leq 54$. Since $m \geq 3$, we have $\log_2(m) \geq \log_2(3) > 0$. Hence, $2^F \cdot n \leq \frac{54}{\log_2(m)} \leq \frac{54}{\log_2(3)} < 34.08 < 35$. This is the upper bound to be shown for $2^F \cdot n$. Since $n \geq 1$, we have $2^F \leq 35$ and therefore $F \leq 5.13$. Since F is integer, we have the given bound $F \leq 5$.

Let us show now that $F \geq -5$. Without loss of generality, we can suppose that F is negative. Let p_i, q_i be prime numbers such that $i \neq i' \Rightarrow p_i \neq p_{i'} \wedge q_i \neq q_{i'}$. Let $\alpha_i, \beta_i \in \mathbb{N}\backslash\{0\}$ be valuations such that $m = \prod_i p_i^{\alpha_i}$ and $k = \prod_i q_i^{\beta_i}$ Since $F \leq -1$, 2^{-F} is integer. Thus, we have $m^n = k^{2^{-F}}$ where m^n and $k^{2^{-F}}$ are integers. In consequence, the equation $\prod_i p_i^{\alpha_i \cdot n} = \prod_i q_i^{\beta_i \cdot 2^{-F}}$ holds and there exists a permutation σ such that $\forall i \,.\, p_i = q_{\sigma(i)} \wedge \alpha_i \cdot n = \beta_{\sigma(i)} \cdot 2^{-F}$. Since m is an odd integer and $m \geq 3$, $\forall i \,.\, p_i \geq 3$ holds. Further $m \leq 2^{53}-1$ and therefore $3^{\alpha_i} \leq 2^{53}$ and $\alpha_i \leq 53 \cdot \frac{\log(2)}{\log(3)} \leq 34$. Let $\kappa_i \in 2\mathbb{N}+1$ be odd integers and $\gamma_i \in \mathbb{N}$ valuations such that $\forall i \,.\, \alpha_i = 2^{\gamma_i} \cdot \kappa_i$. Such κ_i and γ_i exist for all α_i because $\alpha_i \in \mathbb{N}$. Since $\forall i \,.\, \alpha_i \geq 1$, the following upper bounds are satisfied: $\forall i \,.\, \kappa_i \geq 1$. As $\forall i \,.\, \alpha_i \leq 34$, the following holds $2^{\gamma_i} \leq 34$ and $\gamma_i \leq \log_2(34) \leq 5.09$. Since $\gamma_i \in \mathbb{N}$, $\gamma_i \leq 5$. The following identity has been shown to hold: $2^{\gamma_i} \cdot (\kappa_i \cdot n) = \beta_{\sigma(i)} \cdot 2^{-F}$ Since n and all κ_i are odd integers, $\kappa_i \cdot n$ is odd. Further, $\beta_{\sigma(i)}$ is an integer. Thus, $-F$ is bounded above by γ_i which is bounded by 5. So $F \geq -\gamma_i \geq -5$. ■

The test $m^{2^F \cdot n} = k$, $F < 0$, can be decomposed into two tests, $m^{2^F} = j \in \mathbb{N}$ and $j^n = k$:

Lemma 4.3
Assume $m, n, k \in 2\mathbb{N}+1$, $F \in \mathbb{Z}$, $F \leq -1$.
Thus, $m^{2^F \cdot n} = k \Leftrightarrow \exists j \in \mathbb{N} \,.\, \left(j = m^{2^F} \wedge j^n = k\right)$ holds.

Proof (sketch):
It suffices to remark that a 2^{-F}-root of an integer m is integer only if all valuations of the prime factor decomposition of m are divisible by 2^{-F} and that n is odd. So, if $j = m^{2^F}$ is not integer, there exists a valuation in the prime decomposition as well of m as of m^n that is not divisible by 2^{-F} but all valuations of $k^{2^{-F}}$ are divisible by 2^{-F}. ■

The correctness of our Algorithm 1 **detectRoundingBoundaryCase** is also to be shown for negative y that we can classify as follows.

Lemma 4.4
Assume that $x, y \in \mathbb{D}$ such that $x > 0$, $y < 0$, $x^y \in \mathbb{R}$ and $2^{-1075} \leq |x^y| \leq 2^{-1024}$.
Thus, $x^y \in \mathbb{F}_{54}$ iff $\exists a \in \mathbb{Z} \,.\, (2^a = x \wedge a \cdot y \in \mathbb{Z})$.

Proof:
The existence of the indicated a clearly implies $x^y \in \mathbb{F}_{54}$. The other implication can be as follows: Assume that $x^y \in \mathbb{F}_{54}$ but the contrary of the property to be implied. Since $x, y \in \mathbb{F}_{53}$ and $x^y \in \mathbb{F}_{54}$, there exist odd integers $m, n, k \in 2\mathbb{N}+1$ and signed integers $E, F, G \in \mathbb{Z}$ such that $x = 2^E \cdot m$, $y = -2^F \cdot n$ and $x^y = 2^G \cdot k$. This yields to $m^{2^F \cdot n} = 2^{-G - E \cdot 2^F \cdot n} \cdot \frac{1}{k}$. There are two case

depending on the sign of F. If $F \geq 0$ then 2^F is an integer, $2^F \cdot n$ is an integer and $-G - E \cdot 2^F \cdot n$ is a signed integer. There exists therefore an integer $a \in \mathbb{N}$, a signed integer $b \in \mathbb{Z}$ and an odd integer $c \in 2\mathbb{N}+1$ such that $m^a = 2^b \cdot \frac{1}{c}$ by the definitions $a = 2^F \cdot n$, $b = -G - E \cdot 2^F \cdot n$ and $c = k$. If $F < 0$ then 2^{-F} is integer, $-G \cdot 2^{-F}$, $E \cdot n$ and $-G \cdot 2^{-F} - E \cdot n$ are signed integers and $k^{2^{-F}}$ is an odd integer. So there exist an integer $a \in \mathbb{N}$, a signed integer $b \in \mathbb{Z}$ and an odd integer $c \in 2\mathbb{N}+1$ such that $m^a = 2^b \cdot \frac{1}{c}$ by $a = n$, $b = -G \cdot 2^{-F} - E \cdot n$ and $c = k^{2^{-F}}$.

If $b \geq 0$, 2^b is integer. Since m^a is integer, $\frac{2^b}{c}$ is integer. As c is odd, $\gcd\left(2^b, c\right) = 1$. In consequence, c is equal to 1, $c = 1$. If $b < 0$, 2^{-b} and $2^{-b} \cdot c$ are integer. Since m^a is integer, $\frac{1}{2^{-b} \cdot c}$ is integer. Hence, $2^{-b} \cdot c = 1$ and thus, $c = 1$ because c is odd and 2^{-b} is integer.

Since $c = 1$ and $c = k$ or $c = k^{2^{-F}}$, k is equal to 1 in all cases. This implies that $x^y = 2^G \cdot 1$ is an integer power of 2. In consequence, $x = 2^{\frac{G}{y}}$. Since 2^ξ is transcendental for all algebraic ξ that are not signed integers (see [8]), and since x is algebraic, there exists an $a = \frac{G}{y} \in \mathbb{Z}$. Thus, $a \cdot y = G \in \mathbb{Z}$ and $x = 2^{\frac{G}{y}} = 2^{\frac{a \cdot y}{y}} = 2^a$. This yields a contradiction with the hypotheses. ■

Here is finally the correctness theorem of our algorithm for detecting rounding boundary cases.

Theorem 4.2
Algorithm 1 ***detectRoundingBoundaryCase*** *is correct.*
This means $\forall x \in \mathbb{D}, x > 0$ *and* $\forall y \in \mathbb{D}, y \neq 0, y \neq 1$ *such that* $2^{-1075} \leq x^y \leq 2^{1024}$ *and* $\forall z = x^y \cdot (1+\varepsilon)$ *for some* ε, $|\varepsilon| \leq 2^{-116}$, *the algorithm returns* true *iff* $x^y \in \mathbb{F}_{54}$.

Proof (sketch):
Combining Theorem 4.1, Lemmas 4.1, 4.2, 4.3 and 4.4, and considering at which lines the algorithm may return *true* (respectively *false*), the consequence of the theorem follows. ■

5 Experimental results

We have implemented our approaches to correct rounding of the function pow and to rounding boundary detection inside the `crlibm` library [3]. We have compared our implementations to a not-correctly rounded system `libm`, to Ziv's `libultim`, Sun's `libmcr` and to MPFR version 2.2.0. The experiments have been conducted on two systems: first, on an Intel Xeon CPU at 2.40 GHz running GNU/Linux 2.6.19.2-server with `gcc` 3.3.5 and, second, on an IBM Power5 at 1.66 GHz running GNU/Linux 2.6.18.8-0.3-ppc64 with `gcc` 4.1.2. Note that some of the considered implementations are not available on some systems. The only rounding mode supported by all libraries is round-to-nearest. The timing measures have been normalized to 1 for `crlibm`.

We give separate timings for random input and for input consisting only of non-trivial rounding boundary cases ($y \neq 1$, $y \neq 2$). We indicate average and worst-case timings. The worst-case timings given for random input are the worst values observed and not absolute values.

	Intel Xeon		IBM Power5	
	Random avg./worst	$x^y \in \mathbb{F}_{54}$ avg./worst	Random input avg./worst	$x^y \in \mathbb{F}_{54}$ avg./worst
`crlibm`	1/7.70	3.18/6.38	1/7.63	4.06/8.42
`libm`	1.20/134	0.633/0.899	-	-
`libultim`	-	-	1.65/8550	3.19/4.14
`libmcr`	3.54/172	0.636/1.61	-	-
MPFR	170/298	47.9/168	700/1090	188/534

These results show that average performance on random input is increased by at least 39% in our implementation with respect to previous implementations, for instance `libultim`. These improvements are obtained at the sake of a slight slow-down of the very rare rounding boundary cases, for instance, of about 21% with regard to `libultim`. The difference between the timings for hard-to-round cases and rounding boundary cases can be neglected: rounding boundary cases

are not more than 9% slower. For `libmcr`, this overhead of rounding boundary detection could still go up to about 50%. It seems reasonable that an application that can afford a factor 7.70 between average and worst-cases for correct rounding, can afford 9% more for 39% speed-up on average. The differences on the Xeon architecture where the worst-case on random input is slower than rounding boundary cases, can be explained by expensive subnormal rounding. Rounding boundary cases are about 50% faster on average than in the worst-case. This latter experimental result perfectly validates the relevance of our theoretical estimates of 49.6% (see Section 3.1).

6 Conclusions and future work

In this article, we have considered the detection of rounding boundary cases of the function pow : $(x, y) \mapsto x^y$ in double precision. We have presented an algorithm for efficient rounding boundary detection. The algorithm, consisting of practically only a few comparisons with constants, allows for better average performance of an implementation of pow. Typically, the critical path gets no longer delayed by operations that are difficult to pipeline. Loops involving tests and square root extraction are replaced by a straight-line program.

Our algorithm uses pre-computed constants. These constants are derived from the worst-case accuracy of the function pow in a particular sub-domain of double precision. This innovative use of the techniques developed for the correct rounding can be extended to other functions and might be profitable in terms of performance. We will investigate in this direction in the future.

The advantages of our approach and implementation is the improvement of the average performance of pow of about 39% and the drop in overhead of the rounding boundary detection to maximally 9%. The drawback of the approach is the slow-down of the very rare ($p = 2^{-87}$) rounding boundary cases of about 21%. With respect to rounding boundary case average performance, this slow-down could be diminished if not only the trivial cases $y = 2$, $y = 3$ and $y = 4$ could be handled apart without affecting the critical path on pipelined processors. For instance, still 99.1% of the non-trivial midpoint cases are formed by the case $y = \frac{3}{2}$. It is part of our future work to find a special algorithm for this case. The case would require a pipeline-blocking square root extraction on a special path if current techniques were used.

The algorithm presented in this article is a basic brick for a correctly rounded implementation of pow. The worst-case accuracy for correct rounding and – more important – the absolute worst-case timing of the different implementations still cannot be bounded. For the purpose of this article, worst-case information on a partial domain has been useful. Future work might evaluate how correct rounding of pow with bounded worst-case performance could be enabled by only partial information on hard-to-round cases.

References

[1] ANSI/IEEE. Standard 754-1985 for binary floating-point arithmetic, 1985.

[2] S. Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, November 2004.

[3] CRLibm, a library of correctly rounded elementary functions in double-precision. http://lipforge.ens-lyon.fr/www/crlibm/.

[4] F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th IEEE Symposium on Computer Arithmetic*, Cape Cod, Massachussets, June 2005.

[5] F. de Dinechin, Ch. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *RAIRO, Theoretical Informatics and Applications*, 41:85–102, 2007.

[6] A. Feldstein and R. Goodman. Convergence estimates for the distribution of trailing digits. *Journal of the ACM*, 23(2):287–297, April 1976.

[7] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), June 2007.

[8] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[9] P. Kornerup, V. Lefèvre, and J.-M. Muller. Computing integer powers in floating-point arithmetic. Research report RR2007-23, Laboratoire de l'Informatique du Parallélisme, Lyon, France, May 2007.

[10] V. Lefèvre. New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding. In P. Montuschi and E. Schwarz, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 68–75, Cape Cod, MA, USA, June 2005. IEEE Computer Society Press, Los Alamitos, CA.

[11] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 111–118, Vail, Colorado, 2001. IEEE Computer Society Press, Los Alamitos, CA.

[12] Ch. E. Leiserson, H. Prokop, and K. H. Randall. Using de Bruijn sequences to index a 1 in a computer word. `ftp://theory.lcs.mit.edu/pub/cilk/debruijn.ps.gz`.

[13] The MPFR Team. The MPFR library: Algorithms and proofs. `algorithms.tex` File - revision 4629 (July 4, 2007) - available in SVN repository from `http://www.mpfr.org/gforge.html`.

[14] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[15] D. Stehlé, V. Lefèvre, and P. Zimmermann. Searching worst cases of a one-variable function using lattice reduction. *IEEE Transactions on Computers*, 54(3):340–346, March 2005.

[16] Sun Microsystems. libmcr, a reference correctly-rounded library of basic double-precision transcendental elementary functions.
`http://www.sun.com/download/products.xml?id=41797765`.

[17] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.