Towards the post-ultimate `libm`

Florent de Dinechin
LIP, ÉNS de Lyon
46 allée d'Italie
69364 Lyon cedex 07, France
Florent.de.Dinechin@ens-lyon.fr

Alexey V. Ershov
Intel Corporation
Alexey.Ershov@intel.com

Nicolas Gast
École Normale Supérieure
45, rue d'Ulm
75230 Paris cedex 05, France
Nicolas.Gast@ens.fr

Abstract

This article presents advances on the subject of correctly rounded elementary functions since the publication of the `libultim` mathematical library developed by Ziv at IBM. This library showed that the average performance and memory overhead of correct rounding could be made negligible. However, the worst-case overhead was still a factor 1000 or more. It is shown here that, with current processor technology, this worst-case overhead can be kept within a factor of 2 to 10 of current best `libm`s. This low overhead has very positive consequences on the techniques for implementing and proving correctly rounded functions, which are also studied. These results lift the last technical obstacles to a generalisation of (at least some) correctly rounded double precision elementary functions.

1 Introduction

1.1 Correct rounding and elementary functions

The IEEE-754 standard for floating-point arithmetic [3] defines the usual floating-point formats (single and double precision) and specifies precisely the behaviour of the basic operators $+, -, \times, \div$ and $\sqrt{\ }$. The standard defines four rounding modes (to the nearest, towards $+\infty$, towards $-\infty$ and towards 0) and demands that these operators return the *correctly rounded* result according to the selected rounding mode: the result should be as if the calculation had been performed in infinite precision, and then rounded.

The adoption and widespread use of the IEEE-754 standard have increased the numerical quality of, and confidence in floating-point code. In particular, it has improved *portability* of such code and allowed construction of *proofs* of numerical behaviour. Directed rounding modes (towards $+\infty$, $-\infty$ and 0) are also the key to enable efficient *interval arithmetic* [20, 13].

However, the IEEE-754 standard specifies nothing for elementary functions, which limits these advances to code excluding such functions. Indeed, the mathematical libraries (`libm`) provided by operating systems do not guarantee correct rounding. Older `libm`s give no guarantee at all, but all the recent ones return a result with an error smaller than one unit in the last place *(ulp)* and with a high probability of correct rounding. We will refer to such libraries as *accurate-faithful*. Such functions are usually computed using fast table-based methods [10, 11, 23], see the books by Muller [22] or Markstein [18] for recent surveys on the subject.

1.2 The Table Maker's Dilemma

The main reason why no standard imposes correct rounding of elementary functions is the following. In most cases, the image $\hat{y}$ of a floating-point number x by an elementary function f is not a floating point number, and can therefore not be represented exactly in standard numeration systems. The correctly rounded result will be the floating-point number that is closest to this mathematical value (or immediately above or immediately below, depending on the rounding mode). A computer will evaluate an approximation y to the real number $\hat{y}$ with precision $\overline{\varepsilon}$, meaning that the real value $\hat{y}$ belongs to the interval $[y(1-\overline{\varepsilon}), y(1+\overline{\varepsilon})]$. The dilemma (named in reference to the early builders of logarithm tables) occurs when this information is not enough to decide correct rounding. For instance, if $[y(1-\overline{\varepsilon}), y(1+\overline{\varepsilon})]$ contains the middle of two consecutive floating-point numbers, it is impossible to decide which of these two numbers is the correctly rounded to the nearest of $\hat{y}$. In such cases, current `libm`s return one of the two surrounding numbers. From a numerical point of view, both are almost equally good – or equally bad – approximations to $\hat{y}$, since $\hat{y}$ is very close to their middle. However it means that for such arguments, the results returned by two different `libm`s may differ by one ulp.

A technique for computing the correctly rounded value,

Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)
1063-6889/05 $20.00 © 2005 **IEEE**

published by Ziv [24] and implemented in the pioneering IBM Accurate Portable Mathlib [17] (or `libultim`), is to improve the precision $\overline{\varepsilon}$ of the approximation until the correctly rounded value can be decided. Given a function f and an argument x, a first, quick approximation y_1 to the value of $f(x)$ is evaluated, with accuracy $\overline{\varepsilon}_1$. Knowing $\overline{\varepsilon}_1$, it is possible to decide if it is possible to round y_1 correctly, or if more precision is required, in which case the computation is restarted using a slower approximation of precision $\overline{\varepsilon}_2$ better than $\overline{\varepsilon}_1$, and so on. This approach leads to good average performance, as the slower steps are rarely taken.

1.3 Improving on Ziv's approach

However there was until recently no practical bound on the termination time of Ziv's iteration. It may be proven to terminate for most transcendental functions, but the actual maximal precision required in the worst case is unknown. According to a statistical arguments by Gal [11, 22], and assuming the implementation can be proven correct (which is a huge problem in itself), the `libultim` approach provides correct rounding with probability higher than $1 - 2^{-500}$, which we call *astronomical confidence* in the following. Note however that there was no attempt to prove the correct rounding property for this implementation.

The need for arbitrary multiple precision has also a cost in terms of performance: In `libultim`, the measured worst-case execution time is indeed three orders of magnitude higher than that of accurate-faithful `libm`s. This might prevent using this method in critical application. A related problem is memory requirement, which is, for the same reason, unbounded in theory, and much higher than usual `libm`s in practice.

Finally, this library still lacks the directed rounding modes, which might be the most useful. Indeed, correct rounding provides a precision improvement over an accurate-faithful `libm` of only a fraction of an ulp in round-to-nearest mode. This may be felt of little practical significance. However, the three other rounding modes are needed to guarantee intervals in interval arithmetic. Without correct rounding in these directed rounding modes, interval arithmetic may loose up to one ulp of precision.

The goal of the `crlibm` project (at `http://lipforge.ens-lyon.fr/projects/crlibm/`) is therefore to design a mathematical library which is

- portable to any system implementing the ISO-C99 and IEEE-754 standards,
- correctly rounded in the four rounding modes,
- proven, both theoretically and in the implementation,
- and reasonably efficient in terms of performance (both average and worst-case) and resource usage.

The longer-term goal of this research is to enable the standardisation of correct rounding[1] for elementary functions [8].

1.4 Contributions of this article

This article presents recent advances towards this goal, supported by experimental results.

Section 2 recalls a range of techniques used for correctly rounding an elementary function in the portable `crlibm` library. Section 3 then relaxes the condition of *portability* to study the impact of specific processor features such as double-extended precision and hardware fused multiply-and-add. This raises practical questions, which Section 4 tries to answer by implementing two functions ($\arctan$ and $\exp$) on two processor families which support double-extended precision (Pentium and Itanium). These implementations have worst-case execution times respectively less than $3\times$ and $8\times$ the time of the best available accurate-faithful implementation (an improvement over `libultim`'s $1000\times$), all other things (average time, code size and memory consumption) being similar or improved. This has an important impact on the *design cost* of writing a proven, correctly rounded implementation, which is discussed in Section 5 along with other implementation considerations.

2 The `crlibm` approach

Ziv's Ultimate Mathematical Library is entirely based on IEEE-754-compliant double-precision FP arithmetic: The first few steps compute an approximation to the function as a the sum of two double-precision number (referred to as a *double-double* in the following). Subsequent, more accurate steps use a FP-based multiple-precision package which may provide up to 800 bits of precision, hence the astronomical confidence.

2.1 Tight worst cases for correct rounding

A first practical improvement over Ziv's approach derives from the availability of tight bounds on the worst-case accuracy required to compute many elementary functions, computed by Lefèvre and Muller [15] using ad-hoc algorithms. Some functions are completely covered (most notably exponential and logarithm in radix e, 2 and 10, the hyperbolic sine and cosine, and their inverses), some are still

[1] Note that IBM is no longer supporting the `libultim` project, but Sun Microsystems has recently released its own correctly-rounded library, called `libmcr`, which also addresses the main weaknesses of `libultim` but is currently much slower. Therefore there is in 2005 a choice of three correctly rounded libraries for double-precision, plus the correctly-rounded multiple-precision package MPFR [21]

being processed and should be covered within a few years. However, some functions (most notably the trigonometric functions and some special functions) are out of reach of current methods, although the domain for which current algorithms work is the most practically useful. For instance, the near-term goal for the trigonometric functions is to compute their worst cases for a few 2π periods around zero.

Knowing the worst case required accuracy for a function, it is possible to tailor Ziv's approach to match it: `crlibm` implements only two steps of Ziv's algorithm, the second one being accurate enough to cover the worst case required accuracy. This is not only more efficient, it also makes it much easier to *prove* that an implementation actually returns the correctly rounded result. The `crlibm` distribution includes a detailed description of each implementation, including an attempt at such a proof. This proof mostly consists in computing a tight error bound on the overall error of the first step, as explained below.

2.2 Portability

The mainstream `crlibm` implementation intends to be portable to most systems, assuming only C99 compliance and the availability of IEEE-754-compliant format and operations. Therefore its first step computes a result as a double double-precision number $y_h + y_l$. Its second step uses an ad-hoc, integer-based multiple-precision library called `scslib` [7].

2.3 Rounding tests and precision/performance tradeoffs

The approximation $y_h + y_l$ computed in the first step is used to decide if the second step needs to be launched. For rounding to the nearest, the following test (also present in `libultim`) is used:

```
if(y_h == (y_h + (y_l*e)) ) return y_h;
else /* launch accurate phase */
```

Here we use the C syntax ($==$ is the equality comparison operator), and e is a double-precision number computed out of the overall relative error $\overline{\varepsilon}$ of the first step as $e \approx 1+2^{54}\overline{\varepsilon}$. The exact value of e, the validity conditions of this test, and the proof that it ensures correct rounding, are detailed in the documentation of `crlibm` [1]. Similar tests are also given for directed rounding modes, they are conceptually simpler, therefore we concentrate in the sequel on round to the nearest.

The rounding test here depends on a constant e which is computed out of the overall relative error bound. This gives an hint at the performance tradeoff one has to manage when designing a correctly-rounded function: The average evaluation time will be

$$T_{\text{avg}} = T_1 + p_2 T_2 \qquad (1)$$

where T_1 and T_2 are the execution time of the first and second phase respectively (with $T_2 \approx 100T_1$ in `crlibm`), and p_2 is the probability of launching the second phase (typically we aim at $p_2 \approx 1/1000$ so that the average cost of the second step is less than $1/10$.

The value of e in the test implies that p_2 is almost proportional to $\overline{\varepsilon}$. Therefore, to minimise the average time, we have to

- balance T_1 and p_2: This is a performance/precision tradeoff (the faster the first step, the less accurate)
- and compute a tight bound on the overall error $\overline{\varepsilon}$.

Computing this tight bound is the most time-consuming part in the design of a correctly-rounded elementary function. The proof of the correct rounding property only needs a proven bound, but a loose bound will mean a larger p_2 than strictly required, which directly impacts average performance. Compare T_{avg} with $p_2 = 1/1000$ or $p_2 = 1/500$ for $T_2 = 100T_1$, for instance. As a consequence, when there are multiple computation paths in the algorithm, it makes sense to have a different rounding constant e on these different paths [6].

3 Beyond `crlibm`

3.1 Modern Floating-Point Units

Most recent processors offer specific hardware features which cannot yet be used in a portable way. For our purpose, the most significant of these features are:

- Double-extended precision with 64 bits of mantissa instead of 53 in double-precision, as specified in the IA-32 and IA-64 instruction sets (implemented by the Pentium-compatible and Itanium processors respectively). Note that the IEEE-754 standard gives a more general definition of double-extended precision, but it has not yet been translated as a usable, standard combination of processor/compiler/system. Therefore, in the sequel, the meaning of the phrase "double extended" will be that of the IA-32 specification (which is also included in the IA-64 specification).
- Fused floating-point multiply-and-add operators (FMA), as available in the Power/PowerPC and Itanium architectures. These operators improve performance (as they combine two operations in one instruction) but also accuracy, as only one rounding is performed. Most significant to us is that an FMA reduces the cost of Dekker algorithm (which computes the exact product of two FP numbers as the sum of two FP numbers [9, 14]) from 17 operations down to only 2, with a corresponding reduction of the cost of the double-double multiplication.

- A low overhead of changing the rounding mode or working precision, thanks to the availability of several floating-point status registers (FPSR) selected on an instruction basis. This is a feature of the Itanium processor family. Comparatively, other processors have only one FPSR, and changing it (e.g. to change the working precision) typically requires flushing the FP pipelines.

These features are being used for the standard evaluation of elementary functions [16], and some were actually designed for this purpose [19]. We now discuss their impact on the evaluation of a double-precision correctly rounded elementary function.

3.2 Correct rounding using double-extended arithmetic

This section summarises a recent study of the impact of using double-extended arithmetic for computing functions correctly rounded to double-precision [4].

3.2.1 First step using double-extended

A first obvious idea is to compute the first step in double-extended precision, which removes the need for double-double arithmetic in this step. This yields some performance improvement, typically up to 50%.

However, on architectures implementing the IA-32 instruction set, this approach requires changing the rounding mode of the processor, at least when entering the function (to convert the input x to a double-extended) and when leaving it (to return a double). This takes more than 20 cycles on the Pentium-4 processor in our experiments, and takes back a lot of the interest of using double-extended precision. On the Itanium processors, however, there is no such penalty.

3.2.2 Second step in double-double-extended

The performance that double-extended precision can bring to the second step is more dramatic. This format provides 64 bits of mantissa, so that the sum of two double-extended numbers (a *double-double-extended*) will hold 128 bits of precision. Unfortunately, Muller and Lefèvre found that for many functions (including $\exp$, $\cos$ and $\tan$), correct rounding sometimes required an intermediate accuracy higher than 2^{-130} (up to 2^{-157} for the exponential): It is therefore not possible to compute an intermediate result to such relative accuracy as the sum of two double-extended numbers. As this concerns relatively few values, the solution previously considered was to tabulate these values and the expected output. A more careful study, however, shows that it will not be necessary.

The central remark in [4] is that such bad cases always happen for very small values of the input number x. In such cases, a Taylor approximation provides a straightforward method for approximating the function as the sum of *three* double-extended numbers[2]: More specifically, an approximation of the function f as $1+p(x)$ or $x+p(x)$, where $p(x)$ is computed as a double-double-extended $p_h + p_l$, will hold the required relative accuracy [4]. Table 1 illustrates this situation for the two functions studied in this paper.

Exploiting the fact that the most significant term of this sum (1 or x) is representable as a double-precision number, it is then possible to recover the correct rounding of $1+p_h+p_l$ (or $x+p_h+p_l$) to double precision, using a sequence of 5 double-extended additions [4].

Function	Interval of x	WCA on f	WCA on p
e^x	$[2^{-54}, 2^{-44}]$	2^{-158}	2^{-115}
	$[2^{-44}, 2^{-30}]$	2^{-138}	2^{-109}
	$\lvert x\rvert \geq 2^{-30}$	2^{-113}	
$\arctan(x)$	$[2^{-25}, 2^{-18}]$	2^{-126}	2^{-109}
	$[2^{-18}, 2]$	2^{-113}	

Table 1. Worst-case accuracy (WCA) required for double-precision correct rounding of exponential and arctangent.

3.2.3 Practical questions

This work left open a few practical questions relative to the implementation of a correctly rounded functions [4].

1. What is the relative performance of the second step and the first step? What is the cost of the rounding test?

2. What implication does this have on the precision/performance tradeoff ?

3. Can we reuse intermediate results from the first step in the second one? Can we design algorithms sharing tables and intermediate values, which are efficient both for the first step and the second step ?

The experiments described in the next section were carried out to study these questions. The last section will draw more conclusions.

[2] This is not a coincidence: These worst cases are indeed highly improbable according to Gal's statistical argument [11, 22]. This argument predicts that the worst case accuracy for a double-precision correctly rounded function is expected at $2^{-53-64} = 2^{-117}$, and that a worst case accuracy of 2^{-157} has probability 2^{-40} of happening. The fact that such worse-than-expected cases indeed happen is a direct consequence of the availability of a Taylor approximation of the function, which breaks the assumptions of randomness in Gal's reasoning.

4 Experiments and results

For these experiments, we chose the exponential function because it is the easiest to implement, and the most often cited in the literature about elementary function implementation. Conversely, we chose the arctangent as being comparably expensive to implement: It is approximated by polynomials of comparatively larger degree, and its argument reduction requires either very large tables, or a division. Some other functions (trigonometric and special) present specific difficulties but they are left out of this study because their worst-case required accuracy is unknown so far.

In all the tests, input random numbers were chosen in a range which avoids the (less meaningful) special cases. We compare our results to vendor `libm`s which are highly optimised and accurate-faithful.

4.1 Rounding test

The rounding test presented in 2.3 assumes double-precision arithmetic: It has to be adapted if the first step now returns a double-extended number y_{de}. The straightforward idea is to build $y_h = \mathrm{RoundToDouble}(y_{de})$ and $y_l = y_{de} - y_h$ (which will be an exact operation), then use the test of 2.3 on y_h and y_l[3]

However, on both architectures, a test using only integer arithmetic turns out to be more efficient (about 15 cycles faster on Pentium architectures, and 3 on the Itanium 2). This test first casts the mantissa of y_{de} to a 64-bit integer, then considers the bits after the 53rd: A y_{de} difficult to round to the nearest is of the form: $\overbrace{1.\underbrace{xxx...xx}_{53\ bits}011...11}^{m\ bits} xxx...$ or: $\overbrace{1.\underbrace{xxx...xx}_{53\ bits}100...00}^{m\ bits} xxx...$, where m is deduced from the bound $\overline{\varepsilon}$ on the overall relative error. Therefore the test resumes to integer masks, shifts and logical operations testing these two cases. The same holds for directed rounding modes.

4.2 Arctangent on the Pentium processor

Here we chose an algorithm which allows both steps to share special case handling, range reduction, and some intermediate computations. This algorithm is described in [5].

[3]When targeting the Itanium processors, the cost is altogether 4 operations and about 16 cycles, thanks to the fact that the precision and rounding mode are controlled on an operation basis, with no overhead. When targeting IA-32 processors, this option could involve several costly changes of the precision (to double to compute y_h, then back to double-extended to compute y_l, then back to double to compute the test, then possibly back to double-extended for the second step). Fortunately the first conversion of y_{de} to the nearest double may be performed by the memory unit, without changing the precision of the FPU.

Note that the `libultim` implementation uses a table of about 40KB to perform an argument reduction without division. Our versions, in contrast, use less than 4KB of tables, at the expense of a division.

Table 2 show some absolute timings, in cycles. All these timings were measured under Linux, using the gcc-3.3 compiler[4]. They include the cost of a function call, which is about 25 cycles. In other words, to get the time actually spent computing the function, one should subtract 25 to these numbers. It should also be noted that the cost of changing the floating-point status register twice is about 40 cycles.

arctan Xeon	avg time	max time
GNU MPFR	438742	3955724
IBM's `libultim`	343	228904
`crlibm` (portable)	662	29076
`crlibm` (using DE)	350	1680
`crlibm` (DE, first step alone)	*306*	*564*
default `libm`	*339*	*388*

Table 2. Arctangent timings in cycles on a Pentium Xeon. The implementations in italic are not correctly rounded.

The worst case time here is less than 6 times the average time of the best current accurate-faithful. By returning a value before the rounding test, we can measure the incompressible time cost of correct rounding – see line "*`crlibm` (DE, first step alone)*". Here it is 350-306=44 cycles, or, between 10 and 20% of the best current accurate-faithful implementation.

The main conclusion of this experiment is that sharing tables and values between the first and second step doesn't incur a major performance penalty.

4.3 Arctangent on the Itanium

This first Itanium experiment uses the same algorithm as previously (and exactly the same 4KB tables), but tries to perform additional optimisations by using FMAs wherever possible, especially to speed up double-double arithmetic (In the first step, we also replaced the library double-extended division with a less accurate one to save a few cycles, and used a Estrin parallel implementation of the polynomial recurrence [22]). Such low-level optimisations

[4]The source code for these experiments (and more) is available in the `crlibm` CVS repository from `http://lipforge.ens-lyon.fr/projects/crlibm/`, files `arctan-pentium.c`, `arctan-itanium.c` and `exp-itanium.c`.

cannot be done efficiently using current gcc (3.3), because inserting assembly instructions in C code leads to poor scheduling. Therefore we used the Intel icc8.1 compiler for Linux to compile the arctangent. This compiler supports a range of intrinsics giving a high-level access to most assembly-language instructions, including FMAs. Table 3 shows some absolute timings, in cycles. These numbers include the cost of a function call, which is about 20 cycles.

arctan Itanium-2	avg time	max time
GNU MPFR	243,460	1,999,472
IBM's `libultim`	195	139,091
`crlibm` (portable)	441	44700
`crlibm` (using DE)	103	537
`crlibm` (DE, first step alone)	*94*	*130*
default `libm`	*83*	*85*

Table 3. Arctangent timings in cycles on an Itanium-2 processor. The implementations in italic are not correctly rounded.

The improvement over crlibm-portable is more dramatic than for the Pentium, because of the FMA, and because using double-extended doesn't incur the cost of flushing the pipeline on the Itanium as already explained.

Here we measured that the incompressible cost of the correct rounding test is about 9 cycles. It is therefore the main contribution to the average overhead of correct rounding. Again it is about 15% of the best current accurate-faithful implementation.

The worst case time here is again 6 times the average time of the best current accurate-faithful. This is somehow disappointing, because the FMA, speeding up double-double multiplication by a factor 8, should bring a proportionally larger improvement to the second step than to the first step. We believe there is room for improvement here. Note for instance that Hewlett Packard's Markstein [19] described a quad-precision arctangent in HP-UX accurate to 0.5001 ulp (which should be enough to derive a second step) within 321 cycles [19]. It uses a sequence of only 5 FMAs for computing one double-double Horner steps, where our code needs 15 FMAs.

4.4 Exponential on the Itanium

In this experiment, a second step was first derived from an Intel quad-precision routine, and then a first step was derived from the second step, using the same range reduction. An overview of the algorithm used is given in [5]. Note that the Intel oroginal code uses optimised double-double-extended FMAs similar to those mentionned in the previous paragraph. Also note that this is a relatively table-hungry algorithm (8KB), whereas the standard `libm` uses less than 1KB.

We timed the first step alone (to check it matches the performance of the standard `libm`), the second step alone (since it makes a self-sufficient correctly-rounded exp), and the two-step algorithm. We also timed the Linux standard `libm` (derived from Intel open-source optimised `libm`), and the portable version in `crlibm`. Table 4 shows these timings.

exp Itanium-2	avg time	max time
IBM's `libultim`	136	1,520,277
GNU MPFR	17,603	43,352
`crlibm` (portable)	298	4,601
`crlibm` (using DE, two steps)	67	114
`crlibm` (DE, second step alone)	92	92
`crlibm` (DE, first step alone)	*61*	*70*
default `libm`	*63*	*63*

Table 4. Exp timings in cycles on an Itanium-2 processor. The implementations in italic are not correctly rounded.

Here the cost of the function call is about one third the cost of evaluating the function itself. This raises a new question: Will a two-step correctly rounded function be inlined as efficiently as a (more straightline) accurate-faithful function? However studying this question is out of the scope of this paper.

Adding the first step to the code of the second step meant adding 13 lines only, since we reuse the special cases handling and the range reduction. These 13 lines take 114-92 = 22 cycles for a polynomial evaluation, a reconstruction, and the rounding test which is statically predicted not to go for the second step (and does in this case).

The question raised here is wether it is worth having a two-step algorithm. From another point of view, it might be more useful to write a two-step algorithm which has a slower second step and uses a smaller table (a 4KB exp could be written with a less-than-10x worst case).

5 Designing the post-ultimate libm

The immediate conclusion of the previous section is that using double-extended arithmetic, it is possible to design a double-precision correctly-rounded function whose worst case is within 10x of the best accurate-faithful, and whose

average performance is only degraded by the incompressible cost of the rounding test.

We now discuss the effort involved in designing such a function.

5.1 Reuse and share

The first idea is to reuse existing, well optimised algorithms. This is obvious for the first step, which can be derived from a standard `libm` implementation as soon as this implementation is faithful. Recent vendor libraries now return the correct result for more than 95% of the inputs [12, 16]. Deriving a first step from such a function involves

- adding a rounding test, which is easy, and
- proving a tight bound on the overall error, which may be difficult (of course, the more clever and sophisticated the algorithm, the more difficult the proof).

Recently, Andrey Naraikin and Alexey Ershov (from Intel Nizhniy Novgorod Lab) suggested that the second step could be similarly derived from a quad-precision implementation (this led to the Itanium exponential experiment above). Here, quad-precision means a 128-bit format with 112 bits of mantissa. This mantissa size wouldn't be enough to hold the precision required for correct rounding (usually more than 117 bits), however the functions are actually computed with a higher relative accuracy (typically using double-double-extended arithmetic) to keep the error very close to an half-ulp [19]. If this intermediate accuracy is higher than the worst-case required accuracy, such a *quad accurate-faithful* implementation can be easily retargeted as a correctly-rounded double implementation by changing only the last few operations.

As a conclusion, should a vendor commit itself to correctly-rounded double-precision functions, a lot of the work would be shared between double-precision and first step (at least the handling of exceptional cases), and between quad-precision and second step. Again, the real design cost would be in proving error bounds systematically.

5.2 Making proofs easy

Now we discuss the design cost of computing the error bounds. For the second step, it makes sense to have a large overkill of accuracy in the algorithm if it makes the proof simpler (typically having a coarse estimate on each rounding error): The average performance impact will be negligible, and the worst-case impact remains acceptable. In `crlibm` for instance, our ad-hoc multiple-precision library is much too accurate (to 200 bits) because it gives much freedom in designing the algorithms and proving them. More specifically, it allows to concentrate on approximation errors, because coarse bounds on rounding errors will suffice. This is still the case for a double-double-extended second step: The actual worst cases accuracies required are about 2^{-117}, and rounding to double-double-extended entails error smaller than 2^{-128}. Even taking into account that these rounding errors will accumulate, a coarse majoration will be adequate.

Now for the first step, we have to minimise p_2 and this means computing a tight bound on the error. However, an important consequence of the 10x factor and of a double-extended first step is that we may now be much lazier in this computation, and for the same reason. The second step will now be within a factor 10 of the first step, so $T_2 < 10T_1$. A coarse computation of the rounding errors in the double-extended first step will typically sum up to a term smaller than $\overline{\varepsilon} = 2^{-63}$, which would translate to $p_2 \approx 1/1000$. The contribution of this lazy error bound to the average time is therefore about $T_1/100$, which is negligible. Therefore, here also, we may concentrate on the approximation errors, which are simpler to manage.

We therefore find out that the cost of implementing a correctly rounded function using double-extended arithmetic is much reduced when compared to the cost in portable crlibm, where we had to compute a tight error bound on the first step (because being lazy had an impact on performance), and write the second step using `scslib` and its proof, sharing only little of this work with the first step.

As a final remark, it may even happen that a correctly rounded implementation provides faster average performance than an accurate-faithful implementation. The idea here is to compensate the cost of the rounding test by a faster first step, deliberately less accurate than the accurate-faithful version, because misrounds will be caught up and corrected by the second step anyway.

6 Conclusions

It is known since Ziv's work that it is possible to write elementary functions which are correctly rounded with astronomical probability, with a very small average performance overhead over the current best implementation.

This paper shows, with experimental support, that double-extended arithmetic allows to write functions which are proven correctly rounded to double precision, with a worst case overhead of less than a factor ten, and with predictable and acceptable memory consumption. It also explains how to write such efficient correctly rounded functions with little effort.

We believe that those overheads are comparable to those imposed on the hardware by IEEE-754 compliance, and that the benefits of correct rounding are worth this minor performance loss, as it was for the four operations.

Our aim is now to see a gradual generalisation of correctly-rounded functions in mainstream systems (cur-

rently, only Linux incorporates a derivative of Ziv's library, and it is actually enabled only if double-precision is the default in the system [2]). For the functions for which the worst-case accuracy required is known (most notably the exp and log family), there is no longer any technical obstacle preventing this generalisation. For other functions (most notably the trigonometric functions), we will offer proven correct rounding on a small interval only, and on the rest of the function's range, astronomical confidence only. Such a multilevel approach may even be formalised as a standard [8].

In addition to actually writing complete post-ultimate `libms`, there are several research directions to explore. First, the proof framework of `crlibm` needs to be improved. Currently, a proof is a mixture of source code, LaTeX and Maple which provides an extensive and open documentation of each function, but doesn't necessarily inspire confidence as a proof. Then, generic support for FMAs and double-extended precision with compile-time macros could also be added to `crlibm`. Here the difficulty is to manage the error computation in the combination of possible cases. And for processors without double-extended support (and for computing double-extended correctly-rounded functions), a combination of accurate tables [11] and a limited amount of triple-double computation should be explored.

Acknowledgements

Many thanks go the people of Intel Nizhniy Novgorod Lab for interesting discussions, and for giving us access to some of their code. Special thanks go to Andrey Naraikin and Sergey Maidanov, and to Christoph Lauter for his previous work on the exponential. The `crlibm` project was partially funded by INRIA, France.

References

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

[2] Test of mathematical functions of the standard C library. `http://www.vinc17.org/research/testlibm/`.

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

[4] F. de Dinechin, D. Defour, and C. Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Technical Report 2004-10, LIP, École Normale Supérieure de Lyon, Mar. 2004.

[5] F. de Dinechin and N. Gast. Towards the post-ultimate `libm`. Technical Report 2004-47, LIP, École Normale Supérieure de Lyon, Nov. 2004. Available at `http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-47.pdf`.

[6] F. de Dinechin, C. Loirat, and J.-M. Muller. A proven correctly rounded logarithm in double-precision. In *RNC6, Real Numbers and Computers*, Schloss Dagstuhl, Germany, Nov. 2004.

[7] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, Beijing, China, 2002.

[8] D. Defour, G. Hanrot, V. Lefèvre, J.-M. Muller, N. Revol, and P. Zimmermann. Proposal for a standardization of mathematical function implementations in floating-point arithmetic. *Numerical algorithms*, 37(1-4):367–375, Jan. 2004.

[9] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[10] P. M. Farmwald. High bandwidth evaluation of elementary functions. In *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1981.

[11] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.

[12] J. Harrison, T. Kubaska, S. Story, and P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, Q4, 1999.

[13] R. Klatte, U. Kulisch, C. Lawo, M. Rauch, and A. Wiethoff. *C-XSC a C++ class library for extended scientific computing*. Springer Verlag, 1993.

[14] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

[15] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm, 2004.

[16] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, april 2001.

[17] IBM Accurate Portable MathLib. `http://oss.software.ibm.com/mathlib/`.

[18] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[19] P. Markstein. A fast quad precision elementary function library for Itanium. In *Real Numbers and Computers*, pages 5–12, Lyon, France, 2003.

[20] R. Moore. *Interval analysis*. Prentice Hall, 1966.

[21] GNU MPFR. `http://www.mpfr.org/`.

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

[23] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE Computer Society Press.

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