2016 Euromicro Conference on Digital System Design

Unum: Adaptive Floating-Point Arithmetic

Enric Morancho
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Email: enricm@ac.upc.edu

***Abstract*—Usually, arithmetic units represent numeric datatypes employing fixed-length representations. For instance, hardware representations of real numbers usually employ fixed-length formats defined by the IEEE Standard 754 (32-bit single-precision, 64-bit double-precision, ..., floating-point numbers).**

Fixed-length representations allow simpler and faster arithmetic units than variable-length representations. However, fixed-length representations lack the ability to adapt both their accuracy and dynamic range to the application requirements. As some variable-length representations expose this adaptivity, they allow hardware implementations to exploit this adaptivity.

Recently, Unum (universal number) representation has been proposed as an extension of floating-point representations. Unum is a variable-length representation that adapts the bitsize of the representation to the actual numbers being represented and, moreover, Unum associates and propagates accuracy information through arithmetic operations.

In this work we compare Unum versus the floating-point representations defined by IEEE Standard 754. We show that Unum arithmetic improves IEEE 754 arithmetic: a) results obtained using Unum arithmetic are more reliable than IEEE 754's results because Unum does not hide accuracy issues, and b) Unum arithmetic units may implement energy-efficient techniques because Unum dynamically adapts the bitsize of the representation to the actual numbers being represented.

***Keywords*-Floating-point; Unum; Adaptive arithmetic;**

I. INTRODUCTION

Arithmetic units that represent numerical data using fixed-length formats are simpler and faster than arithmetic units that represent numerical data using variable-length formats. However, variable-length formats may allow a more precise representation of numerical data than fixed-length formats

Focusing on the representation of real numbers, since the early days of computer science, computer systems approximate real numbers by floating-point (FP) numbers and operate with them using FP arithmetic [1]. FP numbers are a subset of the real numbers: the rational numbers of the form $(-1)^s \times d_0.d_1d_2 \ldots d_p \times \beta^e$, where $s \in \{0,1\}$ represents the sign, β is the radix (typically 2 or 10), each d_i is a radix-β digit, $d_0 \neq 0$, and e is an integer. FP formats are fixed length and are defined by the parameters radix (β), precision ($p+1$) and exponent range ($e_{min} \leq e \leq e_{max}$).

FP representations are a tradeoff between bitsize and accuracy. Almost all real numbers are inexactly represented; each real number x is rounded to one of the adjacent FP numbers $fl(x)$. Rounding causes an absolute error $|x - fl(x)|$ up to $1/2 \times \beta^{e-p}$ and a relative error $|x - fl(x)|/x$ up to $1/2 \times \beta^{-p}$.

Moreover, FP arithmetic introduces inaccuracies. For instance, operating FP numbers may produce a result that is not representable by the FP format, subtracting nearby FP numbers may introduce a catastrophic cancellation error that magnifies the rounding error of the operands, and both FP addition and multiplication are not associative operations. Furthermore, FP arithmetic does not include accuracy information on its results.

Consequently, FP arithmetic may silently lead to a divergence between computed and exact results. In most cases, this divergence is apparently innocuous and even perfectly acceptable (for instance, approximate computing techniques are a tradeoff between accuracy and energy efficiency [2][3]). However, several works (for instance [4][5][6]) have created pathological arithmetic expressions that FP arithmetic evaluates to extremely wrong results. Also, in some real scenarios, ignoring rounding errors has produced disasters as the failure of Patriot missile battery in 1991 [7].

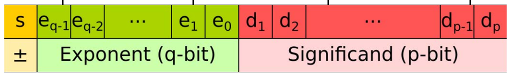
There are several strategies to face the errors that may be introduced by FP arithmetic. Numerical analysis quantifies the error caused by an algorithm and suggests alternatives with a smaller error; for instance, to accumulate the elements of a FP vector, Kahan's summation algorithm [8] reduces the error with respect to the naïve algorithm. [9][10] have proposed runtimes that dynamically detect precision issues. Also, there are alternatives to FP arithmetic like interval arithmetic [5][11][12] and arbitrary-precision libraries [13].

Recently, John L. Gustavson has proposed Unum (universal number) [14], a new representation of real numbers that faces some of the issues of FP representations. Unum representation identifies which real numbers are exactly represented and which ones are not; in the later case, the real number is represented by an open interval. Unum arithmetic takes into account whether the operands of an operation are exact numbers or intervals, and generates the result accordingly. Moreover, unlike conventional FP representations, Unum defines a variable-length format that dynamically adapts the bitsize of the representation to the actual value to be represented.

In this work we compare Unum versus IEEE Standard 754. Our methodology consists in running a set of benchmarks using both representations and comparing their results. We show that Unum arithmetic is not only more reliable than IEEE 754 but also able to adapt the bitsize of the representation to the actual numbers being represented. Consequently, computer architects can exploit this adaptivity to design energy-efficient arithmetic units. The contributions of this work are: a) we have performed a detailed analysis of the behavior of both Unum and IEEE 754 representations over several benchmarks, b) we show the advantages of Unum with respect to IEEE 754.

978-1-5090-2817-7/16 $31.00 © 2016 IEEE
DOI 10.1109/DSD.2016.39

		bitsize $1+q+p$	exponent bitsize (q)	significand bitsize (p)	bias ($2^{q-1}-1$)	dynamic range width
binary32	(single precision)	32	8	23	127	$\approx 6.8\times 10^{38}$
binary64	(double precision)	64	11	52	1023	$\approx 3.6\times 10^{308}$
binary128	(quad precision)	128	15	112	16383	$\approx 2.4\times 10^{4932}$



Exponent (e)		**Fraction (f)** $=\sum_{i=1}^{p} d_i 2^{-i}$ $(0\le f\le 1-2^{-p})$: 0	**Fraction (f)**: $\neq 0$
$e=\sum_{i=0}^{q-1} e_i 2^i$, $e\in\{0,1,...,2^q-1\}$	0	**Signed zeros** ± 0	**Denormal numbers** $\pm f\times 2^{-(bias-1)}$
	2^q-1	**Infinities** $\pm\infty$	**Not a Numbers** qNaN ($d_1=1$), sNaN ($d_1=0$)
	Other	**Normal Numbers** $\pm(1+f)\times 2^{e-bias}$	

Fig. 1. IEEE 754. From top to bottom: main $\beta=2$ formats, layout of an IEEE 754 floating-point number, and interpretation of an IEEE 754 bitstring.

II. Representations of Real Numbers

This section describes two representations of real numbers (IEEE 754 and Unum) and points out their differences.

A. IEEE Standard 754

In 1985, IEEE published the standard 754 for representing and operating radix-2 FP numbers (updated in 2008 to deal with radix-10 numbers [15]). The main goal of IEEE 754 is portability: the same expression should evaluate to the same result on different architectures. The standard defines formats, arithmetic operations, format conversions and exceptions.

IEEE 754 specifies which formats should be implemented by compliant implementations; these formats allow precisely interchanging FP numbers between platforms. Figure 1 details the main characteristics of some $\beta=2$ IEEE 754 formats, encoded using a three-field structure with a 1-bit *sign* (s, $\pm$), a q-bit biased *Exponent* and a p-bit *Significand* (d_0 is implicit).

IEEE 754 considers four classes of FP numbers: normals ($d_0=1$), denormals ($d_0=0$), zeros and infinities. Also, it represents NaNs (not a numbers), that is, undefined (e.g. $0/0$) or unrepresentable (e.g. $\sqrt{-2}$) results. There are two kinds of NaNs: quiet (qNaN) and signaling (sNaN); qNaNs use the *Significand* field to propagate payload information. Figure 1 shows how to interpret and decode an IEEE 754 bitstring.

B. Unum (universal number)

Like IEEE 754, Unum is also based on FP numbers, but there are two key differences with respect to IEEE 754:

- In addition to exactly representing a finite subset of the real numbers, Unum also represents open intervals between two exactly-represented real numbers.
- Unum defines a *bounded* variable-length format that dynamically adapts its length to the represented number.

Unum codification uses 6 fields (Figure 2), where the bitsize of two of them are defined by the Unum format (a 2-tuple <x, y>). The interpretation of each field is: 1) *Sign* (s): 1-bit field like IEEE 754, 2) *Exponent*: variable-length field (from 1 to 2^x bits) that contains the biased exponent, 3) *Significand*: variable-length field (from 1 to 2^y bits) that contains the significand, 4) *Ubit* (u): 1-bit field that denotes if the Unum number corresponds to a FP number (0) or to an open interval (1), 5) *ExponentSize*: x-bit field that contains the current bitsize of the *Exponent* field, and 6) *SignificandSize*: y-bit field that contains the current bitsize of the *Significand* field.

Figure 2 shows the main characteristics of some Unum formats. Unum <3, 4> and <4, 7> formats are comparable to IEEE 754 binary32 and binary128 formats respectively; however, no Unum format is a close match to binary64.

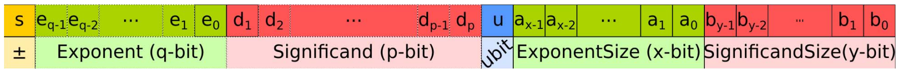
The interpretation of an Unum bitstring depends on the *ubit* field. Figure 2 describes how to interpret an Unum bitstring.

- $ubit=0, s=0$: exactly represented positive real numbers (zero, denormal and normal numbers) and $+\infty$.
- $ubit=1, s=0$: almost all interpretations are intervals one-ulp wide except qNaN, the interval $(0, mindenor)$ and intervals like (t,∞).
- $ubit=0, s=1$: numbers shown in case $ubit=0, s=0$ must be considered as negative, but both -0 and $+0$ represent the same value (exact 0).
- $ubit=1, s=1$: intervals shown in case $ubit=1, s=0$ should be reversed (i.e., transformed from (x,y) to $(-y,-x)$) and qNaN transformed into sNaN.

Unlike IEEE 754, Unum's exponent bias is variable because it depends on the current value of the *ExponentSize* field. For instance, using <3, 5> format, 0.5 is exactly represented using 13 bits (1-bit *Significand*, 2-bit *Exponent*) and bias=1; however, 0.125 is coded using a 3-bit *Exponent* and bias=3.

As Unum arithmetic units deal with numbers and intervals, they are more complex than IEEE 754 units. But, as Unum represents some numbers with fewer bits than IEEE 754, Unum units may be more energy efficient than IEEE 754 units.

	minimum bitsize	maximum bitsize	dynamic-range width
<x, y>	$1+1+1+1+x+y$	$1+2^x+2^y+1+x+y$	$2\times(2-2^{-(2^y-1)})\times 2^{2^{2^x-1}}$
<3, 4>	11	33	$\approx 1.4\times 10^{39}$
<4, 7>	15	157	$\approx 5.7\times 10^{9864}$



$q = 1+\sum_{i=0}^{x-1} a_i 2^i$ $\quad e = \sum_{i=0}^{q-1} e_i 2^i$ $\quad max\_e = 2^{2^x}-1$ $\quad bias = 2^{q-1}-1$

$p = 1+\sum_{i=0}^{y-1} b_i 2^i$ $\quad f = \sum_{i=1}^{p} d_i 2^{-i}$ $\quad max\_f = 1-2^{-2^y}$ $\quad ulp = 2^{e-bias-p}$

$\mathcal{N} = (1+f)\times 2^{e-bias}$ $\quad \mathcal{D} = f\times 2^{-(bias-1)}$ $\quad mindenor = 2^{-bias-p}$

s=0		**ubit = 0** **Fraction (f)**			**ubit = 1** **Fraction (f)**		
		0	$(0, max\_f)$	max_f	0	$(0, max\_f)$	max_f
Exponent (e)	0	**Zero** 0	**Denormal numbers** $\mathcal{D}$		$(0, mindenor)$	$(\mathcal{D}, \mathcal{D}+ulp)$	
	max_e	**Normal Numbers**		**Infinities** ∞	$(\mathcal{N}, \infty)$ if $\overset{q-1}{\underset{i=0}{\forall}} e_i = 1 \wedge \overset{p}{\underset{j=1}{\forall}} d_j = 1$		**NaN's** qNaN
	$Other$	$\mathcal{N}$			$(\mathcal{N}, \mathcal{N}+ulp)$ otherwise		

Fig. 2. Unum. From top to bottom: characteristics of several Unum formats, layout of an Unum number, and interpretation of an Unum bitstring (for $s = 0$).

C. *Differences between IEEE 754 and Unum*

- Rounding: instead of rounding non-exactly-representable real numbers, Unum represents them as an open interval.
- Bitsize: Unum defines a variable-length format that represents some real numbers with fewer bits than IEEE 754.
- Accuracy information: Unum attaches accuracy information to the result of all arithmetic operations.
- Underflow/overflow: Unum neither underflows nor overflows. Positive numbers smaller than the minimum positive denormal number are represented as $(0, mindenor)$ interval. Finite numbers larger than the maximum representable number are represented as an interval like (t, ∞).
- Zeros: Unum interprets both +0 and -0 as an exact 0.
- Arithmetic operations: Unum refines the behavior of the arithmetic operations with respect to IEEE 754. For instance, in Unum, $1/0 = NaN$, but $1/(0, mindenor) = (1/mindenor, \infty)$; however, in IEEE 754, $1/+0 = +\infty$.
- Nan's: Unum does not allow tagging signaling NaN's.

We wonder if Unum representation could be an effective alternative to IEEE 754. In next section we compare the behavior of both representations over a set of benchmarks.

III. Evaluation

A. *Evaluation environment*

Our platform is a conventional x86-64 computer under Ubuntu 14.04. Unum is emulated by a Python module derived from [16], and IEEE 754 is tested by C programs compiled with flags `-mfpmath=387 -ffloat-store` to prevent side-effects with SSE units and x86 extended precision.

B. *Benchmark 1: Numerical instability*

This benchmark consists in computing the 80^{th} term of the recurrence $x_0 = 4$, $x_1 = 17/4$, $x_n = 108 - \frac{815-1500/x_{n-2}}{x_{n-1}}$, created by J.M. Muller, that analytically converges to 5. We have selected this benchmark because it involves just rational numbers and it is numerically unstable [4].

Using either binary16, 32 or 128 IEEE 754's formats, this recurrence wrongly converges to exactly 100 (from term 14, 28 and 46 respect.). However, some intermediate terms differ significantly (x_{14} takes 100, -7.81... and 4.998... respect.).

Computing this recurrence with Unum, we observe:

- Using Unum formats with low precision (up to <3, 8>), we get a NaN because some x_i is an interval that contains 0; then x_{i+1}, that depends on $1/x_i$, is undetermined.
- Using Unum formats with enough precision (for instance, <3, 9>), the result is a tiny interval (about 2×10^{-47} wide) that includes the correct solution.

Consequently, unlike IEEE 754, Unum arithmetic is able to warn about the numerical instability of this computation. This behavior is more reliable than IEEE 754's that silently returns a result that may turn out to be incorrect.

C. *Benchmark 2: Accuracy issues*

This benchmark consists in evaluating a polynomial in the neighborhood of one of its roots. We have chosen this benchmark because the inaccuracies introduced by computer arithmetic may lead to wrong conclusions.

We consider the strictly increasing polynomial $p(x) = x^3 - 3x^2 + 3x - 1$. Figure 3 plots $p(x)$, evaluated using Horner's rule, in the neighborhood of its root $x = 1$ using binary32, Unum <3, 5> and perfect arithmetics.

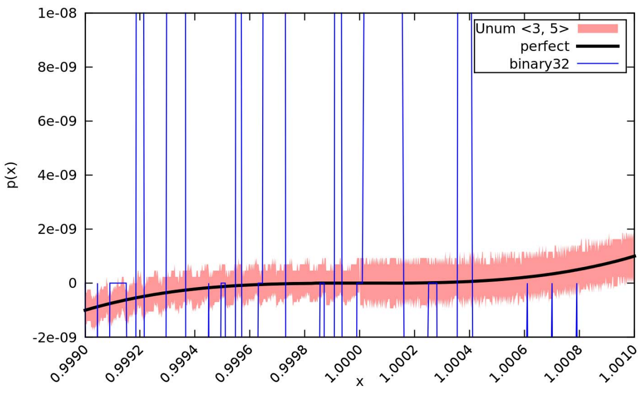


Fig. 3. Plots of $p(x) = x^3 - 3x^2 + 3x - 1$ in the neighborhood of its root $x = 1$ using Unum <3, 5>, perfect and IEEE 754 binary32 arithmetics.

For binary32 arithmetic, we plot $p(x)$ for only 66 equispaced points (some evaluations lie outside the graph, at most $\pm 2 \times 10^{-7}$): we observe that multiple $p(x)$ turn out to be exact zeros and that the plot crosses the x-axis multiple times before and after the root. Comparing binary32 and perfect plots, binary32 arithmetic does not reflect $p(x)$'s behavior.

For Unum <3, 5> arithmetic, we plot a shaded band with $p(x)$ for 1000 equispaced inputs; as expected, the perfect plot lies inside this band. We observe that Unum <3, 5> is also unable to determine the existence of only one root.

Like IEEE 754, Unum can not exactly plot the polynomial. However, as Unum bounds the error of the evaluations, this allows deciding if the obtained results meet the accuracy requirements. If not, one option consists in employing a higher-precision Unum arithmetic; another option consists in adapting to Unum arithmetic an iterative algorithm proposed by Hammer et al. [11] that computes an interval where the exact polynomial evaluation lies in.

D. Benchmark 3: Catastrophic cancellations

This benchmark evaluates polynomials with catastrophic cancellations: a) $f(x, y) = x^4 - 4y^4 - 4y^2$ and b) $g(x, y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/2y$

a) We evaluate $f(665857, 470832)$. Exactly computing this expression involves 78-bit integers and evaluates to 1.

Table I shows the results obtained by IEEE 754 and Unum. IEEE 754 produces erratic results; only binary128 format generates the correct result because its significand field is larger than 78 bits. Using Unum, as precision increases, we get a sequence of nested intervals that includes the correct result.

IEEE 754	Result	Unum	Result
binary32	$\approx -8.8 \times 10^{11}$	<3, 4>	$\approx(-4.6 \times 10^{18},\ 1.1 \times 10^{19})$
binary64	11885568	<3, 5>	$\approx(-3.6 \times 10^{13},\ 1.0 \times 10^{14})$
binary128	1	<3, 6>	(-1024, 7168)
		<3, 7>	1

TABLE I
EVALUATION OF $f(665857, 470832)$ ON SEVERAL ARITHMETICS.

b) We consider computing $g(77617, 33096)$, that evaluates to $-0.8273....$ This expression (proposed by Rump [6] and slightly modified by Loh et al. [17]) shows that increasing IEEE 754 precision is not a reliable technique to detect accuracy issues. The expression evaluates to almost the same result (1.172603...) using binary32, 64 and 128 formats; however, these results are absolutely wrong.

Table II shows the results obtained by several Unum formats. Although small precisions are unable to generate an accurate result, the wideness of the interval warns us about the accuracy issue. However, a large enough precision like <3, 7> can compute up to 14 correct decimal figures.

Unum	Result
<3, 4>	$\approx (-1.9 \times 10^{32},\ 1.9 \times 10^{32})$
<3, 5>	$\approx (-9.9 \times 10^{27},\ 7.4 \times 10^{27})$
<3, 6>	(-1441151880758558718.875, 1441151880758558721.1875)
<3, 7>	(-0.8273960599468..., -0.8273960599468...)

TABLE II
EVALUATION OF $g(77617, 33096)$ ON SEVERAL UNUM FORMATS.

To sum up, although most formats suffer from catastrophic cancellations, Unum does not hide the problem because the wideness of the result warns us about its lack of accuracy; if the accuracy does not meet the requirements, the expression must be re-evaluated using a higher-precision arithmetic.

E. Benchmark 4: Associative law

This benchmark consists in approximating $\pi^2/6$ by summing enough initial terms of the infinite series $\sum_{i=1}^{\infty} 1/i^2$ (convergent to $\pi^2/6$ = 1.644934066...). We have chosen this benchmark because a naïve implementation of this summation reveals that floating-point addition is not associative.

The naïve implementation of this summation starts at $i = 1$ and accumulates the following terms until saturating the accumulation or until reaching the target number of terms.

Unfortunately, this approach leads to an inaccurate result. As the FP adder aligns the radix point of the operand's significands before performing the addition, this alignment can produce a truncation error because some low-order bits of the significand of the smallest (in absolute value) operand get discarded. The extreme case arises when the magnitude-order difference between operands is larger than the significand length; it truncates all significant digits of the smallest operand. In this benchmark, this case arises because the terms of the series are positive and quadratically decreasing.

A better option to perform this summation is proceeding from the smallest significant term to the largest. This way, the operands to be added would have similar magnitude orders, so this reduces the truncation error caused by the alignments in the floating-point adder.

Table III shows the results obtained by several computer arithmetics. Using binary32 arithmetic, the naïve implementation saturates at 1.64472... after summing the first 4.100 terms, but only four figures are correct. However, doing the summation backwards, the summation of the first 4.100 terms

Arithmetic	Summation from i=1 to 4100	Summation from i=4100 to 1
IEEE 754 binary32	1.64472...	1.64469...
Unum <3, 4>	(1.6392..., 1.7017...)	(1.64465..., 1.64472...)
Unum <3, 5>	(1.6446896..., 1.6446906...)	(1.64469013409..., 1.64469013526...)

TABLE III
EVALUATION OF $\sum_{i=1}^{4100} 1/i^2 = 1.64469013466...$ ON SEVERAL COMPUTER ARITHMETICS.

is 1.64469..., and we can accumulate up to 10^8 terms at 1.644934058... with eight correct figures.

Using Unum arithmetic, the resulting interval always contains the correct answer although the loss of information also caused by Unum adder. Moreover, Unum also benefits from inverting the summation order because the resultant interval is narrower than the obtained with the naïve approach.

Although FP addition is not associative both in IEEE 754 and in Unum, Unum arithmetic computes an interval that contains the correct result. Like IEEE 754, Unum also benefits from carefully implemented algorithms: the better the algorithm, the narrower the result. As some high-performance techniques reorder arithmetic operations, they may also exhibit accuracy issues due to associative-law fails; for instance, distributing a computation between several processors and later combining the partial results into a final result. Unum obtains accurate results independently on operation ordering.

F. Benchmark 5: Representation bitsize

To test the effectiveness of Unum's variable-length format, in this benchmark we compute the determinant of an $n \times n$ Vandermonde matrix ($M_{i,j} = \alpha_i^{j-1}$) with exactly-represented real coefficients. To stress both IEEE 754 and Unum arithmetics, we will use a naïve algorithm to evaluate the determinant, although we know in advance that the determinant of these matrices is exactly $\prod_{1 \leq i<j \leq n}(\alpha_j - \alpha_i)$. To generate matrices with exactly-represented coefficients, we consider $n = 5$ and $\alpha_i = 2^{-ik}$ where $1 \leq i \leq n$ and k is an integer.

Table IV shows, for several Unum formats the bitsize range of each format (computed as detailed in Figure 2) and, for $k = 1$ to $k = 4$, the average bitsize of the Unum numbers involved in the computation of the Vandermonde determinants. Shaded cells mean that the related Unum formats are unable to compute the exact value of the determinant. Also, the bitsize distribution is shown for a particular case.

We observe that Unum is able to automatically adapt the bitsize of the representation to the actual values involved in the computation.

The precision required to compute the exact result of the determinant depends on the value of parameter k; for instance, for $k = 2$, at least <3, 6> is required. If we use a Unum format with a higher precision, the bitsize increment is restricted to the fixed-length fields *ExponentSize* and *SignificandSize*; the effective length of the variable-length fields remain unaltered. If a computation generates a result that is not exactly representable by the Unum format, the bitsize of this result is the maximum bitsize supported by the format.

Finally, as IEEE 754 formats are fixed-length, their average bitsize is exactly the format length (Figure 1), independently on the actual values.

We expect that a hardware implementation of Unum will exploit the bitsize adaptivity we have shown. It may be useful to reduce the arithmetic-unit latency, to increase energy efficiency and to reduce memory bandwidth with respect to fixed-length implementations.

IV. RELATED WORKS

Some works have proposed alternatives to approximating real numbers by FP numbers. For instance, Swatzlander and Alexopoulos [18] proposed the Sign/Logarithm Number System that allows a faster implementation of multiply and divide operations, without excessively downgrading addition and subtract operations. Matula [19] proposed Fixed-Slash and Floating-Slash representations, that approximate real numbers by means of rational numbers p/q where, unlike FP numbers, q can take any representable integer value. Ewe et al. [20] proposed Dual Fixed-Point, a tradeoff between fixed-point and FP representations where only two scaling factors can be applied to the significand field; this allows the design of arithmetic units simpler and faster than FP arithmetic units.

Interval arithmetic is an attempt to automatize error analysis in computer arithmetic. R. Moore [12] wrote the seminal work on computer interval arithmetic. Hammer et al. [11] developed C++ libraries based on interval arithmetic that automatize the verification of numerical results. In 2015, IEEE published the standard 1788 for interval arithmetic [21]. The main differences between Unum and traditional interval arithmetic are: a) interval arithmetic does not consider open intervals, b) typically, interval arithmetic represents an interval as two FP numbers, c) Unum refines the behavior of some arithmetic operations in the corner cases, d) interval arithmetic usually produces as a result too wide intervals due to both the wrapping and the dependency problems. Unum minimizes both problems by using *uboxes* [14], that allow dividing the computation into several parts and solving each one using the required precision.

Some authors have developed, for specific problems, algorithms that minimize the accumulation of rounding and truncation errors. For instance, Kahan [8] proposed an algorithm for summing a vector of floating-point numbers. Also, Kulisch et al. [22] proposed a hardware implementation of the dot-product algorithm that is a key component on its proposal of an advanced computer arithmetic [5].

To identify computations that suffer from catastrophic cancellations, Lam et al. [10] and Benz et al. [9] instrument binary code and, by computing in alternate precisions, suggest

	bitsize range	Average bitsize			
		k = 1	k =2	k=3	k=4
<3, 4>	11…33	17.06	20.54	21.90	22.49
<3, 5>	12…50	17.52	21.71	24.28	26.53
<3, 6>	13…83	18.52	21.37	23.62	28.60
<3, 7>	14…148	19.52	22.37	24.62	26.90

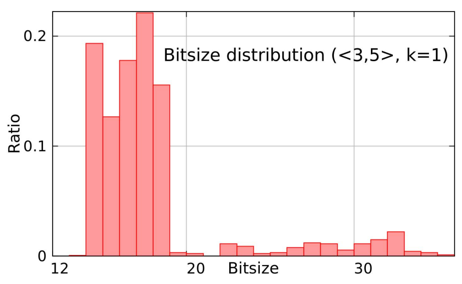


TABLE IV
AVERAGE BITSIZE OF UNUM NUMBERS INVOLVED IN THE COMPUTATION OF VANDERMONDE'S DETERMINANT (SHADED CELLS MEAN THAT THE COMPUTED DETERMINANT IS NOT EXACT), AND BITSIZE DISTRIBUTION FOR CASE <3, 5>, K=1.

the sources of inaccuracies. As Unum does not hide accuracy issues, Unum automatically reveals cancellations.

As employing the highest precision throughout all the programs may be unnecessary (from the accuracy point of view) and inefficient (from the performance point of view) some works suggest that programs should be implemented in mixed FP precision. Rubio et al. [23] and Lam et al. [24] developed frameworks that determine the right precision for each variable so that the program produces an accurate enough result. Unum achieves the same goal because it dynamically adapts the bitsize of numbers to the actual value being represented.

Finally, as some application domains are error-resilient (for instance, multimedia processing), some authors have proposed approximate computing techniques that relax the accuracy requirements in the computed result [2][3]; these techniques are a tradeoff between energy and accuracy. Using a low-precision Unum format achieves the same goal.

V. CONCLUSION

Unum has been proposed as a variable-length representation of real numbers that is aware of whether it is dealing with an exact number or with an approximation. Unum is able to propagate this information through the computations and to attach accuracy information to each number.

Our evaluations show that Unum arithmetic is more reliable way than IEEE 754 because Unum does not hide accuracy issues. As Unum offers accuracy information on its results, the user can easily verify if the results satisfy the accuracy requirements of the problem. Moreover, Unum adapts the bitsize of the representation to the actual numbers to be represented; consequently, in some scenarios, Unum uses less bits than IEEE 754 to represent numbers.

We believe that Unum is an interesting alternative to IEEE 754: its reliability may increase programmer's productivity and its bitsize adaptivity may allow the implementation of energy-efficient arithmetic units.

We plan to continue this work by designing and evaluating a hardware implementation of Unum arithmetic.

ACKNOWLEDGMENT

We acknowledge the support of the Spanish Ministry of Education (TIN2015-65316-P) and the HiPEAC4 NoE.

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley, 1997.
[2] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design." IEEE Computer Society, 2013.
[3] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.
[4] W. Kahan, "How futile are mindless assessments of roundoff in floating-point computation?" p. 37, Nov. 2004.
[5] U. W. Kulish and W. L. Miranker, "The Arithmetic of the Digital Computer: A New Approach," *SIAM Review*, vol. 28, no. 1, March 1986.
[6] S. M. Rump, "Algebraic computation, numerical computation and verified inclusions," in *Trends in Computer Algebra*, ser. Lecture Notes in Computer Science, vol. 296. Springer, 1987, pp. 177–197.
[7] U. G. A. Office, "Patriot Missile Defense. Software Problem Led to System Failure at Dharan, Saudi Arabia." feb 1992.
[8] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Commun. ACM*, vol. 8, no. 1, pp. 40–, Jan. 1965.
[9] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," in *33rd Conference on Programming Language Design and Implementation*, 2012, pp. 453–462.
[10] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic floating-point cancellation detection," *Parallel Computing*, vol. 39, no. 3, 2013.
[11] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz, *C++ Toolbox for Verified Computing 1*. Springer-Verlag GmbH, 1995.
[12] R. E. Moore, *Interval analysis*. Prentice-Hall, 1966.
[13] "The GNU MPFR Library." [Online]. Available: http://www.mpfr.org/
[14] J. Gustafson, *The End of Error: Unum Computing*, ser. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015.
[15] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*.
[16] J. Muizelaar, "Python port of the mathematica unum prototype," oct 2015. [Online]. Available: https://github.com/jrmuizel/pyunum
[17] E. Loh and W. Walster, "Rump's example revisited," *Reliable Computing*, vol. 8, no. 3, pp. 245–248.
[18] E. Swartzlander and A. Alexopoulos, "The sign/logarithm number system," *IEEE Transactions on Computers*, vol. 24, no. 12, 1975.
[19] D. W. Matula, "Fixed-slash and floating-slash rational arithmetic," in *3rd Symposium on Computer Arithmetic (ARITH)*, Nov 1975.
[20] C. T. Ewe, P. Y. K. Cheung, and G. A. Constantinides, "Dual fixed-point: An efficient alternative to floating-point computation." ser. LNCS, vol. 3203, 2004, pp. 200–208.
[21] "IEEE Standard for Interval Arithmetic," *IEEE Std 1788-2015*.
[22] U. W. Kulisch and W. L. Miranker, *Computer arithmetic in theory and practice.*, ser. Computer science and applied mathematics, 1981.
[23] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 27:1–27:12.
[24] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Proceedings of the 27th International Conference on Supercomputing*, 2013, pp. 369–378.