On Basic Financial Decimal Operations on Binary Machines

Abhilasha Aswal, M. Ganesh Perumal, and G.N. Srinivasa Prasanna, *Senior Member*, *IEEE*

Abstract—Financial transactions are specified in decimal arithmetic. Until the introduction of IEEE 754-2008, specialized software/hardware routines were used to perform these transactions but it incurred a penalty on performance. In this paper, we show that if binary arithmetic is used to emulate decimal operations, then arbitrary error sequences can be generated by carefully chosen sequences of transactions which can lead to monotonically increasing/decreasing capitalization errors. In addition, we describe methods for correctly performing basic decimal operations, such as addition, subtraction, multiplication, and division, on binary machines, which are not conformant with IEEE 754-2008 decimal floating point standard (ISO/IEC/IEEE 60559:2011), at high speed.

Index Terms—Floating-point arithmetic, IEEE 754 standards, computer arithmetic.

1 Introduction

Many early computers used decimal arithmetic at the hardware level, but binary computing in hardware soon took over after Von Neumann et al. [6] pointed out the advantages of simplified hardware. In 2008, the IEEE Standard 754-1985 [23] was revised to define standards for decimal floating-point arithmetic (IEEE 754-2008, ISO/IEC/IEEE 60559:2011) [24], [25], yet most general purpose computers still implement binary arithmetic. Financial calculations, which are specified in decimal arithmetic, are required to be exact both by the law and to keep the account books free of inconsistencies. If we use binary floating-point arithmetic to perform the decimal calculations in financial transactions, then errors that are legally unacceptable may creep in [3], [4]. Although, machines that are pre IEEE 754-2008 compliant can use specialized software libraries to perform decimal calculations exactly on the underlying binary hardware, it comes with a penalty on performance. In this paper, we show that blindly using binary hardware will result in unacceptable errors and also present an alternative approach that shows how high speed decimal arithmetic can be implemented on a binary machine conformant with IEEE 754 for binary floating point, but not with IEEE 754-2008 for decimal floating point.

1.1 Contributions of the Paper

Beginning with financial transaction properties, we

1. Devise an appropriate computational model:
   a. Compared to previous work [3], [4], we systematically categorize financial transactions and show the existence of arbitrary error patterns by a suitable choice of transaction amounts.
2. Investigate the impacts of using standard computer architectures without modification, for financial applications:
   a. Our results are based on examination of the error (ϵ) which is defined as the difference between the answer produced using arbitrary precision decimal arithmetic, and that produced by the IEEE 754 binary arithmetic, after financial rounding rules are applied.
   b. We show using a matrix representation of possible errors, that extensive errors can appear for certain sequences of transaction values. Such special sequences of transactions can lead to monotonically increasing/decreasing errors which are erroneously attributed to wrong implementation rather than the limitations of using binary arithmetic for decimal calculations. We show that it is possible to design sequences of transactions which can yield an error at every step making the total error increase/decrease arbitrarily, when computing using binary arithmetic. We show that such error sequences can be deliberately triggered by analyzing the possible errors. The errors are typically very small in magnitude, but sequences of transactions exist, whose cumulative error is significant, and we show that almost any required error process can be generated.
3. Investigate appropriate modifications required to contemporary architectures, if any (compliant with IEEE 754, not IEEE 754-2008 decimal floating point):
   a. We show in this paper that existing binary hardware in conjunction with arbitrary precision software libraries are capable of exact decimal calculations as encountered in finance, at high speed.

- *A. Aswal and M.G. Perumal are with the International Institute of Information Technology-Bangalore and the Education and Research Department at Infosys Limited, 26/C Electronics City, Hosur Road, Bangaluru 560100, Karnataka, India. E-mail: {abhilasha.aswal, ganesh_perumal}@iiitb.ac.in.*
- *G.N.S. Prasanna is with the International Institute of Information Technology-Bangalore, 26/C Electronics City, Hosur Road, Bangaluru 560100, Karnataka, India. E-mail: gnsprasanna@iiitb.ac.in.*

Manuscript received 15 Sept. 2011; revised 4 Apr. 2012; accepted 9 Apr. 2012; published online 19 Apr. 2012.
Recommended for acceptance by E. Antelo, D. Hough, and P. Ienne.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2011-09-0654.
Digital Object Identifier no. 10.1109/TC.2012.89.

Our methods rely on an analysis of the possible errors in using binary for decimal calculations, and the very coarse rounding grid for currencies (which may not be the case for general applications utilizing decimal arithmetic). In rare cases, when our methods are detected to be potentially erroneous, we safely take recourse to a standard decimal calculation software library (e.g., BigDecimal, decNumber).

We need no special hardware decimal instructions. The results will enable the vast majority of pre-IEEE 754-2008 binary processors to be used for financial calculations, at high speed. Power limited mobiles can be employed using our techniques for financial computations also.

4. Our methods can generalize to implementing arithmetic in any radix, given hardware implementing any other radix with sufficient extra precision.

In the rest of this paper, we discuss these ideas in detail. Section 2 discusses some generic issues in handling decimal financial calculations on binary machines. Section 3 discusses a mathematical model of transactions. Based on this, Section 4 presents an analysis of the error process and Section 5 does a worst case analysis of error patterns, and shows that sequences of transaction volumes exist which can cause errors in each transaction. Section 6 presents software optimizations that will enable us to use binary hardware to perform exact decimal calculations faster than the software libraries. Here, we show how binary arithmetic compliant machines can do high-speed decimal financial calculations, without using decimal hardware. Section 7 concludes.

2 Financial Calculations on Finite Precision Machines

The IEEE 754 floating point standard defines the rules for the approximation of real numbers in finite precision machines, as well as for arithmetic operations like addition, subtraction, multiplication, division, etc. Floating-point representations of a number define a base β and a precision p. IEEE 754-1985 is a binary standard where $\beta = 2$ and $p = 24$ (7.22 decimal digits) for single precision and $p = 53$ (15.95 decimal digits) for double precision. Length of a word in single precision is 32 bits, 23 bits for the significand, and 8 bits for the exponent and 1 sign bit. In double precision a word is 64 bits long, 52 bits for the significand, and 11 bits for the exponent and 1 sign bit. In 2008 this standard was updated to IEEE 754-2008, where the radix can be either 2 or 10. IEEE 754-2008 defines the interchange formats, rounding algorithms, operations, and exception handling. It also includes the recommendations for sophisticated exception handling, other operations such as logarithmic and trigonometric, expression evaluation and to achieve reproducible results. Although the IEEE standard was revised in 2008, partial hardware support for decimal arithmetic (not compliant with IEEE 754-2008) has existed for a long time, for example, in IBM Z9 mainframes at microcode level, and in IBM Z10 and POWER6 and later POWER processors at hardware level.

Whenever the result of an operation is inexact then by default IEEE standards rounding rule approximates the answer to the nearest representable number. There are two rounding to nearest schemes for this. The standard defines three other rounding modes, called directed roundings, round toward 0, round toward $+\infty$ (ceiling function), and round toward $-\infty$ (floor function). The exact result lies between result rounded toward $+\infty$ and result rounded toward $-\infty$. Let X be the exact value and $\hat{X}$ be the approximate value, which follows:

$$\hat{X} \in [floor(X), ceiling(X)].$$

The difference between the exact value and the rounded value is known as the rounding error, $\epsilon = X - \hat{X}$. Financial regulations define more rounding rules (Standard rounding, Swiss rounding, Argentine rounding, etc.), to round the results to the nearest representable currency unit. These rounding rules do not always round to two decimal places, but sometimes may round to three decimal places as in the case of Argentine rounding. In this case, the rule states that if the third digit after the decimal is less than three, it is dropped, otherwise, if the third digit is between two and eight, it is changed to five, otherwise, if the third digit is greater than seven, then the third digit is dropped and the second is incremented by 1 [17].

Financial calculations are specified in decimal, and significant rounding errors may occur due to conversion to/from the binary representation. For example, the binary representation of 0.1 lies strictly between two binary floating-point numbers and cannot be exactly represented by either of them so it is approximated to $1.1001100\ 11001100110011001100110011001100110011001100110011010 \times 2^{-4}$ in IEEE 754 double precision, which when converted back to decimal gives 0.1000000000000000005551115123126.

Indeed among the 10 quantities $[0.0, 0.1, 0.2, 0.3, \ldots, 0.9]$, only two (0.0, 0.5) have exact binary representations. Further complications arise in financial calculations, due to huge monetary volumes (10^{13} units or more), and legal requirements of accuracy to the last currency unit. Even small relative errors in calculation may lead to large errors, with possible legal consequences over a period of time and this may have an impact on a banks balance sheet. As such, financial calculations require specialized software libraries which include implementations of the basic operations like addition, subtraction, multiplication, and division to preserve accuracy. These workarounds are typically slow (because of lack of hardware support for decimal arithmetic in majority of machines) [4] and hence the performance of these systems may be affected. If IEEE 754 binary arithmetic is used directly instead of specialized software/hardware then we show that the results may not be accurate enough to satisfy legal requirements. In addition to this, we show how to correct such errors in an efficient manner. We also give an algorithm for financial transactions involving multiplications and divisions which can be used to perform transactions with sufficient accuracy using binary arithmetic.

3 Model of Financial Transactions

We precisely specify the operations of a financial system in this section. We assume that all transactions begin with an exactly representable account balance in decimal, in the currency in which the account is specified. An exactly representable decimal amount Δ is deducted from the payer, in the payer's currency, and an exactly representable decimal

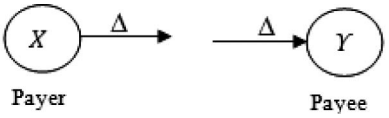

Fig. 1. A payer and payee—single sided.

amount is added to the payee, in the payee's currency. Intermediate calculations and representations may be in either double precision binary or exact decimal arithmetic. This financial model, while simplified, is adequate for obtaining insight. Let x be a decimal amount specified in currency c. Let $\xi(x)$ convert a x into its binary equivalent $\hat{x}$, $\rho_c(x)$ be the function to round the decimal amount x based on the rounding rules for currency c and let $\xi'(\hat{x})$ convert a binary number $\hat{x}$ into its decimal equivalent.

3.1 Single Node

An account in a bank can act as a payer or a payee, and the transactions are withdrawals and deposits. The transaction amount Δ could be an arbitrary real value based on some calculations like interest payment. This amount Δ may not be exactly representable if IEEE 754 standard for binary arithmetic is used. In this case, we are interested in ϵ, the error in the capitalization (difference between the exact value, and that calculated using IEEE 754 followed by currency specific rounding methods) at the node X (payer) or Y (payee) as shown in Fig. 1.

For computing this in exact decimal arithmetic, we have the relations between the initial and final quantities as shown in (3.1) and (3.2). Let X_i and Y_i represent the initial balance amount at payer node and payee node, respectively. Let X_f and Y_f represent the final balance, after a transaction, at payer node and payee node, respectively. We assume that both the payer node and the payee node have the same currency c.

$$X_f = \rho_c(X_i - \rho_c(\Delta)), \tag{3.1}$$

$$Y_f = \rho_c(Y_i + \rho_c(\Delta)). \tag{3.2}$$

For calculations in double precision, we have an initial step where the decimal account balances are converted from their exact decimal values. Carets are used to refer to binary approximations of decimal quantities.

$$\hat{X}_i = \xi(X_i), \tag{3.3}$$

$$\hat{Y}_i = \xi(Y_i), \tag{3.4}$$

$$\hat{\Delta} = \xi(\rho_c(\Delta)), \tag{3.5}$$

$$X_f = \rho_c(\xi'(\hat{X}_i - \hat{\Delta})), \tag{3.6}$$

$$Y_f = \rho_c(\xi'(\hat{Y}_i + \hat{\Delta})). \tag{3.7}$$

The computations in (3.3) to (3.7) are in accordance to IEEE 754 standards and include any IEEE 754 specific rounding as the operands are binary numbers.

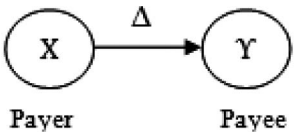

Fig. 2. A payer and payee—single sided.

3.2 A Transaction Pair

A pair of accounts transacting with each other forms a transaction pair. The nodes of the pair may have the same base currency, in which case the transactions are single currency transactions, or they could have different base currencies, where the transactions are multicurrency. Of interest here are the errors in the capitalization of the payer, X or the payee, Y or the total, $X + Y$. If the base currency of both payer and payee is same (c) as shown in Fig. 2, then there is no currency conversion in the transaction and there are no errors in decimal arithmetic if rounding is not applied.

$$X_f = X_i - \rho_c(\Delta), \tag{3.8}$$

$$Y_f = Y_i + \rho_c(\Delta), \tag{3.9}$$

$$X_i + Y_i = X_f + Y_f. \tag{3.10}$$

The error due to binary computation is derived using (3.3) to (3.7) as

$$\epsilon \triangleq (X_f + Y_f) - (X_i + Y_i).$$

Here, X_f and Y_f are derived using (3.3) to (3.7).

Similarly, if the base currency of X and Y is different, then there is a currency conversion step in the transaction, which involves a multiplication by the exchange rate η_{XY} to convert currency of X into currency of Y, as shown in Fig. 3.

In this case, the transaction equations will change. There is an additional currency specific rounding step which will make the final total capitalization different from the exact decimal answer. Due to difference in currencies, the total capitalization is best calculated using infinite precision. Let x, y be the currencies of X and Y, respectively.

$$X_f = \rho_x(X_i - \rho_x(\Delta)), \tag{3.11}$$

$$\begin{aligned} Y_f &= \rho_y(Y_i + \rho_y(\eta_{XY}\rho_x(\Delta))), \\ \eta_{XY}X_i + Y_i &\stackrel{?}{=} \eta_{XY}X_f + Y_f. \end{aligned} \tag{3.12}$$

The use of binary arithmetic can cause additional error. The binary equations will change to the following:

$$\begin{aligned} \hat{\Delta} &= \xi(\rho_x(\Delta)) \\ \hat{\eta} &= \xi(\eta) \\ X_f &= \rho_x(\xi'(\hat{X}_i - \hat{\Delta})), \end{aligned} \tag{3.13}$$

$$\begin{aligned} Y_f &= \rho_y(\xi'(\hat{Y}_i + \xi(\rho_y(\xi'(\hat{\eta}_{XY}\hat{\Delta}))))) \\ \eta_{XY}X_i + Y_i &\stackrel{?}{=} \eta_{XY}X_f + Y_f. \end{aligned} \tag{3.14}$$

3.3 Interest Payments and Splits

Periodic interest payments result in a correlated sequence of transaction volumes. A k-way split disbursement of funds is

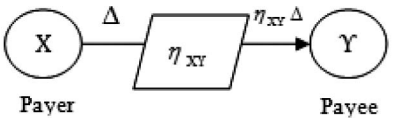

Fig. 3. A payer-payee pair (multicurrency).

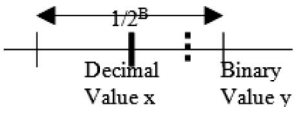

Fig. 4. Error between decimal and binary values.

similar to a pairwise transaction, except that the initial amounts are in general nonexactly representable (e.g., Rs. 1.00 split into three pieces). Additional errors beyond the cases discussed above result. If an amount Z is to be split into n parts, then an error between 0 and $Mod(Z,n)$ can occur, i.e., the error, $\epsilon \in \{0,1,\ldots,n-1\}$. Hence, for any given n, the maximum error can be $n-1$. A careful selection of Z based on the information of the value of n can be used to maximize ones gains.

3.4 A Financial Network

A network is formed when accounts across multiple banks or accounts are transacting with each other. Here, each node (account) can have a different base currency so it is basically a multicurrency network. Let there be N nodes (accounts across various banks) in the network. Let X_{j_i} and X_{j_f} represent the initial and final balances, respectively, of the jth account. The total capitalization of the network can be computed by converting the account balances of all the accounts into a common currency (say, \$) and then summing them up. Even if all computations are exact, the total capitalization computed in this manner at two different instances may differ. If there are N accounts in the network and Let X_{j_i} and X_{j_f} represent the initial and final balances, respectively, of the jth account, then

$$\sum_{j=1}^{N} X_{j_i}\eta_{X_j\$} \stackrel{?}{=} \sum_{j=1}^{N} X_{j_f}\eta_{X_j\$}$$

This is because first, the currency exchange rates are specified only up to six significant figures and second, there are currency specific rounding rules. Due to this, opportunities for arbitrage exist. Whenever there are erroneous computations, these differences may be magnified.

4 Analysis of Error Process

We analyze the behavior of the error process using the model of transactions discussed in Section 3. We discuss the properties of the error in representing a single value, followed by properties of error accumulation in basic operations. Our analysis is general, and applies to *approximating arithmetic in one base, by computations in any other base*.

4.1 Error in Representing a Single Value

First consider a single decimal value in D digits, and its nearest binary representation in B bits. We assume that default round to nearest rounding direction mode is being used. We consider that all numbers are normalized to unity (we need only consider this case for insight, if D and B are suitably chosen). Let the decimal value be denoted by x and its nearest binary equivalent y (see Fig. 4). Analysis of the error between x and y yields insight into the behavior of decimal arithmetic approximated by binary representation. Below, we shall use the notation $[m_{10}, m_2]$ to represent a pair $[x,y]$, where the first component is the exact decimal value (scaled by 10^D), and the second its (approximate) binary representation. We have:

$$\begin{aligned} x &= 10^{-D}m_{10}, 0 \le m_{10} < 10^D \\ y &= 2^{-B}m_2, 0 \le m_2 < 2^B \\ & 10^{-D}m_{10} - 2^{-B-1} \le 2^{-B}m_2 \le 10^{-D}m_{10} + 2^{-B-1}. \end{aligned} \tag{4.1}$$

Lemma 4.1 below gives information about decimal values that are exactly representable in binary.

Lemma 4.1. *The error between the decimal number and binary approximations is periodic in 10^{-D} and 2^{-B}.*

Proof. Let L be a generalized version of least common multiple (LCM) for fractional numbers 10^{-D} and 2^{-B}

$$\begin{aligned} L &= lcm(10^{-D}, 2^{-B}) \\ \text{Period in decimal, } P_{10} &= \frac{L}{10^{-D}}, \end{aligned} \tag{4.2}$$

$$\text{Period in binary, } P_2 = \frac{L}{2^{-B}}. \tag{4.3}$$

Since $L \le 1$, cycling through all possible decimal fractions will encounter at least one full period of the error. □

This is used for analysis in Section 5.

For example (Fig. 5), for $D=1$ and $B=3$, (one decimal digit and its nearest binary equivalent) $L = lcm(\frac{1}{10}, \frac{1}{8}) = \frac{1}{2}$.

- The decimal fractional numbers are 0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9.
- The binary fractional numbers are 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, and 0.875.

Every fifth decimal number is approximated with the same error, with the binary number incrementing by four. Example of the periodic behavior is in Fig. 9 in Section 5.

It is well known that read-write cycles have no errors if,

Lemma 4.2. *The errors occur when the binary is rounded to the next nearest decimal, i.e., the rounding boundary is crossed. This can be avoided by encoding the decimal with sufficient binary digits as given below.*

$$B \ge \lceil log_2 10^D \rceil. \tag{4.4}$$

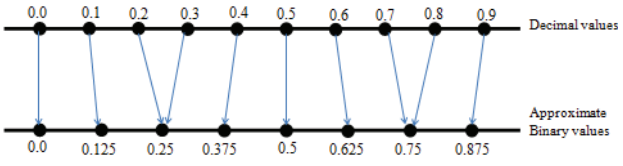

Fig. 5. Mapping decimal values (with one place of significance) to the corresponding approximate binary values (of three significant bits).

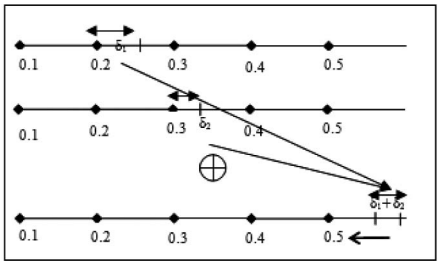

Fig. 6. Error in addition.

Proof. Using Lemma 4.1, we can see that for the choices of values of B that satisfy (4.4), the binary window will be smaller than the decimal window, hence ensuring that the rounding boundary is not crossed. □

Our contribution is to generalize this decimal-binary-decimal conversion cycle to those involving at least one arithmetic operation.

With these basic definitions, we now examine the properties of basic mathematical operations.

4.2 Error in a Single Addition

If the balance amount x is to be represented in binary format, let δ_x be the error associated with it (Fig. 6). Let y be the transaction (deposit) amount and δ_y be the error associated with it.

$$\delta_x = x - \xi'(\hat{x})$$
$$\delta_y = y - \xi'(\hat{y}).$$

The result after the addition will be $\xi'(\hat{x} + \hat{y})$. The exact result is $(x + y)$ which is equal to $\xi'(\hat{x} + \hat{y}) + \delta_x + \delta_y$. As long as $\delta_x + \delta_y < \frac{1}{2} \times$ (smallest unit for x and y), no errors are made after standard rounding is applied.

To ensure that $\delta_x + \delta_y < \frac{1}{2} \times$ (smallest unit for x and y) the binary approximation has to have at least two more bits than the decimal. For most currencies, we have two decimal places (7 bits), and the 53 bits in IEEE 754 significand cause no addition errors for numbers with integer portions up to 2^{44}. Errors are made in addition only with large numbers (greater than or equal to 2^{45}), which reduce precision of significand.

4.3 Error in a Single Multiplication

The story is different for multiplications (Fig. 7), since the product of two exactly representable numbers can easily hit a rounding boundary. For example, consider $1.5 \times 0.01 = 0.015$, which gets rounded to 0.02 according to currency rounding rules for currencies specified to two digits after the decimal. If 0.01 is approximated as 0.0099..., then the same operation yields $1.5 \times 0.0099 = 0.0149999$, which gets rounded to 0.01. Rounding errors in multiplications occur with even small numbers (but this is not frequent for coarse decimal grids See Section 6 for details).

$$(x \times y) = (\xi'(\hat{x}) + \delta_x) \times (\xi'(\hat{y}) + \delta_y)$$
$$\cong \xi'(\hat{x})\xi'(\hat{y}) + \xi'(\hat{x})\delta_y + \xi'(\hat{y})\delta_x + \delta_x\delta_y.$$

Another common decimal operation in financial applications is the IEEE 754-2008 *quantize* operation. This is essentially a multiplication by a power of 10 and the same analysis as above is applicable (Details in Section 6).

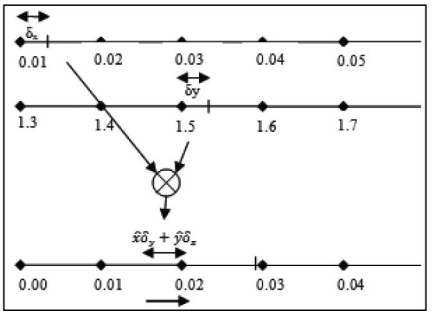

Fig. 7. Error in multiplication.

With this analysis of errors in number representation and basic operations, we can discuss the impact of finite precision in financial calculations, following our taxonomy given in Section 3.

5 Error Process: Worst Case

Based on (3.1) to (4.4) we discuss the various transactions below, in order. A novel tabular approach is used to examine the worst case errors, as outlined below. These errors are rare, requiring either very large numbers or accidental matches. This fact will be exploited to obtain high-speed routines using only binary arithmetic in Section 6.

5.1 Single Node, Payer/Payee: Transaction Error Matrix

With capitalization amounts exceeding $10^{13}(2^{45})$, errors are possible in additions, as shown in Section 4. Here, using a tabular approach, we demonstrate a sequence of transaction amounts, such that an error is made in every transaction. First, we define the capitalization-transaction error matrix (CTEM) $T(D, B)$, where D is the number of decimal digits in the fraction and B is the number of binary bits in the binary approximation, as an $n \times m$ matrix, with entries $T_{ij} =$ Error in adding/subtracting Transaction amount Δ_i to/from Capital C_i, where n is the number of possible capitalization values and m is the number of possible transaction amounts. This error is with respect to exact decimal arithmetic. To differentiate between the CTEM for deposits and withdrawals, we represent CTEM as $T_+(D, B)$ and $T_-(D, B)$, respectively. We can also have a CTEM, $T_\times(D, B)$, for multiplication which would result when a currency conversion rate is multiplied with a transaction amount or interest rate is multiplied with a capital amount. Table 1 shows a CTEM $T_+(1, 3)$ for deposit transactions (additions). Each entry T_{ij} is the error that results from adding the ith capital to jth transaction amount. For example, if the account balance is 0.1 units and we want to deposit 0.2 units in the account, the following steps are followed to find the error:

1. Three digit binary equivalent of 0.1 is 0.001 and the three digit binary equivalent of 0.2 is 0.010.
2. Adding the binary equivalents we get new total amount as 0.011.

TABLE 1
CTEM $T_+(1,3)$

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.1	-0.1	-0.1	0	0	0	-0.1	-0.1	0	0
0.2	-0.1	-0.1	0	0	-0.1	-0.1	-0.1	0	0
0.3	0	0	0.1	0.1	0.1	0	0	0.1	0.1
0.4	0	0	0.1	0.1	0	0	0	0.1	0.1
0.5	0	-0.1	0.1	0	0	0	-0.1	0.1	0
0.6	-0.1	-0.1	0	0	0	-0.1	-0.1	0	0
0.7	-0.1	-0.1	0	0	-0.1	-0.1	-0.1	0	0
0.8	0	0	0.1	0.1	0.1	0	0	0.1	0.1
0.9	0	0	0.1	0.1	0	0	0	0.1	0.1
1	0	-0.1	0.1	0	0	0	-0.1	0.1	0

TABLE 2
CTEM $T_\times(1,3)$

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.1	0	0	0	0	0	0	0	0	0
0.2	0	0	0	0	0	0	0	0	0
0.3	0	0	0	0	0	0	0	0.1	0.1
0.4	0	0	0	0	0	0	0	0	0
0.5	0	0	0	0	0	0	0	0	0
0.6	0	0	0	0	0	0	-0.1	0	0
0.7	0	0	0	0	0	-0.1	-0.1	0	0
0.8	0	0	0.1	0	0	0	0	0.1	0.1
0.9	0	0	0.1	0	0	0	0	0.1	0.1
1	0	0	0	0	0	0	0	0	0

3. Converting this new total amount back to decimal we get 0.375.
4. Rounding this result to 1 decimal digit, we get 0.4.
5. The rounding error can be calculated by taking the difference between the correct sum (0.3) and the computed sum (0.4) which is -0.1, which is the T_{12}^{th} entry in the CTEM $T_+(1,3)$.

Table 1 shows a CTEM $T_+(1,3)$ for deposit transactions (additions). Similarly, Table 2 shows a multiplication CTEM $T_\times(1,3)$ where each entry T_{ij} is the error that results from multiplying the ith capital to jth interest rate. **Properties of the CTEM. It is easy to see the following:**

1. For a given binary precision, the CTEM is completely specified by using only all possible values of the fractional (decimal) portion of transactions and amounts. The integer portions add exactly, and do not cause errors, assuming there are no overflows.
2. $T_+(D,B)$ is a symmetric matrix and rows and the columns correspond to the smallest currency unit.

Proof. Since addition is commutative, the matrix is symmetric. □

$T_-(D,B)$ is a skew symmetric matrix and the rows and columns correspond to the smallest currency unit.

Proof. Since in subtraction, $A - B = -(B - A)$, the matrix is skew-symmetric. □

3. Since a row of $T_+(D,B)$ CTEM corresponds to using all possible decimal fractional values less than or equal to unity, the error period is $\leq$ the row length. For CTEM in Table 1 we can get the error period using (4.2) and (4.3). As discussed earlier in Section 4, P_{10} is five and P_2 is four. From the CTEM, we observe that the error is indeed repeating itself after every five entries.
4. Table 3 shows $T_+(2,4)$ for two decimal digit values approximated using 4 bits. We have shown only every fifth row and fifth column of the CTEM for brevity. The period of the error in decimal arithmetic, P_{10} in this case is determined from (4.2) and (4.3), with $D = 2$ and $B = 4$. Equations (4.2) and (4.3) yields $P_{10} = 25$, and we can see from the table that the error indeed repeats itself after every 25 values. This means that the same error occurs four times at least in a row.

5.1.1 Generation of Arbitrary Error Sequences Using the CTEM

Analysis of this matrix CTEM enables us to generate worst case, and in general arbitrary transaction sequences, which we describe below. For simplicity of illustration, we use CTEM $T_+(2,4)$ in Table 3. Fig. 9 (later) illustrates the same for a CTEM for an actual currency transaction approximated in IEEE 754 double precision.

So there are four possible transactions values that give the same error. If there are N transactions, then for getting a given error sequence, there are 4^N possible ways of choosing

TABLE 3
CTEM $T_+(2,4)$

	0.05	**0.1**	**0.15**	**0.2**	**0.25**	**0.3**	**0.35**	**0.4**	**0.45**	**0.5**	**0.55**	**0.6**	**0.65**	**0.7**	**0.75**	**0.8**	**0.85**	**0.9**	**0.95**
0.05	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0
0.1	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01
0.15	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04
0.2	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03
0.25	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01
0.3	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0
0.35	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01
0.4	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04
0.45	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03
0.5	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01
0.55	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0
0.6	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01
0.65	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04
0.7	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03
0.75	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01	0	-0.01	-0.03	0.03	0.01
0.8	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0	-0.01	-0.03	-0.04	0.01	0
0.85	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01	-0.03	-0.04	-0.05	0	-0.01
0.9	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04	0.03	0.01	0	0.05	0.04
0.95	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03	0.01	0	-0.01	0.04	0.03

the transaction amounts. Of course, the total number of transaction sequences is 100^N for two decimal digits, far higher than the number of transaction sequences which yield any given error sequence. However, the exponential number of transaction sequences having a given error sequence, enables a pseudorandom combination of these sequences to pass at least simple statistical tests, and such errors to escape detection. We have heuristically generated experimental sample sequences from the CTEM in Table 3 such that there was an overall bias across the transactions. The algorithm to generate a sequence of N transactions is given below:

1. Select a random row (R) from the CTEM.
2. IF the row has any positive errors THEN Select a random column (C) that has a positive error with probability P or select a random column (C) that has a nonpositive error with probability $(1 - P)$ ELSE Select a random column (C) from anywhere in the row.
3. Find the next row: $R \leftarrow$ decimal portion of $(R + C + error)$.

Repeat steps 2 and 3, N times.

If P is tuned to be biased, that is $P > 0.5$, and the number of rows with no positive error is smaller than the number of rows with some positive error, then an overall positive bias will result in the sequences. The algorithm is fast—for generating sequences of 1,000 transaction amounts and performing those transactions took around 1.7 seconds on an Intel Core 2 Duo, 2.00 GHz, 2 GB RAM machine.

It is possible to generate these sequences in a manner which pass statistical tests on the transaction amounts, together with the payee and payer account amounts (this is the only information available at the level of banker's logs). Sample sequences generated using the above algorithm of varied sizes, passed the basic tests for randomness such as chi-squared test, t-test, and F-test. The details are given in the Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2012.89.

5.1.2 CTEM for Transactions Using IEEE 754 Double Precision

For a given precision in binary, the CTEM will vary for different capitalizations. For additions and subtractions, the CTEM is unique over a very large range of values. For example, with IEEE 754 double precision (not IEEE 754-2008 decimal floating point) as the binary approximation, there are 53 bits available for the significand. For capitalization amounts whose integer portions are less than 2^{45} (35184372088832), i.e., occupy only up to 45 bits, there are no errors because we begin and end on a coarse currency grid. For capitalizations having integer portions between 2^{45} and 2^{46} (35184372088832 and 70368744177664), there are errors and for those having integer portions greater than 2^{46} (70368744177664), there are an even larger number of errors. We can see the snapshots of the three corresponding CTEMs in Fig. 8. Each of them is obtained by considering a base integer capitalization C_0 and creating from it a set of all possible decimal capitalizations, specified to two decimal places, as follows:

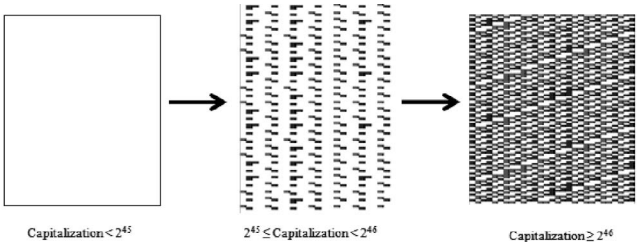

Fig. 8. Changes in CTEM from a small capitalization to a larger capitalization.

$$C_0 = \text{integral base capitalization}$$
$$C_j = C_{j-1} + 0.01,\ j > 0.$$

We now add transaction amounts from 0.01 to 0.99 to each of these capitalizations to generate CTEM. Every transaction will either produce an exact answer or an approximate answer, depending on the numbers involved. We classify these results as zero error, positive error, and negative error, where positive error is encountered when the approximated binary result is greater than the exact decimal result and negative error is encountered when the approximated binary calculation result rounded to decimal is less than the exact decimal result.

In the succeeding examples, IEEE 754 double precision is used as the binary approximation. The Appendix, available in the online supplemental material, gives details of a couple of sample additions based on IEEE 754 double precision as the binary approximation. The white cells of CTEM denote zero error, black cell denote a positive error, and gray cells denote a negative error. We can see that the number of errors increases with the size of the capitalization.

For all base capitalization values lying between 2^x and 2^{x+1}, the number of significand bits is the same, and hence the errors and CTEM is the same. This CTEM can be used to find a sequence of transactions which have an overall bias but are statistically indistinguishable from a random sequence. As an example, consider the set of capitalizations specified to two decimal places $C_i, 0 \leq i \leq N$, given as follows:

$$C_0 = 42345678901234.00$$
$$C_j = C_{j-1} + 0.01, j > 0.$$

Using this CTEM, we can generate transaction amount sequences such that a positive (or negative) error is encountered in every transaction.

Fig. 9 represents the output of this exercise in color-coded form where the positive errors are represented by black cells and negative errors are represented by gray cells and zero errors by white cells. It can be seen that a large number of cells have either a positive or a negative error. The error is also clearly periodic. It is possible to find a sequence of transaction amounts for this such that either always a negative error is made or always a positive error is made. Two such paths are shown, each for all positive and all negative errors.

For the positive error sequence, the starting capitalization was 42345678901234.02, chosen randomly. To this capital amount, 0.24 is added. This results in an error of $+0.01$, which means that instead of 42345678901234.26, the double result is 42345678901234.27. To this new capital amount, 0.03

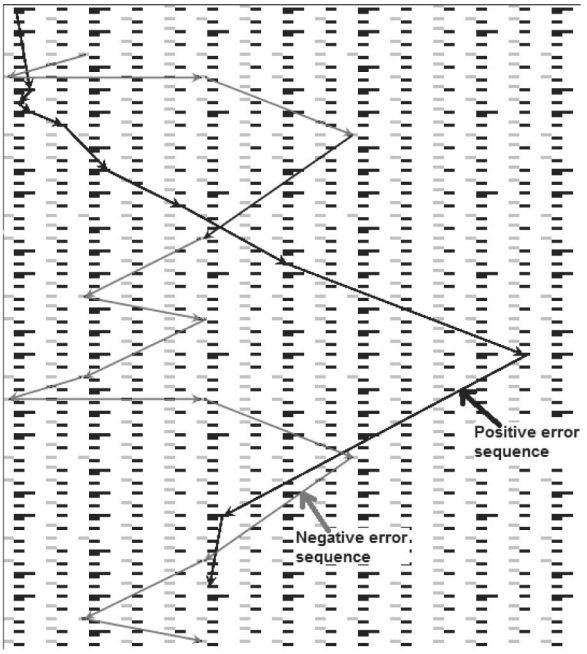

Fig. 9. Sample error sequences in a CTEM matrix.

is added, which again leads to an error of $+0.01$, and so on. The complete sequence of transaction amounts is [0.24, 0.03, 0.02, 0.03, 0.06, 0.06, 0.10, 0.17, 0.27, 0.49, 0.21, 0.20]. At the end of this sequence, the capitalization amount becomes 42345678901236.02 and we can use the same sequence again to always get a positive error. Hence, if we carry on cyclically going through this sequence of transaction amounts, we will continuously accumulate money in our account.

Similarly, it is possible to find a sequence such that always a negative error is made. Starting from the capitalization 42345678901236.16, the following sequence of transactions will always give a negative error: [0.08, 0.01, 0.19, 0.33, 0.19, 0.08, 0.19]. This is marked by gray colored arrows in Fig. 9.

If one takes short sequence of transaction amounts and cycles through them thousands of times, such sequences can be caught by statistical tests. However, it is possible to have a sequence that appears random and still makes large number of errors. For the example above, the following pseudo random sequence [0.24, 0.03, 0.02, 0.03, 0.06, 0.06, 0.10, 0.17, 0.27, 0.49, 0.21, 0.20, 0.10, 0.56, 0.63, 0.28, 0.31, 0.06, 0.35, 0.99, 0.70, 0.42, 0.35, 0.13, 0.53, 0.38, 0.10, 0.02, 0.60, 0.67, 0.24 . . .], makes error at every step. Someone with the knowledge of capitalization can deliberately trigger such a sequence to their advantage.

Table 4 shows the results of chi-squared test done on two biased sequences of 1,000 transactions each (not shown for brevity), derived from the CTEM in Fig. 9. The result gives enough evidence of the randomness of the sequences.

Similarly, the CTEM enables us to design transaction volumes which do not always cause errors, but whose error process appears random and passes statistical tests but gains/loses money over a long enough time horizon.

5.2 Pair, Single Currency

If a pair of banks/accounts are transacting as a payer-payee pair and both have the same base currency, then with capitalization amounts exceeding 10^{13}, it is possible to make an error in every transaction. Here too, we find an always increasing or an always decreasing path in the CTEM. Consider, for example, that the payer's initial balance is 56426386341314.01 currency units, and the payee balance is 39879899999879.03. Then, if the payer makes the following sequence of payments repeatedly to the payee, then it is always possible to make an error in every transaction at the payer, and in a large number of transactions at the payee: [0.08, 0.15, 0.12, 0.22, 0.19, 0.30]. There will be a positive error at the payer and a negative error at the payee.

TABLE 4
Chi-Squared Test Results on Sequences on CTEM in Fig. 9 with Number of Transactions $= 1{,}000$

Bias	Chi-square value	$p-$value (%)	Degrees of freedom
21.33	12.1742	85.64	8
21.07	7.57	52.3	8

5.3 Pair, Multicurrency

While the preceding examples used very large transaction amounts, this is not required when multicurrency transactions are used. When we have a transacting pair with different base currencies, it is still possible, with relatively small capitalizations, to find a sequence of transactions that will always give an error. This is due to the multiplication in the conversion—this can yield errors with small numbers also (Fig. 7, in, Section 5.1, and explanations therein). Arbitrary error sequences can be found by finding a path through the CTEM. Fig. 10 shows the partial CTEM for a payer-payee pair of Australian Dollar-Iranian Rial, at an exchange rate observed on 12/08/2011 (10923.3).

The capitalization amounts started from 1.01 Rials with every subsequent capitalization obtained by adding 0.01 Rials to the current one. We add transaction amounts from 0.01 to 0.99 Australian Dollars to each of these capitalizations to generate the CTEM. This involves converting the Australian Dollar amount to Rials and then adding the resultant amount to the capitalization. Every transaction will either produce an exact answer or an approximate answer, depending on the numbers involved. The red cells show a negative error. We found that for this currency pair, we either got a zero error or a negative error for any capitalization. The following sequence of 100 transaction amounts gives a negative error of 0.01 Rials 25 out of 100 times and passes the

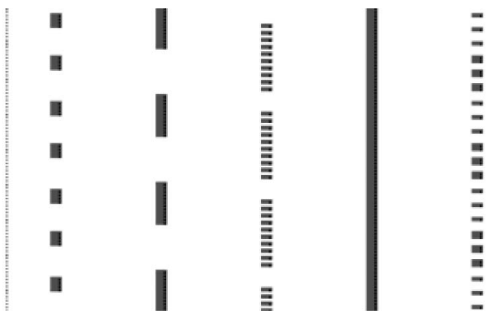

Fig. 10. CTEM for Iranian Rial (payee) for Australian Dollar-Iranian Rial pair with exchange rate as on 12/08/2011 (10,923.3 Rial/Australian Dollar).

TABLE 5
Errors Resulting from a Simulation of 50,000 Multicurrency Transactions between 10,000 Accounts

Account	IEEE balance	Exact arithmetic balance	Error	Currency
1	9999542.29	9999542.3	-0.01	US Dollar
2	10714249.51	10714249.52	-0.01	Indonesian Rupiah
3	10088641.42	10088641.43	-0.01	Iranian Rial
4	13843428.15	13843428.16	-0.01	Cambodian Riel

Chi-squared test of randomness—[0.45, 0.5, 0.61, 0.85, 0.95, 0.24, 0.57, 0.25, 0.95, 0.25, 0.85, 0.35, 0.35, 0.45, 0.12, 0.25, 0.24, 0.23, 0.86, 0.3, 0.75, 0.35, 0.65, 0.25, 0.61, 0.9, 0.25, 0.25, 0.65, 0.11, 0.19, 0.6, 0.25, 0.75, 0.25, 0.9, 0.65, 0.55, 0.45, 0.85, 0.31, 0.85, 0.4, 0.85, 0.34, 0.45, 0.17, 0.65, 0.95, 0.25, 0.47, 0.25, 0.31, 0.45, 0.65, 0.65, 0.82, 0.65, 0.58, 0.77, 0.74, 0.48, 0.35, 0.95, 0.21, 0.25, 0.45, 0.65, 0.22, 0.25, 0.25, 0.31, 0.79, 0.59, 0.72, 0.85, 0.61, 0.12, 0.95, 0.39, 0.38, 0.85, 0.9, 0.35, 0.25, 0.97, 0.95, 0.17, 0.11, 0.95, 0.38, 0.1, 0.35, 0.97, 0.35, 0.18, 0.29, 0.35, 0.52, 0.47]. The total money lost in 100 transactions was 0.25 Rials. Note that since the double precision result is less than the accurate amount, transaction fees cannot fix it they worsen the problem! For multicurrency transactions, it is imperative to use exact decimal arithmetic even small accounts can show large transaction errors (see Section 6 for more details).

5.4 Interest Payments, Currency Splits

A similar analysis can be made for interest payments and amount splits for dividend payments (e.g., dividing one Rupee into three equal parts), with similar results—we should use exact decimal arithmetic.

5.5 Network

An entire financial system operates as the composition of transactions between individual currency pairs. The results for the system can be inferred from the results for individual pairs. Table 5 shows some of the erroneous results after a simulation run with 10,000 accounts, each having one of 20 different currencies, and 50,000 money transfer transactions randomly occurring between pairs of accounts. If the accounts have different currency, the transaction involves a currency conversion. The starting balance of all the accounts was 10^7 currency units. The exchange rates were the actual rates taken as on 10 August 2011. In all, there were 19 errors in 50,000 transactions. The rarity of these errors will be exploited in Section 6.

6 Safe Decimal Arithmetic on Binary Machines

With the machinery discussed in the earlier sections, we will now illustrate how exact decimal arithmetic can be implemented, using primarily binary arithmetic computations, with recourse to a decimal library occasionally. No new decimal hardware instructions are required.

Historically financial data are stored as strings in databases. We assume this string data are loaded in RAM for our performance estimates (else disk accesses dominate). We show new and improved calculation methods, working on all architectures even those predating IEEE 754-2008, and hence assume that we do not have data in BID/DPD format.

Our methods rely on the fact that the decimal grid in finance is far coarser than the binary precision, and are rarely erroneously crossed by the binary result (Fig. 12). In the majority of cases, the binary answer is good enough to be rounded as per the currency rules. This rounding step is typically currency specific and not in general consistent with IEEE 754-2008 with most of the digits being dropped, unless the numbers are very large. Section 6.2 shows ((6.1) to (6.7)) how this step can be done at high speed.

We show the clock cycle results for calculating basic decimal financial operations—addition and multiplication, with 16 digits of precision which encompasses a range large enough for a large class of financial applications ([14, Table 6] and remarks therein). Note that these 16 digits need not come from decimal32 or decimal128 formats. Our methods are referred to the case when the datum has 16 significant digits irrespective of the format. Clock cycle counts do not translate directly into actual runtimes, but the results are indicative of the promise of our approach. The ideas can be analogously extended for 34 digits of precision, but will come with a conversion overhead. We have not discussed exception processing here for brevity, but the same exceptions can be generated as those by decimal64/128.

Below, we discuss addition briefly and multiplication in detail. For simplicity of exposition, we have not presented the fully optimized versions of our method. We present equations valid for exact binary arithmetic but they straightforwardly generalize to arithmetic accurate to half ulp.

6.1 Addition (Deposit/Withdrawal Single Currency)

This relatively common operation will be briefly discussed. Our methods can be used together with scaling, if the loss of dynamic range is acceptable (this is not so for multicurrency transactions).

Since we begin with string data, the first step is to convert to a format handleable by the machine. To read in decimal data where the exponent is explicitly specified (i.e., specified in terms of the lowest currency unit), we use a standard conversion algorithm (e.g., Gay's algorithm [20]) to convert to binary, assuming round to nearest.

On the other hand, to read in decimal data where the exponent is specified as an attribute (e.g., value is in Millions), we can modify the string read routine to shift by an appropriate number of decimal places *before* conversion to binary. This should not cause much increase in time for the string conversion since it involves no computations, but only a book-keeping of shift of the decimal point (the shift is completely determined by the attribute, e.g., data in millions imply shift the decimal point six places).

The integer portions of the operands are converted to binary as described above. The fractional portion is dealt with as discussed below.

Our method uses the CTEM selectively, based on numeral magnitudes to get exact decimal rounded results, and at high speed. We first note that the CTEM provides not only the error, but also equivalently information about if 1) a carry/borrow results from the addition/subtraction and 2) the fractional part of the answer. This can be used to correct the error.

Our method simply hence performs the transaction by converting the integer portion using an standard algorithm

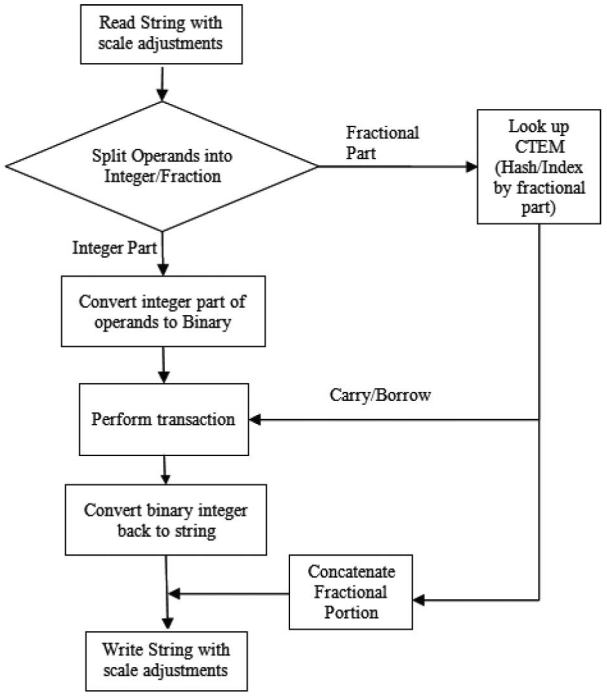

Fig. 11. Flow chart for performing addition/subtraction transactions (database to database cycle).

(e.g., [20]), adds/subtracts using binary integer arithmetic, adds a carry if needed from a CTEM lookup, and converts back to decimal. The CTEM lookup also provides the decimal fractional portion. This is faster than high-precision software libraries, and decimal hardware implementations [14], [21] are not required. The CTEM for addition and subtraction is small enough (only 10,000 entries for currencies specified up to two digits after decimal and at most 1,000,000 entries for currencies specified up to three digits after decimal) to be loaded in to the main memory in contemporary computers. Also, there are at most nine different CTEMs, one each for the range 2^x to 2^{x+1}, $45 \leq x \leq 53$ (see Section 5.1.2, Fig. 11).

To write back, when the scale doesn't have to be modified, the standard binary to string conversion routines can be used. However, in financial operations scales have to be often modified. In such cases, the string write routine can be modified to shift by an appropriate number of decimal places.

An alternative implementation of addition can scale the fractional parts by 100 or 1,000 to only perform integer operations. If in this case a rescaling back of the final answer is required, then we have to divide by an appropriate power of 10 or multiply by the inverse. Here, we can use our (6.7) with $\delta_R = 1$, and $\frac{\delta_R}{2} = 0.5$ which has an exact representation in binary.

The typical cost of the above method is only about 250 cycles. This includes the cost of conversion from string to binary (2×50 cycles), the cost of actual binary addition, and the CTEM lookup (20 cycles) and the cost of converting final answer back to string (130 cycles). Our conversion numbers are derived from Gay [20]—the cost of conversion from string to binary format is in 10 s of cycles (typically 40-60) and the cost of converting back is 130 cycles (typically 120-140). In contrast, an addition operation using BID64 typically takes 320 cycles including the costs of conversions [21]. Hence, we are in the same scale as decimal instructions, while using binary architecture, and faster than software libraries.

In general, for any average bank user, nearly 90 percent of the transactions are deposits and withdrawals and using the additive CTEM alone in the system, we can greatly improve the speed with which the transactions are performed.

This addition algorithm is shown in the self-explanatory flow chart in Fig. 11, which depicts the database to database operations in performing a transaction.

We note that if a series of transactions are to be performed, then the intermediate results can be retained in binary, as long as the resulting capitalization and transaction amount are both less than 2^{44}.

Our analysis will be helpful if databases store binary approximations for decimal numbers rather than using strings. Equation (4.1) will give insight into the approximations. In addition Section 5 gives insight into possible dangers of blindly ignoring the problem.

6.2 Multiplication

Using the CTEM approach directly is not possible for transactions including a multiplication operation such as currency conversion and interest payments as the error and hence the CTEM depends on both the integer portion and the fractional portion of the operands (this is unlike addition). Scaling the numbers to avoid fractional portions completely entails a significant amount of loss of dynamic range. Specifically, in a multicurrency calculation, with an exchange rate represented to six significant digits and with currencies having $2/3$ fractional digits, $8/9$ orders of dynamic range are lost, which is unacceptable. An alternative approach is required, one of which is presented below. We present equations valid for exact binary arithmetic. For IEEE 754 arithmetic accurate to half ulp, our equations can be fixed, but the frequency of calling the software digital library can increase slightly.

When we multiply two decimal numbers x and y in binary, it is the binary approximations of x and y that get multiplied and the result that we get may be different from the exact decimal result. Lets call this approximate result as α.

$$\begin{aligned}\alpha &= (x(1+\delta_x))(y(1+\delta_y)) \\ &= xy + xy(\delta_x + \delta_y) + xy(\delta_x\delta_y).\end{aligned}$$

Here, xy is the exact decimal result and $xy(\delta_x + \delta_y) + xy(\delta_x\delta_y)$ is the error part. Since δ_x and δ_y, the errors in representing decimal values by their nearest binary equivalents, are *unknown* (and can be negative) for any given x and y, we cannot know how far the approximate result α is from the exact result xy, unless we perform some complex calculations. However, δ_x and δ_y are at most one ulp. This enables us to bracket the exact value xy, in the following manner.

Let $\hat{x}$ be the nearest exactly representable binary number that is greater than or equal to x, as per IEEE 754. Let $\hat{\hat{x}}$ be the nearest exactly representable binary number that is less than or equal to x. Similarly $\hat{y}$ and $\hat{\hat{y}}$ are the upper and lower binary approximations to y, as per IEEE 754. These are obtained using standard decimal to binary conversion algorithms (e.g., Gay's algorithm [20]). Clearly, the exact product satisfies

$$\hat{\hat{x}}\hat{\hat{y}} \leq xy \leq \hat{x}\hat{y}.$$

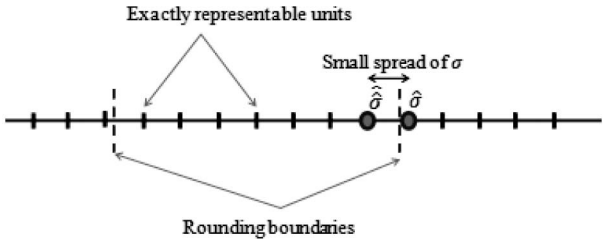

Fig. 12. Upper and lower bounds relative to the currency rounding boundaries.

Bounds on $\hat{x}$ and $\hat{\hat{x}}$ and $\hat{y}$ and $\hat{\hat{y}}$ and the respective products, are readily obtained by adding/subtracting one in the last significand place as

$$\hat{x} = x(1 + \delta_x + 2^{-53}), \tag{6.1}$$

$$\hat{y} = y(1 + \delta_y + 2^{-53}), \tag{6.2}$$

$$\hat{\hat{x}} = x(1 + \delta_x - 2^{-53}), \tag{6.3}$$

$$\hat{\hat{y}} = y(1 + \delta_y - 2^{-53}), \tag{6.4}$$

$$\hat{\sigma} = \hat{x}\hat{y}, \tag{6.5}$$

$$\hat{\hat{\sigma}} = \hat{\hat{x}}\hat{\hat{y}}. \tag{6.6}$$

We assume that round to nearest is being used. Alternatively, if the mode bits can be changed at high speed on a particular implementation of the IEEE 754 (not IEEE 754-2008 decimal floating point) standard, then at least two of these additions are not needed.

Finally, the number of currency units, N_c, corresponding to xy correctly rounded, then satisfies

$$\left\lfloor \left(\hat{\hat{\sigma}} + \left(\frac{\delta_R}{2} \right)_- \right) \delta_R^{-1} \right\rfloor \le N_c \le \left\lfloor \left(\hat{\sigma} + \left(\frac{\delta_R}{2} \right)_+ \right) \delta_R^{-1} \right\rfloor. \tag{6.7}$$

Here, $(\frac{\delta_R}{2})_-$ and $(\frac{\delta_R}{2})_+$ are half the rounding spacing decremented and incremented by one in the lower order significand bit, respectively. The value of δ_R^{-1} is either 100 or 1,000 (from currency rounding spacing).

Equation (6.7) enables us to bracket the numbers of currency rounding intervals, and gives us the rounded value immediately if the upper and lower limits are the same integer. This integer can be converted to a string using any of the standard routines [20] along with scaling as in addition.

If the upper and lower limits differ, then a call can be made to software. However, because of the high precision of IEEE 754 (IEEE 754-2008 decimal floating point is not required), and the coarse currency grid, this case is rare (see the performance calculations in Section 6.2.2 and Fig. 12).

The algorithm is summarized in the flow chart in Fig. 13.

Our method is similar to the *quantize* operation. The comparison between the lower and the upper bound as given by (6.7) is essentially a quantize operation for two or three digits. For more digits we simply adjust the value of δ_R accordingly. A left shift will be exact, unless there is an overflow. Equation (6.7) can be used for a right shift.

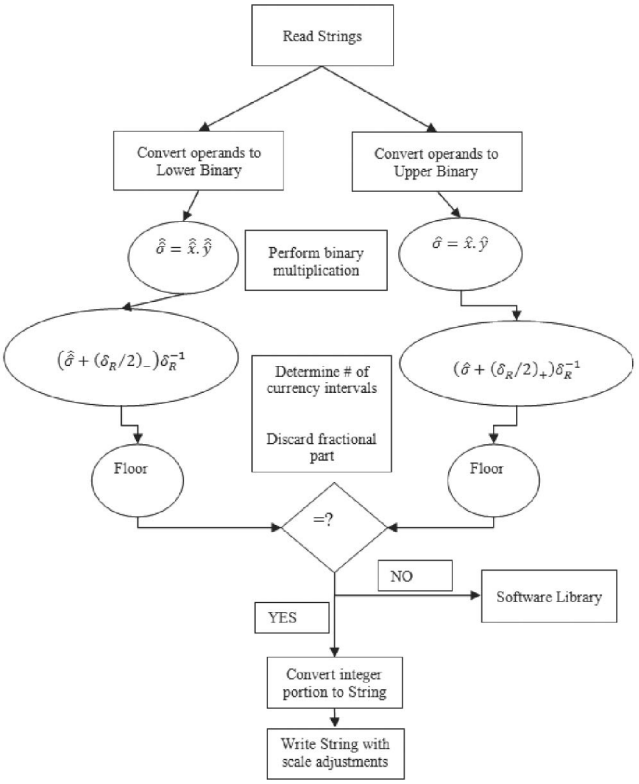

Fig. 13. Flow chart for performing multiplication transactions (database to database cycle).

6.2.1 Speed of Multiplication

Below we estimate the speed of our multiplication algorithm on an Intel Core 2, Wolfdale, 45 nm, the same as that referred to in [14]. Table 7 shows the clock cycles for major operations. The estimated maximum cycles includes the full instruction latency, and the average cycles includes half of the total latency, as a first approximation. The estimated maximum and average clock cycles for selected operations are shown in the Table 7 for Decimal floating point(Binary Integer Decimal encoding-BID64 [14] (for *comparison only*, these are *not used*)) and Binary floating point (Intel Core 2, Wolfdale, 45 nm [15], [16]).

In Table 6, the estimated clock cycles needed for checking if binary is sufficient or not to perform decimal multiplications is estimated, which is 36 cycles on the average. If we include the string to binary and back conversion times to this, then the total cost is 270 ($2 \times 50 + 40 + 130$) cycles. From our implementation of this algorithm (refer to Appendix C, available in the online supplemental material) the estimated time was typically 280 cycles. This includes the cost of arithmetic operations as well as string conversions.

We compare this with a typical of 327 cycles [14] [21] for a Multiplication, using BID64—we are in a similar scale, while using pre IEEE 754-2008 architecture, and faster than software libraries.

In this calculation, we have ignored all possibilities of pipelining and parallelism, e.g., steps 1, 2 and 4, 5 can be done in parallel, etc. Depending on the available parallelism mix, the speeds will be even higher than what we have reported here.

TABLE 6
Estimated Clock Cycles to Check if Binary64 Is Sufficient for the Multiplication Algorithm

S.No.	Expression	Add	Sub	Mul	ToInt	CMP	Max cycles	Avg cycles
1	$\hat{x} = x\left(1+\delta_x+2^{-53}\right)$	1					4	2.5
2	$\hat{y} = y\left(1+\delta_y+2^{-53}\right)$	1					4	2.5
3	$\hat{\sigma} = \hat{x}.\hat{y}$			1			6	3.5
4	$\hat{\hat{x}} = x\left(1+\delta_x-2^{-53}\right)$		1				4	2.5
5	$\hat{\hat{y}} = y\left(1+\delta_y-2^{-53}\right)$		1				4	2.5
6	$\hat{\hat{\sigma}} = \hat{\hat{x}}.\hat{\hat{y}}$			1			6	3.5
7	$(\hat{\hat{x}}.\hat{\hat{y}}).\delta_R^{-1}$			1			6	3.5
8	$(\hat{x}.\hat{y}).\delta_R^{-1}$			1			6	3.5
9	Rounding				2		18	12

6.2.2 Performance Analysis of Multiplication

The overall achieved speed depends on how frequently the slow software decimal library path is executed, which is data dependent. Under uniform assumptions about data, the frequency of invocation of the slow decimal path can be bounded to be less than the ratio of the total spread of the upper and lower limits $(\hat{\hat{\sigma}} - \hat{\sigma})$ to the rounding spacing δ_R. Even with transaction magnitudes xy averaging 10 digits (more than 10 billion), (x can represent a capitalization, y an exchange rate, or x can represent a capitalization, and y an interest rate/tax rate) IEEE 754 decimal32 format gives seven digits of precision. Currency spacings are at least 0.001, and most are 0.01. The ratio $\frac{\hat{\hat{\sigma}}-\hat{\sigma}}{\delta_R}$ is then, under uniform assumptions about input data, upper bounded by $4 \times \frac{10^{-7}}{10^{-3}}$ (there are two operands in each of the two multiplications, each of which is accurate to seven digits), which is four parts in 10,000. Even a typical software library taking 1,000 cycles, will add only 0.4 additional cycle to the total instruction count, which can be neglected (see Fig. 12). If certain values of either operand (prices just under one unit, e.g., 0.95, 0.99, etc., taxes rounded to .05 percent), are common, and trigger the worst case, caching schemes can be employed to reduce the total runtime.

Same analysis is applicable for the *quantize* operation. If we have to do a 15 digit left shift when handling values with 16 significant digits, then using the above logic we will have to use a software library only 10 percent of the time. This being an extreme case, on average the percentage of calling the software library will be much lesser.

The runtimes can be slow if someone deliberately chose a transaction sequence such that the software library is called a large number of times. But note that unlike the CTEM, where a small bias is enough for accumulating money, for significantly affecting the runtimes of the application, a very large number of calls have to be made to the slow software library, which will be caught. Also, anybody doing this will not gain anything, and hence it is unlikely to be done.

TABLE 7
Estimates of Clock Cycles for Selected Operations in BID and BFP

Instruction	BID Cycles	BFP (Intel Core 2, Wolfdale, 45nm) (in cycles)	
		Estimated Maximum Cycles	Estimated Average Cycles
FADD	109	4	2.5
FSUB	126	4	2.5
FMUL	117	6	3.5
Copy	9	4	3
ToInt	148	9	6
Compare	5	2	2

6.3 Division

An analogous procedure can be done for division. The binary result in this case is

$$\alpha = \frac{x(1+\delta_x)}{y(1+\delta_y)}. \tag{6.8}$$

To find a bound on this, we can compute $\hat{x}$, $\hat{y}$, $\hat{\hat{x}}$, and $\hat{\hat{y}}$ using (6.1), (6.2), (6.3), and (6.4). The upper and the lower bounds can be found as follows:

$$\frac{\hat{\hat{x}}}{\hat{y}} \leq \frac{x}{y} \leq \frac{\hat{x}}{\hat{\hat{y}}}. \tag{6.9}$$

Based on this, an analogous procedure can be computed for division. The details are omitted for brevity.

6.4 General Expressions and Other Optimizations

Our methods generalize to calculating any expression in decimal arithmetic, whether or not intermediate rounding is mandated. All we need to do is generate the upper and lower bounds, round at the end or where specified, and compare the answers. If they match the answer is as obtained, else we use a software library.

For example, a compound interest calculation can be made by calculating the upper bound $(\hat{x}\hat{y}\hat{z}\ldots)$ and the lower bound $(\hat{\hat{x}}\hat{\hat{y}}\hat{\hat{z}}\ldots)$ and finally comparing the rounded values. The calculation can be broken into as many rounding stages as are mandated by law—these can range from rounding after every stage, to rounding at the end.

For multiplication, $\hat{\sigma}$ can be alternatively obtained by a single multiplication on $\hat{\hat{\sigma}}$ by an appropriate precomputed scaling factor, reducing two additions and a multiplication to a single multiplication, eliminating the need for (6.3) and (6.4). This may reduce our times somewhat, depending on pipeline structure. Similarly for additions, the upper bound can be obtained from the lower bound by adding an appropriate precomputed factor.

7 Conclusions

If financial software neglect the effects of IEEE 754 finite precision binary arithmetic, then it can result in legally unacceptable monetary losses. This paper quantifies these losses using a matrix approach the CTEM. Using the CTEM, we showed that even the use of double precision can be exploited to create arbitrary error sequences, with considerable possibilities of financial arbitrage. Initial evidence testifies to the fact that these sequences can be chosen so as

to pass many common statistical tests, and thus avoid detection. Thus, we show that using binary arithmetic directly for financial calculations is inappropriate, as errors can build up without bound, and as we have found statistically undetectable sequences in our experiments. However, with a slightly modified use of the same binary arithmetic, plus access to a decimal software library, we can duplicate the results (for financial and similar applications) of hardware decimal arithmetic, without using decimal hardware instructions, at speeds comparable to early decimal implementations, and within a factor of three of direct hardware supported implementations [26]. Our methods are generally applicable to doing floating-point arithmetic in one base, given an implementation in another base, with sufficient excess precision.

Our results are based on cycle counts, although the actual runtimes are expected to differ, nevertheless our results are indicative of the promise of our approach. Our results show that the runtimes are fast using our approach, as long as system is not deliberately gamed which is unlikely as it would be caught.

If databases store binary approximations for decimal numbers rather than using strings, our analysis gives insight into the approximations and possible dangers of blindly ignoring the problem.

In the future work, we plan to improve our bounds, and report results with our methods, on actual banking applications (including the benchmarks in [19]), working with binary hardware, using real transaction logs from banks.

Acknowledgments

The authors would like to thank the anonymous referees for their comments, which have substantially improved this paper. They are also grateful to Professor S. Sadagopan, Director, IIIT-B and Subrahmanya S.V., Infosys Limited, for their support.

References

[1] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *Computing Surveys,* vol. 23, pp. 5-48, 1991.
[2] N.J. Higham, "The Accuracy of Floating Point Summation," *SIAM J. Scientific Computing,* vol. 14, pp. 783-799, July 1993.
[3] M.F. Cowlishaw, "General Decimal Arithmetic," http://speleotrove.com/decimal/, 2012.
[4] M.F. Cowlishaw, "Decimal Floating-Point: Algorism for Computers," *Proc. IEEE 16th Symp. Computer Arithmetic,* 2003.
[5] P.M. Cohen, "Reflections on Early Computers," http://www.paulcweb.com/reflect/Chap04.html, 2012.
[6] J. von Neumann, "First Draft of a Report on the EDVAC," *IEEE Annals of the History of Computing,* vol. 15, no. 4, pp. 27-75, Oct. 1993.
[7] W. Kahan, *IEEE Standard 754 for Binary Floating-Point Arithmetic,"* Standards Committee of the IEEE CS, 1996.
[8] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C,* second ed. Cambridge Univ. Press, 1992.
[9] "The ASTRÉE Static Analyzer,"http://www.astree.ens.fr/, 2012.
[10] D. Knuth, "Chapter 4 - Arithmetic," *The Art of Computer Programming: Seminumerical Algorithms,* third ed. Addison-Wesley, 1997.
[11] W. Kahan, "Floating-Point Arithmetic Besieged by Business Decisions," *Proc. IEEE 17th Symp. Computer Arithmetic,* 2010.
[12] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehle, and S. Torres, *Handbook of Floating-Point Arithmetic.* Springer, 2010.
[13] "Online Binary-Decimal Converter," http://www.binaryconvert.com/index.html, 2012.
[14] M.J. Anderson, C. Tsen, L.-K. Wang, K. Compton, and M.J Schulte, "Performance Analysis of Decimal Floating-Point Libraries and Its impact on Decimal Hardware and Software Solutions," *Proc. IEEE Int'l Conf. Computer Design,* pp. 465-471, Oct. 2009.
[15] *Intel, Intel 64 and IA-32 Architectures Software Developers Manual,* vol. 1: Basic Architecture, 2009.
[16] A. Fog, "Instruction Tables," http://www.agner.org/optimize/instruction_tables.pdf, 2012.
[17] E. Uren, R. Howard, and T. Perinotti, *Software Internationalization and Localization: An Introduction.* John Wiley & Sons, 1993.
[18] T. Benchmark, http://speleotrove.com/decimal/telcoSpec.html, 2012.
[19] Decimal Floating-Point (DFP) Benchmarks, http://mesa.ece.wisc.edu/display_project.php?projectid=10, 2012.
[20] D.M. Gay, *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions. Numerical Analysis Manuscript 90-10,* Murray Hill, NJ: AT&T Bell Laboratories, 1990.
[21] M. Cornea, J. Harrison, C. Anderson, P. Tang, E. Schneider, and E. Gvozdev, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," *IEEE Trans. Computers,* vol. 58, no. 2, pp. 148-162, Feb. 2009.
[22] L.-K. Wang, C. Tsen, M.J. Schulte, and D. Jhalani, "Benchmarks and Performance Analysis of Decimal Floating-Point Applications," *Proc. IEEE 25th Int'l Conf. Computer Design,* pp. 164-170, Oct. 2007.
[23] *ANSI/IEEE Standard for Floating-Point Arithmetic,* IEEE Standard, pp. 754-1985, 1985.
[24] *IEEE Standard for Floating-Point Arithmetic,* IEEE Standard 754-2008, 2008.
[25] The Int'l Version of IEEE 754-2008, ISO/IEC/IEEE 60559:2011, http://www.iso.org/iso/catalogue_detail.htm?csnumber=57469, 2012.
[26] S. Carlough, A. Collura, S. Mueller, and M. Kroener, "The IBM zEnterprise-196 Decimal Floating-Point Accelerator," *Proc. IEEE 20th Symp. Computer Arithmetic,* 2011.

Abhilasha Aswal received the MTech degree from the International Institute of Information Technology-Bangalore (IIIT-B). She is now working toward the PhD degree from IIIT-B. She is interested in optimization under uncertainty, mathematics of operations research, and intellectual property rights.

M. Ganesh Perumal received the MS degree from the International Institute of Information Technology-Bangalore (IIIT-B). He is now working toward the PhD degree from IIIT-B. He is interested in theory of algorithms, mathematics of operations research, simulation, and modeling and complexity theory.

G.N. Srinivasa Prasanna received the BTech degree at IIT Kanpur, and the MS and PhD degrees at MIT, Cambridge. He is a professor at International Institute of Information Technology-Bangalore since 2004, and was previously at Lucent Microelectronics and Lucent Bell Laboratories, for about 11 years. He is interested broadly in the areas of algorithms and robotics. Major focus areas include robust optimization under uncertainty, with applications to supply chains, real time search, banking, gaming, and allied areas. He is a senior member of the IEEE.