

$$g^i = \begin{cases} b^i & \text{for } i = 1 \\ b^i f^{i-1} & \text{for } 2 \leq i \leq k-1 \\ b^i f^{i-1} / f^{i-k} & \text{for } k \leq i \leq n-1 \end{cases}$$

$$h^i = \begin{cases} d^i & \text{for } i = 1 \\ d^i f^{i-1} - c^i h^{i-1} & \text{for } 2 \leq i \leq k-1 \\ (d^i f^{i-1} - c^i h^{i-1}) / f^{i-k} & \text{for } k \leq i \leq n. \end{cases}$$

Note that while this third recurrence involves division, there are  $k$  iterative steps between when each divisor is generated and when it is used. Hence, postnormalization of the  $f^i$ 's becomes more realistic. Note that, by selecting  $k$  large enough, sufficient time can be given to normalize each  $f^i$ , without slowing down the network as a whole.

However, before this approach can be used, several concerns would have to be investigated. One concern is whether the computation is stable in the classical sense; that is, if normalized arithmetic were used. Observe that for  $k = 1$ , the third recurrence degrades into the first recurrence proposed for reducing the matrix equation of Fig. 2. Furthermore, this first method is known to be stable under a reasonable set of conditions. Also, error and stability analysis would have to be repeated assuming now that unnormalized arithmetic is used. In the case of matrices which result from continuous problems, there appears to be no loss in stability or accuracy. However, an exact analysis remains.

#### CONCLUSION

We have shown that the digit serial property of digit on-line arithmetic imposes unique limitations on any arithmetic unit which performs certain operations in a digit on-line manner. These limitations are inherent in the sense that any fully digit on-line arithmetic unit which performs these operations will have some type of similar limitations. We gave several techniques which either try to avoid these limitations and reduce their impact. These techniques can be applied to a relatively wide range of relevant numerical problems. Furthermore, there is a strong belief to believe that these techniques apply to even a much wider range of problems.

#### REFERENCES

- [1] D. E. Atkins, "Introduction to the role of redundancy in computer arithmetic," *Computer*, vol. 8, pp. 74-76, June 1975.
- [2] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Trans. Electron. Comput.*, pp. 389, 1961.
- [3] M. D. Ercegovac, "An on-line square rooting algorithm," in *Proc. 4th Symp. Comput. Arith.*, Santa Monica, CA, Oct. 1978.
- [4] M. D. Ercegovac, and A. L. Grnarov, "On the performance of on-line arithmetic," in *Proc. 1980 Int. Conf. Parallel Process.*, Aug. 1980, pp. 55-62.
- [5] A. Gorji-Sinaki, and M. D. Ercegovac, "Design of a digit-slice on-line arithmetic unit," in *Proc. 5th Symp. Comput. Arith.*, Ann Arbor, MI, May 1981, pp. 72-80.
- [6] D. Heller, private communication.
- [7] M. J. Irwin, "An arithmetic unit for on-line computation," Ph.D. dissertation, Dep. Comput. Sci., Univ. Illinois, Urbana, Rep. UIUCDCS-R-77-873, May 1977.
- [8] —, "A pipelined processing unit for on-line division," in *Proc. 5th Annu. Symp. Comput. Arch.*, Palo Alto, CA, Apr. 1978.
- [9] J. M. Ortega, *Computer Science and Applied Mathematics*. New York: Academic, 1972.
- [10] R. M. Owens, and M. J. Irwin, "On-line algorithms for the design of pipeline architectures," in *Proc. Annu. Symp. Comput. Arch.*, Philadelphia, PA, Apr. 1979, pp. 12-19.
- [11] R. M. Owens, "Digit on-line algorithms for pipeline architectures," Ph.D. dissertation, Dep. Comput. Sci., The Pennsylvania State University, University Park, PA, Rep. CS80-21, Aug. 1980.
- [12] —, "Compound algorithms for digit online arithmetic," in *Proc. 5th Symp. Comput. Arith.*, Ann Arbor, MI, May 1981, pp. 64-71.
- [13] —, "Error analysis of unnormalized arithmetic," Dep. Comput.

Sci., The Pennsylvania State University, University Park, PA, Rep. CS81-16, Aug. 1981.

- [14] K. S. Trivedi and M. D. Ercegovac, "On-line algorithms for division and multiplication," *IEEE Trans. Comput.*, vol. C-26, pp. 681-687, July 1977.
- [15] K. S. Trivedi, and J. G. Rusnak, "Higher radix on-line division," in *Proc. 4th Symp. Comput. Arith.*, Santa Monica, CA, Oct. 1978.
- [16] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1963.

#### The Use of Floating-Point and Interval Arithmetic in the Computation of Error Bounds

DANIEL W. LOZIER

**Abstract**—Three forms of interval floating-point arithmetic are defined in terms of absolute precision, relative precision, and combined absolute and relative precision. The absolute-precision form corresponds to the centered form of conventional rounded-interval arithmetic. The three forms are compared on the basis of the number of floating-point operations needed to generate error bounds for inner-product accumulation.

**Index Terms**—Arithmetic algorithms, error propagation, floating-point computation, inner-product accumulation, interval analysis, interval arithmetic, relative precision, rounding error analysis.

#### I. INTRODUCTION

A numerical procedure consists of a finite succession of arithmetic operations. The output of a procedure is only approximate because of three types of errors: *inherent errors*, which are due to uncertainties in the input data; *abbreviation errors*, i.e., rounding or chopping errors, which are committed when floating-point operations are performed in a computer; and *truncation errors*, which arise when a numerical procedure is used to approximate, for example, an infinite series by a polynomial. The purpose of *interval analysis* [3], [4] is to provide a means of generating bounds on the errors in the output of numerical procedures.

The fundamental idea is to represent each number with associated error bound as an interval. Of course, numbers that are known exactly have an error bound of zero. They can be regarded as intervals of width zero. Usually intervals are represented internally either by storing their upper and lower endpoints or by storing their midpoints and halfwidths (the *centered form*). Intervals containing the input data for a procedure are stored, with care being taken to ensure that the stored intervals are not invalidated by roundoff. As computation proceeds, containing intervals for the results of all arithmetic operations are produced. Strict containment is maintained at every step by using special (directed) rounding in the computer arithmetic. We will call the process just described *interval arithmetic*. Neglecting truncation errors and assuming the process does not fail because of underflow, overflow, or division by an interval containing zero, interval arithmetic produces results complete with error bounds.

Interval arithmetic is directly applicable to rational expressions, i.e., finite combinations of interval variables and interval constants with interval-arithmetic operations [3, Theorem 3.1]. The execution of a numerical procedure results in the evaluation of rational expressions. Interval arithmetic overestimates the actual error in such an evaluation because it bounds the worst possible rounding error at each arithmetic operation and, even if rounding errors are ignored by assuming infinite-precision arithmetic, it does not generally compute the exact set-theoretic range of values of an interval rational expression [3, Section 3.1]. In fact, the overestimation is sometimes excessive and is unusually severe in such matrix computations as the

Manuscript received April 27, 1982; revised November 24, 1982.

The author is with the U.S. Department of Commerce, National Bureau of Standards, Washington, DC 20234.

numerical solution of linear systems by Gaussian elimination [3], [5]. Recent modifications of interval analysis due to F. W. J. Olver [6], [7] have been applied by Olver and Wilkinson [8] to the Gaussian elimination problem. They construct an error-bound formula that does not assume any special properties of the linear system and that essentially agrees with error estimates derived by linear perturbation theory. The bounds are of the same type as those obtained by interval arithmetic, but are vastly more realistic.

The modified form of interval analysis is called *ap-rp error analysis* because it is based on special definitions of *absolute precision* and *relative precision*. The thrust of ap-rp error analysis is to construct analytical error-bound formulas for whole numerical procedures (or subprocedures), for example the evaluation of a polynomial or the solution of a set of linear algebraic equations, and then to evaluate the formulae in floating-point arithmetic without losing the strictness of the error bounds. It is shown in [7] how to evaluate error-bound formulas without using directed rounding.

Interval arithmetic updates the error bound at each arithmetic operation. Consequently, it is suitable for use as a form of computer arithmetic<sup>1</sup> from a language like Fortran. Portable software demonstrating this fact is available [9]. This software uses a Fortran precompiler [2] to make interval arithmetic available for direct use in Fortran programs. Numerical procedures for exponentiation and elementary mathematical functions are included, for which the problem of excessive overestimation of the error is alleviated by abandoning straightforward interval arithmetic and evaluating instead an overall error-bound formula for each procedure. However, the author of [9] leaves open the possibility that the error bounds for mathematical functions may not be completely rigorous. The problem of actually achieving rigorous bounds using existing programming languages and computer arithmetics is difficult; furthermore, satisfactory general algorithms for mathematical functions are not readily available. For an example of a proposed general algorithm for the exponential function, see [1].

Although no purely arithmetic approach can always succeed in producing realistic error bounds, the difficulty of deriving error-bound formulae through analysis and then programming the formulae deters us from abandoning the idea of interval arithmetic. Furthermore, an interval form of representation in which both the data value and the error bound are associated with a single program variable name is convenient for communicating interval data between subroutines and the programs that call them.

In this paper we introduce interval arithmetic in three forms suggested by ap-rp error analysis. The ap form corresponds to the centered form of conventional rounded interval arithmetic. We compare the ap form to the rp form by counting the number of floating-point operations needed to compute a strict error bound for individual floating-point addition, subtraction, multiplication and division operations. Finally, we compare the ap and rp forms to an ap-rp form of interval arithmetic by counting the number of floating-point operations needed to compute a strict error bound for inner-product accumulation.

## II. OLVER'S APPROACH TO ERROR ANALYSIS

The following definitions are found in [6].

**Definition 1:** Let  $\bar{\alpha}$  be a constant such that  $\bar{\alpha} \geq 0$ . If two real numbers,  $x$  and  $\bar{x}$ , satisfy  $|x - \bar{x}| \leq \bar{\alpha}$  then we say that  $x$  and  $\bar{x}$  *approximate each other to absolute precision  $\bar{\alpha}$* , and we write

$$x \simeq \bar{x}; \text{ap}(\bar{\alpha}). \quad (2.1)$$

In this definition,  $\bar{\alpha}$  is an *upper bound* on the absolute difference  $|x - \bar{x}|$ .

**Definition 2:** Let  $\bar{\delta}$  be a constant such that  $\bar{\delta} \geq 0$ . If two nonzero real numbers,  $x$  and  $\bar{x}$ , have the same sign and satisfy  $|x|e^{-\bar{\delta}} \leq |\bar{x}|$

TABLE I  
RULES OF ERROR ARITHMETIC

Approx.	ap	rp	Altern. rp
$x + y \simeq \bar{x} + \bar{y}$	$\bar{\alpha} + \bar{\beta}$	$\ln(\frac{\bar{x}e^{\bar{\delta}} + \bar{y}e^{\bar{\epsilon}}}{\bar{x} + \bar{y}})$ <sup>(1)</sup>	$\max(\bar{\delta}, \bar{\epsilon})$ <sup>(1)</sup>
$x - y \simeq \bar{x} - \bar{y}$	$\bar{\alpha} + \bar{\beta}$	$\ln(\frac{\bar{x} - \bar{y}}{\bar{x}e^{-\bar{\delta}} - \bar{y}e^{\bar{\epsilon}}})$ <sup>(1,2)</sup>	
$xy \simeq \bar{x}\bar{y}$	$ \bar{x} \bar{\beta} +  \bar{y} \bar{\alpha} + \bar{\alpha}\bar{\beta}$	$\bar{\delta} + \bar{\epsilon}$	
$x/y \simeq \bar{x}/\bar{y}$	$\frac{ \bar{x} \bar{\beta} +  \bar{y} \bar{\alpha}}{ \bar{y} ( \bar{y}  - \bar{\beta})}$ <sup>(3)</sup>	$\bar{\delta} + \bar{\epsilon}$	

(1) Provided  $\bar{x}$  and  $\bar{y}$  have the same sign.

(2) Provided  $|\bar{x}|e^{-\bar{\delta}} > |\bar{y}|e^{\bar{\epsilon}}$ .

(3) Provided  $|\bar{y}| > \bar{\beta}$ .

$\leq |x|e^{\bar{\delta}}$  then we say that  $x$  and  $\bar{x}$  *approximate each other to relative precision  $\bar{\delta}$* , and we write

$$x \simeq \bar{x}; \text{rp}(\bar{\delta}). \quad (2.2)$$

In this definition, when  $\bar{\delta}$  is small, expansion of  $e^{-\bar{\delta}}$  and  $e^{\bar{\delta}}$  relates  $\bar{\delta}$  to the relative difference  $\rho = (\bar{x} - x)/x$ . The expansions show that  $|\rho| \leq \bar{\delta} + 0(\bar{\delta}^2)$ , i.e., the absolute value of the relative difference is essentially bounded by  $\bar{\delta}$ .

The unusual characterization of relative differences is preferred because it leads to desirable elementary properties. Elementary properties satisfied by ap and rp error bounds are the *symmetry property*, the *inclusion property*, and the *successive approximation property*. The first of these refers to the fact that the roles of  $x$  and  $\bar{x}$  are reversible. The second is the property that we may replace  $\bar{\alpha}$  by any larger value in (2.1) and  $\bar{\delta}$  by any larger value in (2.2) without nullifying the truth of the relations. The third is the fact that if in addition to (2.1) we have  $\bar{x} \simeq \bar{\bar{x}}; \text{ap}(\bar{\beta})$ , then  $x \simeq \bar{\bar{x}}; \text{ap}(\bar{\alpha} + \bar{\beta})$ ; and if in addition to (2.2) we have  $\bar{x} \simeq \bar{\bar{x}}; \text{rp}(\bar{\epsilon})$ , then  $x \simeq \bar{\bar{x}}; \text{rp}(\bar{\delta} + \bar{\epsilon})$ .

Table I summarizes the basic rules of error arithmetic. In addition to (2.1) and (2.2) we assume that  $y \simeq \bar{y}; \text{ap}(\bar{\beta})$  and  $y \simeq \bar{y}; \text{rp}(\bar{\epsilon})$ . The table gives ap and rp error bounds for the sum, difference, product and quotient of two approximate relations of the same type. The footnotes give sufficient conditions for the validity of the error bounds. Because in ap-rp error analysis we are concerned with "worst-case" rather than "average-case" error analysis, the formulas shown are strict upper bounds on the error in the result due to errors in operands that are within the assumed bounds. Furthermore, the bounds are attainable.

Conversion between the two forms of error bound is possible. If we assume (2.1) is true and  $\bar{\alpha} < |\bar{x}|$ , then we have

$$x \simeq \bar{x}; \text{rp}\{-\ln(1 - \bar{\alpha}/|\bar{x}|)\}. \quad (2.3)$$

Similarly, if we assume (2.2) is true, then we have

$$x \simeq \bar{x}; \text{ap}\{|\bar{x}|(e^{\bar{\delta}} - 1)\}. \quad (2.4)$$

These results follow from the definitions [6, Sec. 3.4]. Table I suggests no clear choice between an ap system and an rp system for bounding errors in the evaluation of rational expressions. The conversion formulas make possible a combined ap-rp system in which

<sup>1</sup> We are using the term *computer arithmetic* here and elsewhere to mean a type of arithmetic supported by a programming language, regardless of whether the arithmetic is realized in hardware, firmware or software.

some error bounds are represented in ap form and some in rp form. The construction and preliminary evaluation of such a system is the subject of this paper.

The arithmetic operations in Table I and (2.3) and (2.4) are exact, not floating-point. The additional effect of abbreviation (i.e., rounding or chopping) errors in computer arithmetic still has to be considered. Henceforth we will adopt the convention that a bar or double bar over a symbol for a real number always means a stored approximation to the number. A real number denoted by an unbarred symbol may or may not be exactly storable. In the case of (2.1) and (2.2), by our convention,  $\bar{x}$  and  $\overline{\bar{x}}$  are stored to represent the interval  $\{x | x \simeq \bar{x}; \text{ap}(\bar{x})\}$  and  $\bar{x}$  and  $\delta$  are stored to represent the interval  $\{x | x \simeq \bar{x}; \text{rp}(\delta)\}$ .

There is no requirement that the error bound should be stored in the same floating-point system as  $\bar{x}$ . Indeed, such a requirement often would be wasteful of storage and processor time, for example when the main computation is being done to double or higher precision. Consequently, we assume two modes of floating-point arithmetic are available, designated  $\mathcal{M}$  (for *main mode*) and  $\mathcal{L}$  (for *lower mode*). Our assumptions about floating-point arithmetic are as follows:

i)  $\mathcal{M}$  and  $\mathcal{L}$  have the same radix,  $r$  say, so in considering internal operations only we avoid the necessity of considering radix conversion processes.

ii) There exists a positive constant  $\bar{\gamma} \in \mathcal{L}$  with the following property: for all  $\bar{x}, \bar{y} \in \mathcal{M}$  and  $o \in \{+, -, \times, /\}$  such that the computer arithmetic approximates the exact result  $\bar{x} o \bar{y}$  by an element  $x o y$  in  $\mathcal{M}$ , the approximating relation

$$\bar{x} o \bar{y} \simeq \overline{x o y}; \text{rp}(\bar{\gamma}) \quad (2.5a)$$

holds true. Similarly, there exists a positive constant  $\bar{\gamma}_l \in \mathcal{L}$  that plays the same role for  $\mathcal{L}$ , i.e.,

$$\overline{\alpha o \beta} \simeq (\overline{\alpha o \beta}) \text{rp}(\bar{\gamma}_l) \quad (2.5b)$$

whenever  $\bar{\alpha}, \bar{\beta} \in \mathcal{L}$  and the computer arithmetic produces  $\overline{\alpha o \beta} \in \mathcal{L}$ .

It is shown in [6] that

$$\bar{\gamma} = \begin{cases} r^{1-d} & \text{if chopping is used} \\ (\frac{1}{2}r)^{1-d} & \text{if rounding is used} \end{cases}$$

where  $d$  is the number of digits carried in the fractional part (or mantissa); similarly for  $\bar{\gamma}_l$ .

iii) Whenever a number in  $\mathcal{M}$  is abbreviated to fit into  $\mathcal{L}$ , the rp error for this process is again bounded by  $\bar{\gamma}_l$ . Furthermore, we will always assume

$$\bar{\gamma} \leq \bar{\gamma}_l \leq \frac{1}{16}. \quad (2.6)$$

iv) Any occurrence of underflow, overflow or division by zero prevents further normal processing, because the relationships (2.5) would be invalidated.

v) Finally, we assume that  $\bar{\gamma}_l$ ,  $\bar{\gamma}$  and small positive integers (up to 25 or so) are members of  $\mathcal{L}$ .

We state two results from [7] that are needed to derive error bounds in this paper. The first is the *exponential rule*, which states that "each time two nonnegative members of  $\mathcal{L}$  are added, multiplied or divided, an upper bound for the true result is given by the product of the stored result and  $e^{\bar{\gamma}_l}$ , and a lower bound for the true result is given by the product of the stored result and  $e^{-\bar{\gamma}_l}$ ... the rule also applies when nonnegative members of  $\mathcal{M}$  are abbreviated to  $\mathcal{L}$ ." The exponential rule is a simple consequence of Definition 2. The second result, restricted slightly for use in this paper, is as follows.

**Lemma (Olver):** Assume  $\bar{\gamma}_l \leq 1/16$  and  $i$  is an integer in the range  $0 \leq i \leq 7$ . Let  $\sigma, (2p_1, 2p_2, \dots, 2p_i), \tau_1, \tau_2, \dots, \tau_i$  be non-negative members of  $\mathcal{L}$ , and let  $p$  be any real number such that  $p + i + 3 \geq 0$ . Then

$$\frac{\sigma e^{p\tau_1 + p\tau_2 + \dots + p\tau_i + p\bar{\gamma}_l}}{\leq \sigma(1 + (2p_1\tau_1 + 2p_2\tau_2 + \dots + 2p_i\tau_i) + q\gamma_l)} \quad (2.7)$$

and

$$\frac{\sigma e^{-p\tau_1 - p\tau_2 - \dots - p\tau_i - p\bar{\gamma}_l}}{\geq \sigma/(1 + 2p_1\tau_1 + 2p_2\tau_2 + \dots + 2p_i\tau_i + q\gamma_l)} \quad (2.8)$$

where  $q$  is any member of  $\mathcal{L}$  such that

$$q \geq 2p + 2i + 6, \quad (2.9)$$

provided that

$$1 + 2p_1\tau_1 + 2p_2\tau_2 + \dots + 2p_i\tau_i + q\gamma_l \leq 2. \quad (2.10)$$

Olver's lemma states a procedure for arriving at a computable upper bound [the right side of (2.7)] or a computable lower bound [the right side of (2.8)] from the analytic form given by the left side of either (2.7) or (2.8). The condition (2.10) is designed for automatic checking in a computer. The factor  $1 + 2p_1\tau_1 + 2p_2\tau_2 + \dots + 2p_i\tau_i + q\gamma_l$  called a *compensating factor* in (2.7) and a *compensating divisor* in (2.8). It is to be computed using floating-point operations in  $\mathcal{L}$  by multiplying out each of the products ( $i+1$  multiplications since Olver's lemma assumes  $2p_1, 2p_2, \dots, 2p_i$  are stored in  $\mathcal{L}$ ) and accumulating the sum. This procedure produces rigorous error bounds regardless of the order of summation. Compensating factors and divisors are the mechanisms by which the need to assume directed rounding in the computer arithmetic is avoided in ap-rp error analysis.

### III. ERROR BOUNDS FOR INDIVIDUAL FLOATING-POINT OPERATIONS

In this section we present error bounds for individual conversion and arithmetic operations, where each operand consists of a *principal value* taken from the main mode  $\mathcal{M}$  and an associated *error bound* taken from the lower mode  $\mathcal{L}$ . In the case of conversion operations, the principal value is not changed and an error bound of opposite type is produced. In the case of arithmetic operations, both operands are given with either ap bounds or rp bounds and an error bound of the same or opposite type is produced for the principal value of the result. The principal value of the result of each arithmetic operation is a number in  $\mathcal{M}$  which is produced by a main-mode floating-point operation. Thus, if  $\bar{x}$  and  $\bar{y}$  are members of  $\mathcal{M}$ , we have for each  $o \in \{+, -, \times, /\}$

$$\bar{x} o \bar{y} \simeq \overline{x o y}; \text{rp}(\bar{\gamma}) \quad (3.1)$$

where  $\overline{x o y} \in \mathcal{M}$  and  $\bar{\gamma}$  is the rp-error bound associated with arithmetic operations in  $\mathcal{M}$ ; compare (2.5a). If  $x$  and  $y$  are any two real numbers lying within the intervals defined by the operands, our task is to produce formulas bounding the ap and rp errors incurred in approximating  $x o y$  by  $\overline{x o y}$ . Naturally, the formulas will use as data  $\bar{x}, \bar{y}, \overline{x o y}, \bar{\gamma}, \bar{\gamma}_l$  and the given error bounds for  $\bar{x}$  and  $\bar{y}$ .

We present the error-bound formulas for each operation in two forms, a *precomputable form* and a *computable form*. The precomputable form is in each case a product  $\lambda e^\phi$  where  $\lambda$  is the linear approximation obtained by neglecting higher order terms in the error bounds given for arithmetic operations in Table I and conversion operations in (2.3) and (2.4), and  $e^\phi$  is a factor exceeding unity which is sufficient to make  $\lambda e^\phi$  a rigorous error bound. The computable form is obtained from the precomputable form through use of Olver's lemma. It is the stored value of the product  $\lambda$  times  $\bar{x}$ , denoted  $\lambda\bar{x}$ , where  $\lambda$  and  $\bar{x}$  are the stored values in  $\mathcal{L}$  of  $\lambda$  and the compensating factor  $\bar{x}$ . Of necessity the computable form exceeds or equals the precomputable form, but in the multivariate Taylor-series expansions in powers of the errors the linear terms agree. The precomputable form is suitable for constructing interval-arithmetic error bounds for compound arithmetic expressions analytically, either by hand or, possibly, by a symbol manipulator. The computable form could be used in a numerical implementation of interval arithmetic.

#### Conversion Operations

These operations refer to changing to an ap form of error bound from an rp form, or vice versa, leaving the principal value in  $\mathcal{M}$  unchanged. Sharp bounds for these operations are given in (2.3) and (2.4). Precomputable and computable forms can be found in [7], and

TABLE II  
ERROR BOUNDS FOR CONVERSION OPERATIONS.  $\bar{x}$  DENOTES THE  
INPUT (AND OUTPUT) PRINCIPAL VALUE

input error bound	linear approx. $\lambda$	output error bounds	exponent $\phi$ and compensating factor $\chi$
$rp(\bar{\delta})$	$\bar{\delta}/ \bar{x} $	$ap(\lambda e^{\phi})$	$\bar{\delta}$
		$ap(\lambda\chi)$ (1)	$1 + 2\bar{\delta} + 12\bar{\gamma}_g$
$ap(\bar{\alpha})$	$\bar{\alpha}/ \bar{x} $	$rp(\lambda e^{\phi})$ (2)	$\lambda$
		$rp(\lambda\chi)$ (1)	$1 + 3\lambda + 12\bar{\gamma}_g$

(1) Provided  $\bar{\chi} < 2$ .

(2) Provided  $\lambda < t_2$ .

we summarize these results in Table II. Reading from left to right, the first row refers to a principal value  $\bar{x} \in \mathcal{M}$  with an associated  $rp$ -error bound  $\bar{\delta} \in \mathcal{L}$ . The linear approximation  $\bar{\delta}/|\bar{x}|$  of the corresponding  $ap$  bound, obtained by linearizing (2.4), appears in the second column. The third and fourth columns have two entries each, the upper entries defining the precomputable form  $\lambda e^{\phi}$  and the lower entries defining the computable form  $\lambda\chi$ . It is understood that  $\lambda\chi$  is the member of  $\mathcal{L}$  obtained by multiplying  $\lambda$  by  $\bar{\chi}$ , where  $\lambda$  and  $\bar{\chi}$  are evaluated from the displayed formulas for  $\lambda$  and  $\chi$ . In the evaluation of  $\lambda$ ,  $\bar{x}$  is abbreviated to the lower mode before multiplying. All arithmetic operations used in producing  $\lambda\chi$  take place in  $\mathcal{L}$ .

Column 3 also refers to footnotes which give sufficient conditions for the error bounds to be valid. The sufficient conditions for computable bounds, in all tables, include the condition that the stored value of the compensating factor  $X$  is less than two; compare (2.10). The quantity  $t_2$  in footnote (2) of Table II arises in the analysis of (2.3) when the inequality

$$-ln(1-t) \leq te^t, 0 \leq t \leq t_2 \quad (3.2)$$

is employed to arrive at the precomputable form. Its value,

$$t_2 = 0.8803 \dots, \quad (3.3)$$

is a root of the equation  $-ln(1-t) = te^t$ . Conditions involving  $t_2$  arise also in precomputable error bounds for the arithmetic operations; see below. However, in every case these conditions are implied by the sufficient conditions for the computable error bounds, so no reference to  $t_2$  is needed in applying the computable bounds.

#### Arithmetic Operations

Again we present the error bounds in tabular form. Table III gives error bounds for addition and subtraction. Table IV does the same for multiplication and division. The tables are analogous to the previous table but because there are two operands instead of one, each presents four cases instead of two. The same conventions apply for reading the tables. *The grouping of operations implied by the parentheses in the linear factor  $\lambda$  is important to observe in the tables when evaluating the computable error bounds; the grouping was chosen to reduce the compensating factor, and if not observed then the computed error bound may not be valid.* The grouping of terms is irrelevant for the precomputable forms.

Because the method of derivation of the bounds is similar in all cases, we only present one case in detail here. The case presented is line 2 of Table III because the previous line is merely a restatement of equations (6.8) and (6.13) in [7]. Suppose  $x$  and  $y$  are any real numbers such that  $x \simeq \bar{x}$ ;  $ap(\bar{\alpha})$  and  $y \simeq \bar{y}$ ;  $ap(\bar{\beta})$ . Reading the precomputable error bound from the first line of Table III, we have

$$x + y \simeq \overline{x + y}; ap\{(\bar{\alpha} + \bar{\beta} + \bar{\gamma}|x + y|)e^{\bar{\gamma}}\}.$$

Applying the precomputable bound from the second line of Table II, we find

TABLE III  
ERROR BOUNDS FOR INDIVIDUAL ADDITION AND SUBTRACTION  
OPERATIONS.  $\bar{x}$  AND  $\bar{y}$  DENOTE THE PRINCIPAL VALUES OF THE  
INPUT OPERANDS,  $\bar{x} + \bar{y}$  DENOTES THE PRINCIPAL VALUE OF THE  
SUM OR DIFFERENCE

first input error bound	second input error bound	linear approx. $\lambda$	output error bounds	exponent $\phi$ and compensating factor $\chi$
$ap(\bar{\alpha})$	$ap(\bar{\beta})$	$(\bar{\alpha} + \bar{\beta}) + \bar{\gamma} \bar{x} + \bar{y} $	$ap(\lambda e^{\phi})$	$\bar{\gamma}$
			$ap(\lambda\chi)$ (1)	$1 + 14\bar{\gamma}_g$
$ap(\bar{\alpha})$	$ap(\bar{\beta})$	$\bar{\gamma} + (\bar{\alpha} + \bar{\beta})/ \bar{x} + \bar{y} $	$rp(\lambda e^{\phi})$ (2)	$\bar{\gamma} + \lambda e^{\bar{\gamma}}$
			$rp(\lambda\chi)$ (1)	$1 + 3\lambda + 18\bar{\gamma}_g$
$rp(\bar{\delta})$	$rp(\bar{\epsilon})$	$\bar{\gamma} \bar{x} + \bar{y}  + \bar{\delta} \bar{x}  + \bar{\epsilon} \bar{y} $	$ap(\lambda e^{\phi})$	$\max(\bar{\gamma}, \bar{\delta}, \bar{\epsilon})$
			$ap(\lambda\chi)$ (1)	$1 + 2\max(\bar{\delta}, \bar{\epsilon}) + 18\bar{\gamma}_g$
$rp(\bar{\delta})$	$rp(\bar{\epsilon})$	$\bar{\gamma} + (\bar{\delta} \bar{x}  + \bar{\epsilon} \bar{y} )/ \bar{x} + \bar{y} $	$rp(\lambda e^{\phi})$ (3)	$\max(\bar{\gamma}, \bar{\delta}, \bar{\epsilon}) + \lambda e^{\max(\bar{\gamma}, \bar{\delta}, \bar{\epsilon})}$
			$rp(\lambda\chi)$ (4)	$1 + 8\bar{\gamma} + \bar{\delta} + \bar{\epsilon} + 20\bar{\gamma}_g$
			$rp(\lambda\chi)$ (5)	$1 + 8\lambda + 20\bar{\gamma}_g$

(1) Provided  $\bar{\chi} < 2$ .

(2) Provided  $\lambda e^{\bar{\gamma}} < t_2$ .

(3) Provided  $\lambda e^{\max(\bar{\gamma}, \bar{\delta}, \bar{\epsilon})} < t_2$ .

(4) Provided  $\bar{\chi} < 2$  and  $\text{sgn}(\bar{x}) = \text{sgn}(\bar{y})$ .

(5) Provided  $\bar{\chi} < 2$  and  $\text{sgn}(\bar{x}) \neq \text{sgn}(\bar{y})$ .

TABLE IV  
ERROR BOUNDS FOR INDIVIDUAL MULTIPLICATION AND DIVISION  
OPERATIONS.  $\bar{x}$  AND  $\bar{y}$  DENOTE THE PRINCIPAL VALUES OF THE  
INPUT OPERANDS,  $\bar{x}\bar{y}$  DENOTES THE PRINCIPAL VALUE OF THE  
PRODUCT OR QUOTIENT

first input error bound	second input error bound	linear approx. $\lambda$	output error bounds	exponent $\phi$ and compensating factor $\chi$
$rp(\bar{\delta})$	$rp(\bar{\epsilon})$	$\bar{\gamma} + \bar{\delta} + \bar{\epsilon}$	$rp(\lambda e^{\phi})$	0
			$rp(\lambda\chi)$ (1)	$1 + 10\bar{\gamma}_g$
$rp(\bar{\delta})$	$rp(\bar{\epsilon})$	$(\bar{\gamma} + \bar{\delta} + \bar{\epsilon}) \bar{x}\bar{y} $	$ap(\lambda e^{\phi})$	$\bar{\gamma} + \bar{\delta} + \bar{\epsilon}$
			$ap(\lambda\chi)$ (1)	$1 + 3\bar{\gamma} + \bar{\delta} + \bar{\epsilon} + 16\bar{\gamma}_g$
$ap(\bar{\alpha})$	$ap(\bar{\beta})$	$\bar{\gamma} + \bar{\alpha}/ \bar{x}  + \bar{\beta}/ \bar{y} $	$rp(\lambda e^{\phi})$ (2)	$\max(\bar{\alpha}/ \bar{x} , \bar{\beta}/ \bar{y} )$
			$rp(\lambda\chi)$ (1)	$1 + 3\max(\bar{\alpha}/ \bar{x} , \bar{\beta}/ \bar{y} ) + 16\bar{\gamma}_g$
$ap(\bar{\alpha})$	$ap(\bar{\beta})$	$(\bar{\gamma} + \bar{\alpha}/ \bar{x}  + \bar{\beta}/ \bar{y} ) \bar{x}\bar{y} $	$ap(\lambda e^{\phi})$ (2)	$\max(\bar{\alpha}/ \bar{x} , \bar{\beta}/ \bar{y} ) + (\bar{\gamma} + \bar{\alpha}/ \bar{x}  + \bar{\beta}/ \bar{y} ) \times e^{\max(\bar{\alpha}/ \bar{x} , \bar{\beta}/ \bar{y} )}$
			$ap(\lambda\chi)$ (1)	$1 + 8\bar{\gamma} + \bar{\alpha}/ \bar{x}  + \bar{\beta}/ \bar{y}  + 20\bar{\gamma}_g$

(1) Provided  $\bar{\chi} < 2$ .

(2) Provided  $\max(\bar{\alpha}/|\bar{x}|, \bar{\beta}/|\bar{y}|) < t_2$ .

$$x + y \simeq \overline{x + y}; \text{rp}(\lambda e^\phi)$$

where

$$\lambda = (\overline{\alpha} + \overline{\beta})/|\overline{x + y}| + \overline{\gamma}, \phi = \overline{\gamma} + \lambda e^{\overline{\gamma}}$$

provided that

$$\lambda e^{\overline{\gamma}} \leq t_2. \quad (3.4)$$

This result is recorded as the precomputable form in the second line of Table III. To find the computable form, we start by estimating  $\lambda$  by repeated use of the exponential rule:

$$\begin{aligned} \lambda &\leq \{(\overline{\alpha} + \overline{\beta})/|\overline{x + y}| + \overline{\gamma}\}e^{\overline{\gamma}} \\ &\leq \{\alpha + \beta/|x + y| + \overline{\gamma}\}e^{2\overline{\gamma}} \\ &\leq \{(\alpha + \beta)/|x + y| + \overline{\gamma}\}e^{3\overline{\gamma}} \\ &\leq \overline{\lambda}e^{4\overline{\gamma}}. \end{aligned}$$

Consequently, using (2.6) we have both

$$\phi \leq \overline{\gamma}_l + \overline{\lambda}e^{5\overline{\gamma}_l}$$

and

$$e^{5\overline{\gamma}_l} \leq \frac{3}{2}. \quad (3.5)$$

Then,

$$\lambda e^\phi \leq \overline{\lambda}e^{3/2\overline{\lambda} + 5\overline{\gamma}_l}.$$

Now we apply Olver's lemma with  $i = 1$ ,  $p_1 = \frac{3}{2}$ ,  $\tau_1 = \overline{\lambda}$ ,  $p = 5$  to obtain

$$\lambda e^\phi \leq \overline{\lambda}(1 + 3\overline{\lambda} + 18\overline{\gamma}_l),$$

valid when (3.4) and

$$1 + 3\overline{\lambda} + 18\overline{\gamma}_l \leq 2 \quad (3.6)$$

are satisfied; compare (2.10).

However, (3.4) is implied by the other inequalities. From (3.6) and the exponential rule we find

$$1 + 3\overline{\lambda} + 18\overline{\gamma}_l \leq \overline{\lambda}(1 + 3\overline{\lambda} + 18\overline{\gamma}_l)e^{3\overline{\gamma}_l} \leq 2e^{3\overline{\gamma}_l}$$

and, consequently,

$$\overline{\lambda} < (2e^{3\overline{\gamma}_l} - 1)/3.$$

Thus,

$$\begin{aligned} \lambda e^{\overline{\gamma}} &\leq \overline{\lambda}e^{4\overline{\gamma}_l + \overline{\gamma}} \\ &\leq \overline{\lambda}e^{5\overline{\gamma}_l} \\ &< e^{5\overline{\gamma}_l}(2e^{3\overline{\gamma}_l} - 1)/3 \\ &\leq e^{5/16}(2e^{3/16} - 1)/3 = 0.6435 \dots < t_2; \end{aligned}$$

compare (3.3). This proves (3.4) and completes the proof of the second line of Table III.

#### IV. APPLICATIONS TO INTERVAL ARITHMETIC

The results of the previous section can be put in a form that is more familiar for interval arithmetic. Let  $\overline{x}$  be any member of  $\mathcal{M}$  and let  $\overline{\tau} \geq 0$  be a member of  $\mathcal{L}$ . Then

$$A(\overline{x}, \overline{\tau}) = \{x | x \simeq \overline{x}; \text{ap}(\overline{\tau})\} \quad (4.1)$$

is an interval of real numbers which we will call an *ap interval*. If in addition  $\overline{x}$  is nonzero, then we will say that

$$R(\overline{x}, \overline{\tau}) = \{x | x \simeq \overline{x}; \text{rp}(\overline{\tau})\} \quad (4.2)$$

is an *rp interval*. In terms of this notation, the second conversion formula displayed in Table II takes the form

$$\begin{aligned} A(\overline{x}, \overline{\alpha}) &\subseteq R(\overline{x}, (\overline{\alpha}/|\overline{x}|)e^{\overline{\alpha}/|\overline{x}|}) \\ &\subseteq R(\overline{x}, (\overline{\alpha}/|\overline{x}|)(1 + 3\overline{\lambda} + 12\overline{\gamma}_l)) \end{aligned}$$

where the first set containment is valid provided  $\overline{\alpha}/|\overline{x}| \leq t_2$  and both

TABLE V  
OPERATION COUNTS FOR ap-ERROR BOUNDS

Main Operation	Error-Bound Operations			
	+	x	/	abbrev.
+, -	3	3	0	1
x, /	4	4	2	3

TABLE VI  
OPERATION COUNTS FOR rp-ERROR BOUNDS

Main Operation	Error-Bound Operations			
	+	x	/	abbrev.
x, /	3	2	0	0
+	6	5	1	3
-	4	5	1	3

set containments are valid provided  $1 + 3\overline{\lambda} + 12\overline{\gamma}_l \leq 2$ . Similarly, the first formula in Table III takes the form

$$\begin{aligned} A(\overline{x}, \overline{\alpha}) + A(\overline{y}, \overline{\beta}) &\subseteq A(\overline{x + y}, (\overline{\alpha} + \overline{\beta} + \overline{\gamma}|\overline{x + y}|)e^{\overline{\gamma}}) \\ &\subseteq A(\overline{x + y}, ((\overline{\alpha} + \overline{\beta}) + \overline{\gamma}|\overline{x + y}|)(1 + 14\overline{\gamma}_l)) \end{aligned}$$

where the first set containment is valid universally and the second set containment is valid provided  $1 + 14\overline{\gamma}_l \leq 2$ .

#### Ap-Interval Arithmetic Versus rp-Interval Arithmetic

We compare the lower-mode operation counts for an interval arithmetic system based on ap intervals to a system that is based on rp intervals. The counts are made by examining Tables III and IV, and the data obtained are given in Tables V and VI. The counts for abbreviation refer to abbreviations of data to  $\mathcal{L}$  from  $\mathcal{M}$ ; these are zero when  $\mathcal{L}$  and  $\mathcal{M}$  are identical.

We note that the total number of lower mode operations needed to compute error bounds when an equal number of main-mode additions, subtractions, multiplications and divisions are present is the same in an ap system as in an rp system when  $\mathcal{L}$  and  $\mathcal{M}$  are identical, i.e., when abbreviation to  $\mathcal{L}$  from  $\mathcal{M}$  is not present. We observe, however, that in this case the rp formulation saves two division operations at the cost of two addition operations, for a net gain in speed over the ap formulation when the reasonable assumption is made that division in  $\mathcal{L}$  is more time-consuming than addition in  $\mathcal{L}$ . Furthermore, when  $\mathcal{L}$  is different from  $\mathcal{M}$  the rp formulation is favored again since it requires two fewer abbreviations than the ap formulation.

As a result of these considerations we conclude that an rp form of interval arithmetic may be more cost-effective than an ap form, especially when the proportion of multiplication and division operations in the main-mode operation mixture exceeds 50 percent. This conclusion is not surprising when we recall that the floating-point representation is associated more naturally with relative errors than with absolute errors.

Conventional interval arithmetic is an ap form of interval arithmetic. If directed rounding is assumed available in the computer arithmetic, the operation counts are substantially lower than those shown in Table VI. We remark, however, that improvements could be made in the operation counts shown in Table VI. For example, rp-error bounds in a given floating-point mode  $\mathcal{M}$  could be represented as integer multiples of the unit of the last place, i.e.,  $\overline{\gamma}$ . However, such integers would have to be allowed to approach the size of  $1/\overline{\gamma}$ . Using such a representation, the compensating factor  $1 + 10\overline{\gamma}_l$ , shown in Table IV for multiplication and division, reduces to unity. In conse-

quence, in Table VI the operation count of two for multiplications in  $\mathcal{L}$  reduces to zero and the operation count of three for additions in  $\mathcal{L}$  reduces to two. Furthermore, these operations are integer operations (although using multilength operands in all likelihood). This may be superior to conventional interval addition and subtraction, which requires two floating-point additions with directed rounding to form the interval for the result. No similar approach can be used to reduce the operation counts for ap-interval arithmetic, because the error bounds must have scale factors associated with them.

#### Combined ap-rp Interval Arithmetic

Another approach to reducing operation counts would be to design interval arithmetic so as to allow the representation of error bounds in either ap or rp form. The distinction could readily be made in a computer representation by using the sign bit, since error bounds are always positive or zero. In such a system all computable forms of error bounds, shown in Tables II–IV, would be employed.

Experience in constructing error-bound formulas by Olver's approach has shown that a judicious combination of ap and rp forms often leads to an efficient form of result. A simple example is the error-bound formula for inner products [7, Equation (7.7)]. Here the data are assumed in rp form and the result is produced in ap form. Each product is multiplied in rp form, with error bound in ap form produced directly (see Line 2 of Table IV), before being added into the accumulator. This process, which we will call the *rp-ap method* for purposes of comparison, could be carried out in a combined system of interval arithmetic. Let us refer to the accumulation of an inner product of  $n$  terms in rp-interval and ap-interval arithmetic as the *rp-rp method* and the *ap-ap method*, respectively. Table VII shows the operation counts required in the evaluation of computable error bounds by these three methods. The data are obtained from Tables V and VI, together with operation counts made from the second line of Table IV in the rp-ap method. We observe that, since conversion of the computed inner product from ap form to rp form or vice versa involves only a constant number of operations, an advantage is gained by the rp-ap method over either of the other two methods.

It should be understood that the rp-ap method is not equivalent to Olver's error-bound formula, in which the errors are not bounded at each arithmetic operation but rather the compensating factor is determined at the end of the inner-product accumulation. A comparison can be made by straightforward application of the precomputable forms of error bounds in Tables II–IV. The bound obtained by interval arithmetic can be put in the form

$$P_{n-1}(e^\gamma) + \gamma e^\gamma Q_{n-2}(e^\gamma)$$

where  $P_n$  and  $Q_n$  are polynomials of degree  $n$  with positive coefficients. Here,  $n$  is the number of terms in the inner product. Comparison with Olver's formula shows that it is equal to this same function with the arguments of  $P_{n-1}$  and  $Q_{n-2}$  replaced by unity.

#### Implementation of ap-rp Interval Arithmetic

We mentioned earlier that a Fortran precompiler exists that has been successfully used to introduce rounded-interval arithmetic operations into Fortran [2], [9]. Here we only suggest that similar techniques could be used to implement combined ap-rp interval arithmetic. In the declarations of the types of the variables used in a program, some would be associated with ap intervals and others with rp intervals. The precompiler would choose the types for temporary variables, and it would preprocess one of the operands by using a conversion formula from Table II when two arithmetic operands are of opposite type.

A simple criterion for use by the precompiler in choosing types is to employ the form of error bound in the current operation that leads to the fastest execution of the next operation that uses the current result as an operand. Thus the ap form is favored if the next usage involves addition or subtraction, otherwise the rp form is favored. Application of this criterion within the scope of a single arithmetic expression would require no fundamental change in the way compilers assign temporary variables or parse arithmetic expressions. If the next

TABLE VII  
ERROR-BOUND OPERATION COUNTS FOR THE ACCUMULATION OF AN INNER PRODUCT OF  $n$  TERMS BY THREE METHODS OF INTERVAL ARITHMETIC

Method	Error-Bound Operations			
	+	x	/	abbrev.
rp-ap	$7n - 3$	$7n - 3$	0	$2n - 1$
rp-rp	$8n - 5$	$7n - 5$	$n - 1$	$3n - 3$
ap-ap	$7n - 3$	$7n - 3$	$2n$	$4n - 1$

usage of the current result cannot be determined, the criterion fails, but this cannot happen within the scope of a single arithmetic expression. The choice of type is more difficult for declared variables but the programmer is free to experiment with alternative choices. Although we would not expect major differences in the sharpness of the error bounds produced, useful differences in execution time would be discernible.

#### V. SUMMARY

The computation of *a posteriori* error bounds in floating-point procedures is rendered more effective by isolating subprocedures which are amenable to more powerful forms of error analysis than straightforward interval arithmetic. The example of Olver and Wilkinson's analysis of Gaussian elimination in terms of ap and rp errors illustrates the possibility of achieving highly realistic error bounds in algorithms where interval arithmetic is not very successful. The solution of a set of simultaneous linear algebraic equations is usually done by calling a subroutine, which could be constructed so as to produce rigorous error bounds on the computed solution. This is the most natural way of reliably delivering the results of error analyses of standard algorithms to general computer users.

The value of interval arithmetic arises in those parts of a numerical computation for which reliable error-bound-producing software is not available, because interval arithmetic permits the programmer to avoid the rather subtle technical details of constructing and coding the error bounds. Our comparison of the operation counts needed to implement ap-interval arithmetic (which corresponds to the conventional Moore type of interval arithmetic in centered form) and rp-interval arithmetic suggests that the rp form may be superior in a floating-point environment. Furthermore, a combined form of interval arithmetic that uses both ap and rp intervals is feasible and may prove to be better than either of the other two forms.

#### ACKNOWLEDGMENT

The author is pleased to acknowledge helpful discussions with F. W. J. Olver and D. J. Orser, as well as the constructive criticism of one of the referees.

#### REFERENCES

- [1] C. W. Clenshaw and F. W. J. Olver, "An unrestricted algorithm for the exponential function," *SIAM J. Numer. Anal.*, vol. 17, pp. 310–331, Apr. 1980.
- [2] F. D. Cray, "A versatile precompiler for nonstandard arithmetics," *Ass. Comput. Mach. Trans. Math. Software*, vol. 5, pp. 204–217, June, 1979.
- [3] R. E. Moore, *Interval Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1966.
- [4] —, *Methods and Applications of Interval Analysis*. Philadelphia, PA: Soc. Indust. App. Math., 1979.
- [5] K. Nickel, "Interval-analysis," in *The State of the Art in Numerical Analysis*, D. Jacobs Ed. London, England: Academic, 1977, pp. 193–225.
- [6] F. W. J. Olver, "A new approach to error arithmetic," *SIAM J. Numer. Anal.*, vol. 15, pp. 368–393, Apr. 1978.



- [7] —, "Further developments of rp and ap error analysis," *IMA J. Numer. Anal.*, vol. 2., pp. 249-274, July 1982.
- [8] F. W. J. Olver and J. H. Wilkinson, "A posteriori error bounds for Gaussian elimination," *IMA J. Numer. Anal.*, to be published.
- [9] J. M. Yohe, "Software for interval arithmetic: a reasonably portable package," *Ass. Comput. Mach. Trans. Math. Software*, vol. 5, pp. 50-63, Mar. 1979.

### Concurrent Error Detection in Multiply and Divide Arrays

JANAK H. PATEL AND LEONA Y. FUNG

**Abstract**—A method proposed for concurrent error detection in ALU's is used in the design of multiplier and divider arrays. This method, called recomputing with shifted operands (RESO), can detect all errors caused by failures confined to a cell of the cellular array. The assumption that the failures are confined to a small area of an integrated circuit and the precise nature of the failures is not known is very applicable to VLSI circuits. RESO uses time redundancy for error detection and requires only a small increase in the hardware of a multiply and divide array.

**Index Terms**—Cellular logic, concurrent error detection, dividers, multipliers, RESO.

### I. INTRODUCTION

In a computer system, the basic arithmetic operations are addition, subtraction, and shifting. More complicated arithmetic operations such as multiplication and division are computed by iterations of the basic arithmetic operations. When concurrent error-detection capabilities are built in the system to check the basic ALU, it means that each iteration step of multiplication and division is also checked. Some arithmetic codes, such as the AN code, residue code, and inverse residue code, and some nonarithmetic codes, such as the check-symbol prediction scheme, were developed to detect errors in addition and subtraction [1]–[5]. If multiplication and division are done using cellular arrays, most of the above schemes cannot be easily extended to detect errors in the computations. Moreover, even if these schemes can be extended, they require a large increase in the complexity of the circuitry and, therefore, the area on the chip. Furthermore, some of them are based on the traditional stuck-at fault model which is shown to be inadequate in describing failures in circuits [6]–[9]. Since the error-detection schemes which utilize a redundancy in time can keep chip area and interconnections to a minimum, with an appropriate fault model they can be very attractive for fault-tolerant VLSI-based systems. Time redundancy for error detection and correction has been used by several researchers [10]–[17]. In particular, the use of time redundancy in arithmetic operations has been reported in [10] and [16].

The time redundant scheme RESO, reported earlier [16], is able to detect in a typical ALU all functional errors resulting from failures confined to a certain area of the chip, for example, a bit slice of the ALU. These include errors caused by permanent as well as intermittent failures. The traditional single fault assumption of stuck-at 1 or 0 logical value is not appropriate for VLSI technology. A more appropriate assumption is that the failure is confined to a small area of the chip and that the nature of the failure is not precisely known. Also, the assumption that physical failures in components simply change the logical values of the outputs of these components may not always be valid since indeterminate logic levels may occur [6]–[9]. However, at some higher functional level, the effect of failures will

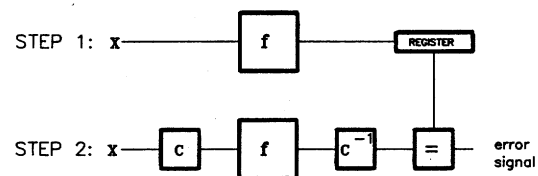


Fig. 1. An approach to error detection using time redundancy.

be felt as changes in logical values. For example, even if the failure in the shifter of a multiplier circuit may not be modelable as changes in the logical values of each individual flip flop, it might still be possible to consider the effect of failures as changes in the logical values of the multiplier outputs. Thus, in the absence of the knowledge of physical failure modes, it is more appropriate to deal with the failures at the functional level. Throughout this paper, we assume that faults will make their effect felt at the level of a small network in terms of altered logical values of the output. However, no assumption is made regarding the logic values being stuck or not or the combinational circuits becoming sequential or not. The functional level chosen in this paper is typically a full-adder or subtractor cell of an array.

One interesting observation pertaining to failures and VLSI design is that the nature of the circuit layout affects the error-detection capability. Since in an integrated circuit a portion of the chip area may contain unrelated lines and components, the assumption that a faulty area only affects the components of a specific subnetwork may not hold true. For example, the circuit cell containing a bit slice of an adder may have lines crossing over it, but not used by it. If these lines are used by other bit slices, then a failure in the circuit cell affects more than one bit slice. It is possible, however, to change the layout so that the area of a bit slice does not contain gates and lines of other slices. In other words, it is possible to lay out a circuit in such a way that it does not contain unrelated components or lines. We shall assume this to be the case for our error detection technique.

### II. ERROR DETECTION USING TIME REDUNDANCY

We briefly review here the method of error detection proposed in [16]. An approach using time redundancy for concurrent error detection in a function unit  $f$  is shown in Fig. 1. Let the function  $c$  be such that  $c^{-1}(f(c(x))) = f(x)$  for all  $x$ . During step 1,  $f(x)$  is computed and stored in a register. During the second computation step,  $c(x)$  is input to the same function unit  $f$ , yielding  $f(c(x))$ . This output is fed to the  $c^{-1}$  unit, yielding  $c^{-1}(f(c(x)))$ , which is the same as  $f(x)$  by assumption. This output is compared to the result in the register; a mismatch indicates an error in computation. If the fault in the function unit  $f$  affects the computation of  $f(x)$  and  $f(c(x))$  differently, then the fault will be detected. One can view the function  $c$  as a coding function and  $c^{-1}$  as a decoding function. For a cost-effective design, the function  $c$  must be such that it provides a very good error coverage and is far less complex than the function  $f$ . The method of error detection called recomputing with shifted operands (RESO) is such a cost-effective method [16].

RESO uses the principle of time redundancy of Fig. 1 where the coding function  $c$  is the left shift operation and the decoding function  $c^{-1}$  is the right shift operation. Thus, in the first computation step,  $f(x)$  is computed and stored in the register. During the recomputation step,  $x$  is shifted left by  $k$  bits and then input to unit  $f$ . The output is shifted right  $k$  bits and compared to the result of the first step. A mismatch indicates an error in either computation step.

RESO- $k$  is defined to be the name of the error-detection scheme achieved by recomputing with operands shifted by  $k$  bits. The hardware that the RESO- $k$  needs to detect errors in a functional unit for  $n$ -bit operations includes two shifters, a register, an  $(n+k)$ -bit functional unit, and an equality checker. A totally self-checking equality checker may be implemented based on 1-out-of-2 code checkers [18]. RESO- $k$  detects all errors when the failures in an ALU are confined to  $k$  adjacent bit slices for logical operations and  $(k -$

Manuscript received April 16, 1982; revised November 17, 1982. This work was supported by the Naval Electronics Systems Command under VHSIC Contract N00039-80-C-0556.

J. H. Patel is with the Coordinated Science Laboratory and the Department of Electrical Engineering, University of Illinois, Urbana, IL 61801.

L. Y. Fung was with the Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801. She is now with the STC Computer Research Corporation, Santa Clara, CA 95051.