Rounding Errors

boley

July 20, 1997

1 introduction

Rounding errors are the errors arising from the use of floating point arithmetic on digital computers. Since the computer word has only a fixed and finite number of bits or digits, only a finite number of real numbers can be represented on a computer, and the collection of those real numbers that can be represented on the computer is called the *floating point system* for that computer. Since only finitely many real numbers can be represented exactly, it is possible, indeed likely, that the exact solution to any particular problem is not part of the floating point system and hence cannot be represented exactly. Ideally, one would hope that one could obtain the *representable number* closest to the true exact answer. When doing simple computations this is usually possible, but is more problematical after long or complicated computations. Even the four basic operations, addition, subtraction, multiplication, and division, cannot be carried out exactly, so the intermediate results in any computation will suffer from contamination of *rounding errors*, and the final results will suffer from the accumulated effects of all the intermediate rounding errors. The field of *numerical analysis* is the study of the behavior of various algorithms when implemented in the floating point system subject to rounding errors. In this article, we describe the main features typically found in floating point systems in computers today, and give some examples of unusual effects that are caused by the presence of rounding errors.

2 representation of floating point numbers.

2.1 mantissa + exponent

All computers today represent floating point numbers in the form $\texttt{mantissa} \times \texttt{base}^{\texttt{exponent}}$, where the `mantissa` is typically a number less than 2 in absolute value, and the `exponent` is a small integer. The `base` is fixed for all numbers and hence is not actually stored at all. Except for hand-held calculators, the `base` is usually 2 except for a few older computers where the `base` is 8 or 16. The `mantissa` and `exponent` are represented in binary with a fixed number of bits for each. Hence a typical representation is

$$[\, s \quad e_7 \quad e_6 \quad \cdots \quad e_0 \quad m_{23} \quad m_{22} \quad \cdots \quad m_1 \quad m_0 \,] \tag{1}$$

where s is the sign bit for the mantissa, $e_7, \ldots, e_0$ are the bits for the `exponent`, and $m_{23}, \ldots, m_0$ are the bits for the `mantissa`. If the `base` is fixed at 2, then the number represented by the bits (1) is

$$(-1)^s \times (m_{23} \cdot 2^0 + m_{22} \cdot 2^{-1} + \cdots + m_0 \cdot 2^{-23}) \times 2^{\texttt{exponent}}, \tag{2}$$

where the `exponent` is an 8-bit signed integer. In this example, we have fixed the number of bits for the `mantissa` and the `exponent` to 24 and 8, respectively, but in general these vary from computer to computer, and even within the computer vary from *single* to *double precision*. Notice that the `mantissa` represented in (2) has the "binary point" (analog to the usual decimal point) right after the leftmost digit. Regarding the `exponent` as a signed integer, it is not typically represented as a ones or twos complement number but more often in *excess 127* notation, which is essentially an unsigned integer representing the number 127 larger than the true exponent. Again, if we have k bits instead of 8 as in this example, then the *127* is replaced by $2^{k-1} - 1$.

We illustrate this with a few examples, where we shorten the mantissa to 7 bits plus a sign and the exponent to 4 bits. Hence the exponent is in *excess 7* notation:

decimal	binary	bits	remarks
$+5/2$	$+1.01 \times 2^1$	0 1000 *1* 010000	
$-5/2$	-1.01×2^1	1 1000 *1* 010000	
$+20$	$+1.01 \times 2^4$	0 1011 *1* 010000	
$1/3$	$+1.010101 \times 2^{-2}$	0 0101 *1* 010101	inexact
$1/10$	$+1.100110 \times 2^{-3}$	0 0100 *1* 100110	inexact

(3)

We remark that this representation, using normalized mantissas and excess notation for the exponents, allows one to compare two positive floating point numbers using the usual integer compare instructions on the bit patterns.

2.2 normalization

Notice that in (3) there can be multiple ways to represent any particular decimal number. If the leading digit of the `mantissa` is zero, then the number if said to be *unnormalized*, otherwise it is said to be *normalized*. So we could also use the representation

decimal	binary	bits	remarks
$+20$	$+0.00101 \times 2^7$	0 1110 0001010	unnormalized
$1/3$	$+0.101010 \times 2^{-1}$	0 0110 0101010	inexact & unnorm.

(4)

When unnormalized, we lose space for significant digits, hence floating point numbers are always stored in normalized fashion. We see that in (3), the normalized representation for the number $1/3$ captures more nonzero bits than the unnormalized representation (4). When the `base` is equal to 2, then the leading digit of the *mantissa* is just a bit whose only possible nonzero value is 1, and hence it is not even stored. So in the representation (1), the bit m_{23} is always 1 and is not actually stored in the computer. When not stored in this way, the bit m_{23} is called an *implicit bit*. These bits are written in *italics* in (3).

2.3 special numbers, overflow, underflow

The representation (1) with the implicit bit m_{23} does not admit the number 0, since 0 would have an all zero `mantissa` that must be unnormalized. To accommodate this, certain special bit patterns are reserved to zero and certain other special "numbers". A zero is often represented by a word of all zero bits, which would otherwise represent the smallest representable positive floating point number. If a calculation gives rise to an answer less than the smallest representable number (in absolute value), then an *underflow* condition is said to exist. In the past, the result was simply set to zero, but more recently, the result was denormalized.

The use of gradually denormalized numbers involves those floating point numbers which are less (in absolute value) than the smallest representable normalized number. As discussed in [3], there is a relatively big gap between the the smallest representable normalized number and zero. To fill this gap, the IEEE decided to allow for the use of unnormalized numbers. We can illustrate this with the representation in (3). The smallest normalized number representable in (3) is $+1.00_{\text{binary}} \times 2^{-7}$. However we can represent smaller numbers in an unnormalized manner, such as $+0.10_{\text{binary}} \times 2^{-7}$. Since we have adopted the convention of using the implicit bit, such an unnormalized number cannot be encoded in this format. The solution is to provide that the smallest representable normalized number be actually $+1.00_{\text{binary}} \times 2^{-6}$, reserving the smallest possible exponent value for unnormalized numbers. This was what has been adopted in the IEEE standard (see below). Since this smallest exponent value has all its bits equal to zero, the representation of the number zero in this format becomes just a special case of such unnormalized numbers. As pointed out by [3], the use of denormalized numbers also guarantees that the computed difference of two unequal numbers will never be zero.

A more serious problem occurs if the result of the calculation is larger than the largest representable number. This is called an *overflow* condition, and in most older computers this would generate an error. However in the recent IEEE floating point standard (discussed below), such a result would be replaced with a special bit pattern representing *plus-infinity* or *minus-infinity*. When two such *infinities* are combined, the result can be totally undefined, so yet another special bit pattern is reserved for such a result. This last result is called *Not A Number*, and is often printed by most computer systems as `NaN`. By not generating an exception upon overflow, programs may fail more gracefully.

2.4 rounding vs chopping

Another issue affecting rounding errors is the choice of rounding strategy. Given any particular real number, which nearby floating point number should one use? For example in (3), when we represented 1/3 as an unnormalized number we chopped away the last bit, but an alternative choice would be to round up to the next higher number to yield $+0.101011_{\text{binary}} \times 2^{-1}$. The error committed in chopping in this case is .0052 but in rounding is only .0026. But rounding requires slightly more computation since the digits being removed must be examined. This issue arises when converting a number from an external decimal representation and when trying to fit the result of an intermediate arithmetic operation into a memory word. This is because the arithmetic logic unit on most computers actually operate on more digits than can fit in a word, the extra digits being called *guard digits* discussed below.

The IEEE Standard actually provides that the default rounding strategy should be a "round to even" strategy. The "round to even" mode is exactly the rounding strategy described above, except when the number being rounded lies exactly half way between two representable numbers, as in rounding 12.5 to an integer. The default "round to even" strategy selects the representable number whose last digit is even, so that 12.5 would round to 12 and not 13. If the rounding in this case were always up, then more numbers would end up being increased than decreased during the rounding process. If the combinations of trailing digits occur equally likely, it is generally desirable that the number of times the rounding is up is about equal to the number of times the rounding is down, to try to cancel out their effect as much as possible.

2.5 guard digits

Guard digits are extra digits kept only within the Arithmetic Logic Unit during the course of individual floating point operations. They are never stored in memory. The Arithmetic Logic Unit carries out the operation using at least one extra *guard digit*, then the result is rounded to fit in the register of memory word. We illustrate the effect of guard digits using the simple addition of two decimal floating point numbers $1.01 \times 10^{+1}$ and -9.93×10^{0} (this example is from [3]), where we keep 3 decimal digits in the mantissa. To accomplish this, the first step for the arithmetic logic unit is to shift the decimal point in the second operand to make the exponents match, yielding $-.993 \times 10^{+1}$. Then the mantissas may be added together directly. The accuracy of the answer is greatly affected by the number of digits kept for the computation. The simplest approach is to use simple chopping and to keep only the digits corresponding to the larger operand. The result in this case is $1.01 \times 10^{+1} - 0.99 \times 10^{+1} = 2.00 \times 10^{-1}$. If, however, we keep at least one extra *guard digit*, then we obtain $1.010 \times 10^{+1} - 0.993 \times 10^{+1} = 1.70 \times 10^{-1}$. The latter answer is exact whereas the former result has no correct digits.

The reader may ask whether keeping just one guard digit suffices to make a significant enhancement to the accuracy of floating point arithmetic operations. The answer can be found in [3] in which it is proved that if no guard digit is kept during additions, then the error could be so large as to yield no correct digits in the answer, whereas if just one guard digit is kept during the operation, the result being rounded to fit in the memory word, then the error will be at most the equivalent of 2 units in the last significant digit. In this context, the "correct answer" is regarded as the answer computed using all available digits and keeping "infinite precision" for the intermediate results.

2.6 IEEE standard

The previous discussion has shown that there are many choices to be made in representing floating point numbers, and in the past different manufacturers have made different, incompatible, choices. The result is that the behavior of floating point algorithms can vary from computer to computer, even if the precision (number of bits used for `exponent` and `mantissa`) stays the same. In an attempt to make the behavior of algorithms more uniform across platforms, as well as improving the performance of such algorithms, the IEEE has established a *floating point standard* which specified some of these choices [6, 7]. This standard specifies the kind of rounding that must be used, the use of guard digits, the behavior when underflow or overflow occurs, etc. The first standard [6] was limited to 32 and 64 bit floating point words, and provided for optional extended formats for computers with longer words. The second standard [7] extended this to general length words and bases. The principal choices made in [6] include the following:

- rounding to nearest (also known as round to even).
- base 2 with a sign bit and an implicit bit.

- single precision with 8 bit exponent and 23 bit mantissa fields (not including the implicit bit).
- double precision with 11 bit exponent and 52 bit mantissa fields (not including the implicit bit).
- the presence of $\pm\infty$ and `NaN` (Not a Number), as well as ± 0.
- gradually denormalized numbers for those numbers unrepresentable as normalized numbers.
- user-settable bits to turn on exception handling for overflow, underflow, etc. and to vary the rounding strategies.

We have tried to motivate some of these choices with the above discussion, but detailed formal analyses of these choices can be found in [3].

2.7 usual model for round-off error

In order to analyse the behavior of algorithms in the presence of round-off errors, a mathematical model for round-off errors is defined. The usual model is as follows, where $\odot$ represents any of the four arithmetic operations:

$$fl(a \odot b) = (a \odot b) \cdot (1 + \epsilon)$$

where $|\epsilon| \leq$ `macheps`, and `macheps` is called the *unit round-off* or *machine epsilon* for the given computer. The motivation behind this model is that the best any computer could do is to perform any individual arithmetic operation exactly, and then round or chop to the nearest floating point number when finished. The rounding or chopping involves changing the last bit in the (base 2) `mantissa`, and hence the `macheps` is the value of this last bit – always relative to the size of the number itself. Needless to say, this model can be expensive to implement, so some computer manufacturers have designed arithmetic operations which do not obey this model, but one can show that one or two guard digits suffice to be consistent with this model.

For most users of higher level languages, the details of the floating point representation (especially the length of a computer word) are generally hidden from the user. Hence the `macheps` has a definition that can be computed in a higher level language, not specifically by the number of bits in a word. The `macheps` is defined by the value of ϵ yielding the minimum in

$$\min_{\epsilon > 0} fl(1 + \epsilon) > 1. \tag{5}$$

This formula can be used to calculate `macheps` by trying a sequence of trial values for ϵ, each entry one half the previous, until equality in (5) is achieved. The specific value of `macheps` depends on the rounding strategy. This can be most easily illustrated with 3 digit decimal floating point arithmetic. The smallest s such that $fl(1+s) > 1$ is 1.00×10^{-3} in chopping, 5.00×10^{-4} if a traditional rounding strategy is used, and 5.01×10^{-4} if rounding to even is used. In general, `macheps` in rounding is approximately half that obtained using chopping.

3 examples of catastrophic effects of round-off error

To illustrate how rounding errors can accumulate catastrophically in unexpected ways, we give two examples adapted from [9]. An extensive introductory discussion on the effects of rounding error in scientific computations involving the use of floating point can be found in [9, 4].

Of the four arithmetic operations, subtraction and addition are really the same operation. Most loss of significance and cancellation errors described below arise from these two operations. Multiplication and division give rise to problems only if the results overflow, underflow or must be denormalized. An unusual effect of the fact that floating point numbers are discrete in nature is that the operations no longer obey the usual laws of the real numbers. For example, the associative law for addition does not hold for the floating point numbers. If s is a positive number less that `macheps`, but more that `macheps`$/2$, then $1 + (s + s)$ will be strictly bigger than 1, but $(1 + s) + s$ will equal 1. This is an extreme case, but the order in which numbers are added up can affect the computed sum markedly. This is further illustrated by the first example below.

It has been pointed out that the use of the denormalized numbers mean that programs can depend on the fact $fl(a - b) = 0$ implies $a = b$. However, it can still happen that $fl(a * b) = a$ when $a \neq 0$ and $b \neq 1$. This can happen, for example, when a is the smallest representable floating point number, and b is a number between .6 and 1, when rounding is used. Programs whose logic depend on $fl(a * b)$ being always different

from a can suffer very mysterious failures. However, generally, multiplication and division do not give rise to catastrophic rounding errors unless numbers near the ends of the exponent range are involved, or when combined with other operations.

3.1 taylor series for exp(-40)

A simple algorithm to compute the exponential function e^x is to use the well known Taylor series for it:

$$e^x = \sum_{i \geq 0} \frac{x^i}{i!}.$$

When $x \geq 0$, this can yield accurate results if one is willing to take enough terms, but if used when $x < 0$, this can yield to catastrophic results, all due to the finite word length of the machine. To take an extreme case, let $x = -40$. Then all the terms after the 140-th term are much less than 10^{-16} and decay rapidly, and the result is also very small: $e^{-40} = 4.2484 \times 10^{-18}$. But simply adding up the terms of the Taylor series will yield 1.8654, which is no where near the true answer. The problem is the terms in this series alternate in sign, and the intermediate terms reach $1.4817 \times 10^{+16}$ in magnitude, and we end up subtracting very large numbers that are almost equal and opposite. This results in severe cancellation.

3.2 numerical derivative of exp(x) @ x=1

Suppose we take the naive approach to approximate the numerical derivative of a function f:

$$f'(x) = \frac{f(x+h) - f(x)}{h},$$

for some suitable small h. Applying this to $f(x) = e^x$ and taking the derivative at $x = 1$, we find that we get as much accuracy with $h = 2 \times 10^{-6}$ as with $h = 10^{-10}$ on a machine with approximately 16 decimal digits in the mantissa. In both cases, the error is about 3×10^{-6}, and less than half the computed digits are good. Here again we have severe cancellation from subtracting numbers that are almost equal. Hence simply making the stepsize h smaller does not lead to more accuracy.

4 Effect on algorithms

4.1 round-off causes perturbation to data and to intermediate results

The examples above are extreme cases showing that catastrophic loss of accuracy can result if floating point arithmetic is not used carefully. The effect of round-off error is applied to each intermediate result and is guaranteed to be small relative to those intermediate results. However, in some cases those intermediate results can be larger than the final desired results, leading to errors much larger than would be expected from just the sizes of the input and final output of a particular algorithm. However, in some algorithms such as when simulating an ordinary differential equation (such as a control system) $\dot{x} = Ax + f$ where f is a forcing function, the intermediate results may not be any larger than than the final or initial values, yet severe loss of accuracy can result. One source of error is the propagation of intermediate errors, and in nasty cases the effect of those intermediate errors can grow becoming more and more significant as the algorithm proceeds

4.2 algorithm stability vs conditioning of problem

In an attempt to analyse and alleviate the effects of rounding errors, numerical analysts have developed paradigms for the analysis of the behavior of numerical algorithms and have used these paradigms to develop algorithms themselves for which one can prove that the effect of rounding errors is bounded. It is useful to describe these paradigms. The first and most fundamental is the concept of algorithm stability versus conditioning of the problem. The latter refers to the ill-posedness of the problem. If a problem is ill-posed, then slight variations to the coefficients in the problem will yield massive changes to the exact solution. In this case, no floating point algorithm will be able to compute a solution with high accuracy. If the problem is well-posed, then one would expect a good algorithm to compute a solution with full accuracy. An algorithm that fails that requirement is called *unstable*. An algorithm that is able to compute solutions with reasonable accuracy for well-posed problems, and does not lose more accuracy on ill-posed problems than the ill-posed problems deserve, is called *stable*.

4.3 relevance to fault tolerance

The study of rounding errors is relevant to fault tolerance in two ways. At the most elementary level, the presence of rounding errors means that no computed solution will be exact, and we cannot check for the presence of faults by checking if the computed solution satisfies some condition *exactly*. Any fault detection system would have to allow for the presence of errors in the solution arising naturally from normal rounding errors. This thus leads to the difficult task of distinguishing between errors arising from natural rounding errors and errors arising from faults. If the underlying problem is ill-posed to any degree (called *ill-conditioned*) then one is also faced with the issue that the accuracy of the computed solution will be very poor, even if that solution were computed correctly.

On the other hand, many numerical algorithms have been shown to be stable in a certain sense. Algorithms arising in matrix computations have been especially well studied. In particular, in the domain of solving systems of linear equations, certain algorithms have been shown to compute the exact solution to a system within a small multiple of `macheps` of the original system of equations, even when the system is moderately ill-posed. In some cases, precise bounds on the possible discrepancy have been derived. These can be used to develop conditions that can be used to check for faults. Note that even if the computed solution exactly satisfies a nearby system of equations, that does not imply that the error in the solution is small, unless the system of equations are very well conditioned. As a consequence, any validation procedure for fault detection can only check for the correctness of the computed solutions indirectly, and not by computing the accuracy of the solution itself.

The result of this analysis has been the development of conditions to check the correctness numerical computations, mainly in the domain of matrix computations and signal processing. These conditions all involve the determination of a set of precise tolerances that are tight enough to enforce sufficient accuracy in the solutions, yet guaranteed to be loose enough to be satisfiable even when solving problems that are moderately ill-posed. Principal approaches in this area involve the use of checksums, the use of backward error assertions and the use of mantissa checksums. In all cases, it has been found that applying these techniques to series of operations instead of checksumming each individual operation has been most successful.

Instead of using tolerances, an alternative approach that has been used with some success is interval arithmetic. Space does not permit a full treatment here, since most software, languages, compilers and architectures do not provide interval arithmetic as part of their built-in features. A synopsis of interval arithmetic, including its uses and applications can be found in [10]. In this article we limit our discussion to a short description. The easiest way to view interval arithmetic is to consider replacing each real number or floating point number in the computer with two numbers representing an interval $[a, b]$ in which the "true" result is supposed to lie. Arithmetic operations are performed on the intervals. For example, addition would result in $[a_1, b_1] + [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$. If all endpoints are positive, then multiplication of intervals would be computed by $[a_1, b_1] \cdot [a_2, b_2] = [a_1 \cdot a_2, b_1 \cdot b_2]$. All the other arithmetic operations and more general situations can be defined similarly. However, if no special precautions are taken, the size of the intervals can grow too large to give useful bounds on the location of the "true" answers. So most successful applications involve more sophisticated analysis of whole series of arithmetic operations such as an inner product rather than analysing each individual operation, or else use some statistical techniques to narrow the intervals. As pointed out in [3], in order to maintain the guarantee that computed intervals contain the "true" answer, it is necessary to round down the left endpoint and round up the right end point of each computed interval. This requires the user to vary the rounding strategy used within the computer. The IEEE standards provide that the hardware provide a way for the user to vary the rounding strategy as well as some other parameters of the arithmetic, but as pointed out by Prof. Kahan [8] most compilers and systems today do not actually provide the user access to that level of hardware.

The remaining part of this article is devoted to a discussion of some of these fault tolerance techniques.

5 Synopsis of fault tolerance techniques for Linear Algebra

We present a short synopsis of various techniques that have been proposed for the verification of floating point computations, mostly in the area of linear algebra. The use of checksums was made popular by Abraham [5]. This method takes advantage of the fact that the result of most computations in linear algebra bears a linear relation to the arguments originally supplied. So a linear combination of those results bears the same linear relation to that same linear combination of the original data. For example, the row operations in Gaussian Elimination (used to solve systems of linear equations) can be checksummed by taking linear combinations of the entries in each row. When two rows are added in a row operation, the checksums are

also added and compared with the checksum generated from scratch from the new computed row. In a floating point environment, the checksums will be corrupted by round-off error, and hence a tolerance must be sued to decide if they match. This tolerance depends on the condition number of the matrix of *checksum coefficients* [2].

Diskless Checkpoint [11] methods are an alternative approach based on saving simple combinations of preselected intermediate states (e.g. checksums) on several independent processors. They can save information at low cost so that once an error has been detected, a correction can easily be applied. The simplest approach is the generate periodic checkpoints of the state of each processor, and keep an "exclusive or" of those checkpoints on yet another processor. If one processor has a failure, the checkpoint that was on that processor can be reconstructed from all the other checkpoints. This process is generally independent of the floating point representation.

Another class of methods involve comparing the results with certain error tolerances. For matrix multiplication the error tolerances are forward error bounds ("how far is the computed answer from the true answer?") [12]. For solving systems of linear equations, the error tolerances are backward error bounds ("how well does the computed answer fit the original problem" or more precisely "how much must the original problem be changed so that the computed answer fits it exactly?") [1]. In these methods, the error bounds used depend critically on the properties of the arithmetic, particularly the `macheps`, and in some cases on the conditioning of the underlying system being solved. Hence these techniques can sometimes detect violations of the mathematical assumptions of solvability due to ill-posedness of the problem.

Yet a third class of methods is derived by considering only the mantissas alone. It turns out that for certain floating point operations (like addition), one can compute checksums of the mantissas alone treating them as integers [**ref?**]. Then the checksum computed the same way derived from the mantissa of the result must match the combination of the original mantissa checksums. Since the checksums are computed using integer arithmetic, round-off errors do not apply. The only limitation to this approach is that this technique cannot be applied to all floating point operations (like multiplication), but can be used to check the addition part of inner products.

References

[1] D. L. Boley, G. H. Golub, S. Makar, N. Saxena, and E. J. McCluskey. Floating point fault tolerance using backward error assertions, 1995.

[2] D. L. Boley and F. T. Luk. A well conditioned checksum scheme for algorithmic fault tolerance. *Integration, the VLSI Journal*, 12:21–32, 1991.

[3] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[4] M. Heath. *Scientific Computing, An Introductory Survey.* McGraw Hill, 1997.

[5] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, C-33(6):518–528, June 1984.

[6] IEEE. *ANSI/IEEE Standard 754-1985 for Binary Floating Point Arithmetic.* IEEE, 1985.

[7] IEEE. *ANSI/IEEE Standard 854-1987 for Radix-Independent Floating Point Arithmetic.* IEEE, 1985.

[8] W. Kahan. The baleful effect of computer languages and benchmarks upon applied mathematics, physics and chemistry. presented at the SIAM Annual Meeting (Stanford Calif), 1997.

[9] D. K. Kahaner, C. Moler, and S. Nash. *Numerical Methods and Software.* Prentice Hall, 1989.

[10] R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996.

[11] J. S. Plank, Y. Kim, and J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *25th International Symposium on Fault-Tolerant Computing*, pages 351–360, 1995. IEEE Computer Society.

[12] A. Roy-Chowdhury and P. Banerjee. Tolerance determination for algorithm based checks using simple error analysis techniques. In *Fault Tolerant Computing Symp. FTCS-23*, pages 290–298, 1993. IEEE Press.