

# Robust Assertions and Fail-Bounded Behavior

**Paula Prata**

Universidade da Beira Interior  
Dep. Informática /IT/CISUC  
Rua Marquês d' Ávila e Bolama  
P- 6200 Covilhã, Portugal  
pprata@di.ubi.pt

**Mario Rela, Henrique Madeira,  
João Gabriel Silva**

Universidade de Coimbra  
Dep. Eng. Informática /CISUC  
Pinhal de Marrocos  
P-3030 Coimbra, Portugal  
{mzrela, henrique, jgabriel}@dei.uc.pt

## Abstract

*In this paper the behavior of assertion-based error detection mechanisms is characterized under faults injected according to a quite general fault model. Assertions based on the knowledge of the application can be very effective at detecting corruption of critical data caused by hardware faults. The main drawbacks of that approach are identified as being the lack of protection of data outside the section covered by assertions, namely during input and output, and the possible incorrect execution of the assertions.*

*To handle those weak-points the Robust Assertions technique is proposed, whose effectiveness is shown by extensive fault injection experiments. With this technique a system follows a new failure model, that is called Fail-Bounded, where with high probability all results produced are either correct or, if wrong, they are within a certain bound of the correct value, whose exact distance depends on the output assertions used.*

*Any kind of assertions can be considered, from simple likelihood tests to high coverage assertions such as those used in the Algorithm Based Fault Tolerance paradigm. We claim that this failure model is very useful to describe the behavior of many low-cost fault-tolerant systems, that have low hardware and software redundancy, like embedded systems, where cost is a severe restriction, yet full availability is expected.*

**Keywords:** Hardware faults, Error detection, ABFT, Robust assertions, Failure models, Fail-bounded.

## 1 INTRODUCTION

An important class of hardware faults is transient bit-flips. They can result from external disturbances like power supply instability in industrial applications or as cosmic radiations in space applications [1]. Some recent studies claim that transient and intermittent faults are becoming also a problem in general purpose groundbased systems. [2] and [3] show that the advances in semiconductor technology, like smaller transistor and interconnect dimensions, lower power voltages, and higher operating frequencies have contributed to increased rates of occurrence of transient and intermittent faults. It is expected that fault tolerance features initially devised for critical systems will have to be embedded into commercial-off-the-shelf systems in the future.

Systems with high dependability require high levels of costly redundancy. A focal point of research is thus the search for techniques that require less redundancy while maintaining high levels of dependability. Behavior-based error detection techniques are quite promising in that respect. Several studies have investigated the possibility of using them to build low redundancy failsilent systems ([4] [5] [6] [7]). A common conclusion of those studies is that the crash of a system due to a fault is easy to detect, but when the system produces some output, it is much harder to know whether it has been affected by some fault or not.

Techniques like control flow checking or memory access monitoring [8] [9] [10] [11] [12] do not detect most of data manipulation errors, nor design faults, because they

cannot evaluate the correctness of the output [5]. To do that we need to have the knowledge of the application in order to build adequate assertions that verify the computation [6]. Generally, assertions use some property of the problem or algorithm. For instance, they can be based on the inverse of the problem, the range of variables, or the relationship between variables [13].

A practical example of using assertions is the Remote Exploration and Experimentation (REE) project that was developed at the Jet Propulsion Laboratory of NASA. That project has investigated the possibility of using assertion based mechanisms to detect the errors induced by cosmic radiation on scientific applications running on state of the art commercial off the self components and handling them on the fly [14].

A particular technique based on assertions is Algorithm Based Fault Tolerance (ABFT), which provides very effective error detection coverage for linear operations with matrices [15]. In spite of its high coverage, in [6] [7] it was shown that ABFT still exhibits some weak points. In this paper, we present the research we have carried out to better characterize those weak points, and the mechanisms we propose to solve them, like protecting data during input/output and ensuring that the ABFT acceptance tests are indeed executed. We essentially claim that those mechanisms, collectively called Robust ABFT, guarantee that, if results are produced by the system, they have been filtered by the acceptance test of ABFT algorithms with very high probability.

To justify our claims, we present the results of extensive fault/error injection campaigns on a sequential version of matrix multiplication with ABFT using double precision matrices. The main objective of this research was to characterize the limits of using assertions in general and not to quantify the error detection capabilities of any specific assertion. Several reasons make matrix multiplication with ABFT appropriate for our study: First, ABFT includes a high coverage assertion, and thus most of the undetected errors result from the weakness of the mechanism itself and not from the weakness of the assertion. Second, matrix multiplication involves linear transformations over the most common data structure used in number crunching applications that is the matrix, and the kind of codification used in that ABFT can be applicable to any linear operation. Using large matrices we should expect to get a significant number of faults that just affect data. The errors produced by such faults are the ones that other mechanisms have greater difficulty in detecting. Finally, working with floating point arithmetic introduces the problem of rounding errors as an additional weak point of using assertions over real numbers.

Since the acceptance tests of ABFTs are just a special kind of assertions, we generalize Robust ABFTs to Ro-

bust Assertions. To evaluate that new technique we used a real control application whose output is protected by some reasonableness checks. Results from extensive fault/error injection campaigns showed the effectiveness of Robust Assertions. This technique ensures that the system results are filtered by the output assertions defined by the programmer, with very high probability. This means that, even if the system does output wrong results, they are not arbitrary: they are within some "distance" of the correct result, where the meaning and scope of that "distance" depends on the particular assertion executed and its detection capability. A system with such behavior cannot be called fail-silent [4], but it is a large exaggeration to say it exhibits Byzantine failure [16], since its behavior is not arbitrary. We thus propose a new failure model, somewhere between Fail silent and Byzantine failure, that we call Fail-Bounded. In other words, although we cannot completely prevent a system from delivering wrong results, we can at least say something about them: they are bounded by the limits imposed to them by the executed assertions, hence Fail-Bounded.

The structure of this paper is that: section 2 presents the state of the art in ABFTs followed by the experimental study we have made to characterize the weak points of ABFT and the mechanisms proposed to handle them. The "robust assertions" technique is proposed in section 3, as a generalization of robust ABFT to any kind of assertion. Section 4 presents the new failure model and section 5 presents the conclusions.

## 2. ALGORITHM BASED FAULT TOLERANCE

The basic approach of ABFT is to apply some encoding to the input data of the calculation, execute the algorithm on the encoded data, and check that the encoding is preserved at the end (correctness test). In the original scheme for matrix multiplication [15], the input matrices are augmented with an additional checksum row and an additional checksum column. Each element of the checksum column/row is the sum of the elements of the original matrix that are in the same row/column. The augmented matrices are then multiplied using an unchanged multiplication algorithm - in the end, the additional row and column of the result matrix should still be the sum of the elements of the same column or row. If that is not the case, some error has occurred. Note that we consider "acceptance test" the verification of the checksum row and column (correctness test) and the final "if" clause where, in the case the verification was successful, the output matrix is written to disk, otherwise an error message is generated (if useful) and no matrix is outputted.

ABFT schemes exist for a number of algorithms, like linear operations on matrices, fast Fourier transforms and iterative solvers for partial differential equations ([15] [17]

[18] [19] [20]). Their main advantage is the close to perfect coverage of data errors, in spite of the low processing and memory overhead. Indeed, in the results presented in [6] and [7], it was shown that ABFT was able to detect most of the data errors, especially those that escaped other more general purpose error detection methods like memory protection and control flow checking.

Although being a quite effective technique, ABFT is not perfect. On one hand, whenever floating point arithmetic is used, as is usually the case for the kind of algorithms where ABFT is applied, difficulties due to round off error arise. On the other hand, the fault models used to calculate the coverage of ABFT schemes are not general enough. The next subsection describes the fault model used to evaluate the techniques proposed in this work. Subsection 2.2 identifies the weak points of ABFT and subsection 2.3 analyses the round off problem. Afterwards, the experimental study of the error detection coverage is presented (subsection 2.4), and some mechanisms to improve that coverage are proposed.

## 2.1. THE FAULT MODEL

The original paper on ABFTs considered algorithms executing on systolic architectures [15], with each processor calculating a bounded number of elements of the result matrix so that, if only one processor at a time could fail, only a bounded number of output data items could be wrong. The case of a uniprocessor was also considered, but the fault model assumed was that at most a random number of consecutive bits of the output could be wrong.

Since then, several works on Graph-Theoretic models for studying the error detecting and locating capabilities of ABFT schemes have been presented. In [21] is considered the initial model, where each processor owns only one data element. The redundant processors, required to check the calculations, were supposed to be fault free. In [22] the model is extended to systems where the data elements can be shared by multiple processors, and [23] [24] studied the case where the processors computing the checks can fail. Anyway, all these models assume multiprocessor architectures like systolic arrays, where very good error containment exists between the several processors involved in the computation.

In most applications it is expensive to have dedicated processors computing the checks, and the data subset that can be affected by a faulty processor is not so well delimited. Experimental studies about the error coverage of the ABFT algorithms designed for general purpose multiprocessors [17] [18] have assumed the following model: Each processor performs some calculations and checks the results of another processor, in order to achieve the desired fault detection and location. A faulty processor can corrupt

all the data it possesses but not the data of another processor. The processor responsible for inputting the data, collecting and interpreting the results (the host) is considered fault free. To evaluate ABFT according to that model, permanent and transient data errors were simulated: After each floating point expression an error injection routine was called to replace, with some probability, the result by a random word [17] [18].

In the experiments reported in this paper we used a much broader and much more realistic fault model, corresponding to general transient faults occurring in the hardware. Using the Xception tool [25], bit flips of one bit, affecting only one machine instruction, one processor, and one functional unit at a time, were injected at random in both time and space. To be random in time, faults were injected at any moment during the execution of the programs under test, affecting both system and application code and data. To be random in space, the functional unit where injection was done was also randomly chosen among Floating point ALU, Integer ALU, Data Bus, Address Bus, General Purpose Registers, Condition Code Register(s), Memory Management Unit and Main Memory. Only one fault was injected in each execution of the programs, with the system being reset between injections, to start always from a clean state. This fault model can cause quite varied and unpredictable results [26]. It is relevant to note here that only the most subtle type of faults (single instruction single bit flips) were injected, since other types of faults like permanent faults and longer lasting transients are easier to detect, because they have a bigger impact on the system, and thus are not so demanding on the error detection mechanisms. The system used to run the experiments was a Parsytec PowerXplorer, a parallel machine with 4 PowerPCs 601, each node with 8 Mbytes of memory, running Parix, a parallel OS similar to UNIX. One of the nodes was connected to a host computer, a Sun SPARC-station, that provides I/O and permanent storage capabilities.

## 2.2. WEAK POINTS OF ABFT

The programs used in the experiments were a parallel version of matrix multiplication and of QR factorization, implemented exactly as described in [18], and a uniprocessor version of matrix multiplication implemented as described in [15], all using double precision floating point arithmetic. All of them read the input matrices from a file and wrote the output to another file.

After the random fault-injections, that were used to collect the statistical results (around 30.000 injections) faults were injected in a more focused way to help in understanding why did the system in some cases output wrong data. This very extensive injection campaign uncovered essentially two weak points of ABFT:

- Data errors can happen outside of the "protection window" of ABFT, e.g. before the checksum rows and columns are calculated on the input matrices, or after the checksum rows and columns are verified on the result matrix;
- A defective matrix can go unnoticed if the fault interferes with the execution of the acceptance test, e.g. because of a control flow error that "jumps" around the conditional branch that finally decides on the matrix validity. For parallel versions where the checking processor is different from the one that made the calculations, this problem can happen in the processor that acts as host, where all test results are collected.

These results confirm those of a more limited experiment described in [6] that only used integer arithmetic.

An additional problem that we have faced was that some faults cause "Not a Number" (NaN) - a special code used in the IEEE floating point format to signal exceptions (signal NaN) or to represent the results of certain invalid operations (quiet NaN) [27] [28]. When a NaN is compared with anything, including itself, it always yields false. We had to introduce some code to detect them, thus including an additional assertion.

### 2.3. ROUNDING ERROR

The correctness test after a calculation using floating point arithmetic cannot be exact because of the finite precision of that kind of arithmetic. A tolerance interval has to be calculated for each program using ABFT. The original approach involves executing the program with several "representative" data sets [17], using a tolerance value starting with zero. Since many false alarms will happen because of round off errors, the tolerance value is slowly raised until there are no false alarms. This trial and error method requires a slow tuning, and does not guarantee false alarms will not arise with other inputs. An alternative technique was proposed in [29], based on simplified error analysis techniques where the tolerance interval is calculated in parallel with the computation. Since the error analysis is approximate, the computed tolerance can be some orders of magnitude higher than the maximum rounding error.

Recently, we have proposed that the tolerance value should be a parameter that establishes the desired accuracy for the solution [30]. Based on the study of the behavior of a checksum scheme for some ill conditioned problems we concluded that besides detecting errors produced by hardware faults the checksums will also detect whether the algorithm is able to compute a solution with the specified accuracy. That proposal is corroborated by the results presented in [14], where the study of numerical tolerances for checksum schemes applied to several matrix computa-

tions has shown that if the computation did not succeed, because the matrix is badly scaled, ill conditioned or numerically unrealistic, the checksum test detects it.

In any case, the most important point to retain here is that the test is inexact: if erroneous results are produced because of a fault, they will not be detected if they fall inside the tolerance margin. We can no longer say that a particular result is correct, but only that it is within a certain interval around the correct result.

### 2.4. IMPROVING THE ERROR DETECTION COVERAGE

The most demanding situation for ABFT usage is when the whole program is executed on a single processor, since in that case no processor boundaries exist to define error containment regions. This is why we will address essentially the uniprocessor case. Besides, if the problem can be solved for uniprocessors, very low redundancy solutions will be possible, and the generalization for assertion-based systems will be easier, since the latter are generally not based on multiprocessors.

We have studied the behavior of a sequential version of matrix multiplication, computing the product of two 200 by 200 double precision matrices with random entries chosen in the range ( -100, +100). The faults injected were transient bit flips of one bit randomly chosen, applied to a randomly selected functional unit at a random instant of time during program execution. In the comparison of the checksums vectors we have used a fixed tolerance of  $10^{-8}$ . This was the minimum value for which we didn't get false alarms in the data set used. To classify the outputs we did a component wise comparison between the result matrix and a reference solution obtained in a run with no injected fault. The outputs were classified in one of the following five categories: 1- wrong result matrix with error inside the tolerance margin; 2- wrong result matrix with error outside the tolerance margin; 3- no outputs because of a crash; 4- no outputs because the program detected an error and does not output the matrix in that case; 5- correct output.

In order to establish the statistical validity of the data presented in this paper we calculate the upper 99% confidence limit estimated for the experimental non-coverage of the error detection techniques studied. We assume that the injected faults are representative faults, that is, the selection probability of each injected fault is equal to its occurrence probability. Since our experiments are Bernoulli trials the number of undetected faults,  $f$ , has a binomial distribution with parameters  $(n, \bar{C})$ , where  $n$  is the number of injected faults and  $\bar{C}$  the non-coverage. Then  $f/n$  is an unbiased estimator for the non-coverage of each mechanism studied. An upper  $100\gamma\%$  confidence limit estimator for the non-coverage is given by

$$\frac{(f+1)F_{2(f+1),2(n-f),g}}{(n+f)+(f+1)F_{2(f+1),2(n-f),g}}, \text{ where } F_{v1, v2, g}$$

is the 100 $\gamma$ % percentile point of the  $F$  (Fisher) distribution with  $v1, v2$  degrees of freedom [31].

The next subsection presents the error detection coverage for the standard version of matrix multiplication with ABFT. After, subsection 2.4.2 describes the technique proposed to solve the problem of data corruption outside the scope of ABFT, and subsection 2.4.3 describes how to protect the execution of the correctness test and presents the results that evaluate both techniques.

#### 2.4.1. STANDARD ABFT

We start by characterizing the behavior of the sequential version of matrix multiplication without any assertion. Table I shows the outcomes of the experiments. As can be seen 25.9% of the injected faults originate undetected wrong outputs outside the tolerance margin. For this version without assertions, of the 604 errors detected by the execution system, 91.55% correspond to data exceptions, 7.95% are illegal instructions and 0.5% are alignment exceptions.

column checksums, several under threshold errors in the elements of a row or column sometimes result in an error in the checksum that is above the threshold, triggering an error detection.

Considering that the mechanism fails only when it does not detect a wrong result outside the tolerance margin, it only fails at detecting 1.8% of the injected faults. Estimating the upper 99% confidence limit for the non-coverage,  $\bar{C}$  of this version of standard ABFT, we get a value of 2.14%. Next, we present the proposals for reducing that non-coverage.

#### 2.4.2. END-TO-END PROTECTION OF DATA

To solve the problem of data corruption outside the part of the program covered by ABFT, some error detection code can be used to protect the data, since data is not transformed there, just transported. For instance, whoever generates the input matrices should also generate a CRC of all their elements. The program with ABFT should then proceed in the following sequence: read the data, calculate the checksum rows and columns, and verify the input CRCs. If it fails, a second attempt at reading the files can be made. If it fails again probably the input files contain corrupted data and the program should be aborted. Note that the CRCs

| Matrix Multiplication                  | without assertions |              | with standard ABFT |              |
|--|--------------------|--------------|--------------------|--------------|
|  | no. of faults      | % of total   | no. of faults      | % of total   |
| Correct output                         | 1283               | 45.7%        | 3719               | 43 %         |
| No output (crash)                      | 64                 | 2.3%         | 165                | 1.9 %        |
| No output (error detected)             | 604                | 21.5%        | 4465               | 51.7 %       |
| Wrong output (within tolerance)        | 129                | 4.6%         | 136                | 1.6 %        |
| <b>Wrong output (out of tolerance)</b> | 729                | <b>25.9%</b> | 160                | <b>1.8 %</b> |
| TOTAL                                  | 2809               | 100.0%       | 8645               | 100.0%       |

**Table I** - Behavior of matrix multiplication without assertions and with standard ABFT

A second version studied is matrix multiplication with standard ABFT, in which the checksum vectors are checked for NaNs before being compared. Table I shows also the outcomes of these experiments using the same fault parameters as before except for the duration of the program that was adapted to the new situation. We have increased the number of injected faults to better understand the undetected errors. As can be seen the results change drastically. Now, 51.7% of the injected faults produce detected errors. They are divided among 20.6% that were detected by the execution system, and 31.1% that were detected by the assertions. The percentage of wrong outputs within tolerance is also significantly lower in the ABFT version, because since the ABFT algorithm is based on row and col-

should only be verified after the checksum row and column have been calculated, otherwise if the matrices would be corrupted after the CRC verification and before the calculation of the row and column checksums, that corruption would go undetected. A similar procedure occurs at the end: the program calculates a CRC over the result matrix, verifies the checksum row and column and, if they are correct, writes the result matrix in the output file together with the CRC. If the result matrix is somehow corrupted after the verification of its validity, whoever reads it from the output file will be able to detect the corruption by verifying the CRC. This method, illustrated in figure 1, is in fact a direct application of the end-to-end argument [32]: we have to protect data all the way, and not just during part of the way.

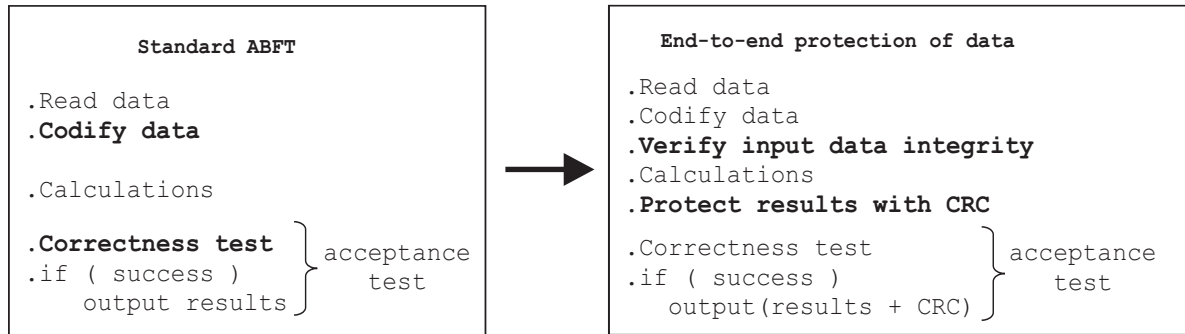


Figure 1 – Standard ABFT versus ABFT with end-to-end protection of data.

### 2.4.3. CHECKING THE CORRECTNESS TEST

The other problem that ABFT algorithms face is the possible incorrect execution of the assertions. A possible scenario is, for example, a control flow error that just skips the whole correctness test, or at least the last “if” clause, and jumps directly to the “write” instruction where the matrix is sent to disk.

Standard systems have few error detection methods: essentially behavior based methods like memory protection and exception traps, with an unsatisfactory coverage. There are software based error detection methods (thus usable on standard systems where no specific hardware support exists) with very high coverage, like double execution and assembly-level software-based control flow checking, but they have prohibitive execution time overhead for most applications.

Fortunately, ABFT assertions typically account for a very small percentage of the total execution time (see table II). Since we just have to protect the execution of the correctness test, high overhead (but effective) software based error detection methods can be applied, without any significant overall performance degradation.

To protect the execution of the assertions a very simple software-based control flow checking technique can be used [8] [33]. To understand it, it should be noted that whoever is going to read the output file will want to have some mechanism to both know that the matrix is not corrupted (the CRC gives that assurance), and that

the matrix was indeed checked by the correctness test. For this second guarantee we propose the double execution of the correctness test and the use of a “magic” number, written together with the matrix in the output file, such as the sum of two unrelated numbers. This magic number, initially set to zero, will be assembled along the execution of the assertions in such a way that there is a high probability that at the end, it just holds the right number if all the assertions were successfully executed. If the first execution of the correctness test is successful, the first partial number is added to it; if the second execution of the test is also successful, the second partial number is also added to obtain the magic number. The resulting number is then written to disk together with the result matrix, using the same “write” statement. We call this carefully protected piece of code “gold code” (see figure 2).

When the program using the output file reads it, it just verifies that the correct magic number is in the right place in that file (it is independent of the contents of the matrix). At the end of the acceptance test the magic number variable is again set to zero. The existence of the magic number should be as short as possible to avoid an erroneous usage of it. The partial numbers associated with each assertion can be assigned freely (see [8]). For the purpose of this work we just chose numbers such that no number is the sum of a set of the other used numbers. Instead of adding the partial numbers to obtain the magic number an XOR operation can be used as suggested in [34].

| Matrix Dimension | execution time without ABFT (seconds) | execution time of standard ABFT |              | execution time of correctness test |              |
|------------------|---------------------------------------|---------------------------------|--------------|------------------------------------|--------------|
|                  |                                       | (seconds)                       | Overhead (%) | (seconds)                          | overhead (%) |
| 200              | 27.1                                  | 27.7                            | 2.21%        | 0.183                              | 0.66%        |
| 500              | 439                                   | 442                             | 0.68%        | 1.033                              | 0.23%        |
| 1000             | 3551                                  | 3563                            | 0.34%        | 4.150                              | 0.12%        |

Table II - Execution times of matrix multiplication on a SparcStation 2.

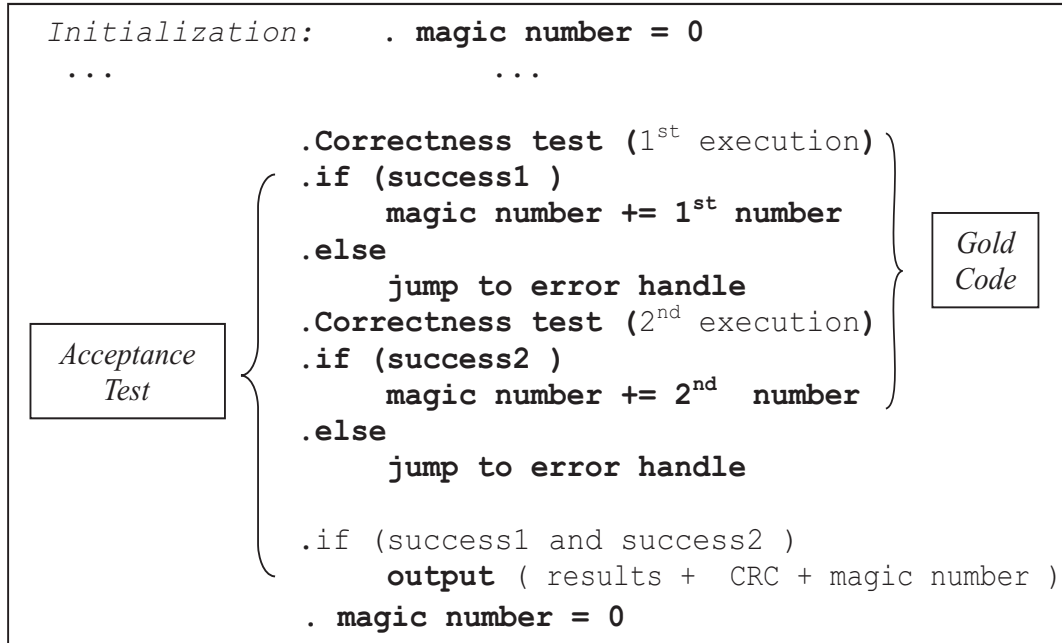


Figure 2 – The “Gold Code”

It is the simultaneous use of both the end-to-end protection of data and the gold code that we call robust ABFT. In order to determine the effectiveness of this method, we have injected faults in a third sequential version of matrix multiplication now using robust ABFT. Again the injected faults were randomly distributed during all the program execution time. As can be seen from table III, this version never outputted wrong matrices outside the tolerance margin. It is interesting to refer that, for the Robust ABFT version, of the 4929 detected errors, 60.2% (30.4% of the total injected faults) were detected by the assertions and the remaining 39.8% (20.1% of the total) were detected by the execution system (most of them correspond to illegal memory accesses detected by the system memory protection). The 60.2% of errors detected by the assertions can be further subdivided among 3.3% that were detected by the data integrity assertion at the beginning of the program, and thus correspond to data corruption during input, 56.6% that were detected by the first execution of the correctness test, 0.3% that were detected by the second execution, and

none was detected by the final assertion that tests if both executions were successful. As expected, the second execution is needed very seldom, and the final test has to be responsible for the detection of bad results only in very rare cases, that never occurred in our experiments. Finally, the upper limit estimated for the non-coverage of errors greater than the tolerance value,  $\bar{C}$  for this version of robust ABFT, is 0.05% with a confidence level of 99%.

Table IV summarizes the results obtained for standard and robust ABFT versions showing the distribution of detected errors and the upper 99% confidence limits for the non-coverage of both mechanisms.

At this point it should be clear that the “gold code” structure can be used to protect any kind of assertion, including assertions on the input data. A completely robust version of ABFT should also follow the gold code structure when it verifies the data integrity of the inputs. The generalization of the gold code structure to input assertions will be presented in section 3.

| Matrix Multiplication<br>with robust ABFT | robust ABFT   |             |
|---|---------------|-------------|
|   | no. of faults | % of total  |
| Correct output                            | 4454          | 45.6 %      |
| No output (crash)                         | 237           | 2.4 %       |
| No output (error detected)                | 4929          | 50.5 %      |
| Wrong output (within tolerance)           | 148           | 1.5 %       |
| Wrong output (out of tolerance)           | <b>0</b>      | <b>0.0%</b> |
| TOTAL                                     | 9768          | 100.0%      |

Table III - Behavior of matrix multiplication with robust ABFT.

Although we did not protect the input assertions in the experiments presented in table III, none of the results were out of tolerance. Those results point to a very small probability that a fault corrupts the data and simultaneously causes the incorrect acceptance of this data. But the input data can already be corrupted and due to a second fault the integrity assertion may not detect the error. The protection of the input assertion was done in the case of the gás-fired furnace controller described in the end of the next section and in [35].

into  $2n$  components. As before, the variable associated with the magic number is initially set to zero and then the magic number will be assembled along the execution of the assertions. Generally, for each assertion  $i$ , if the first execution is successful the  $(2i - 1)^{\text{th}}$  component of the magic number is added to the variable and if the second execution is successful the  $(2i)^{\text{th}}$  component is also added to the same variable (fig. 3).

Additionally, when deciding on the validity of a par-

| Matrix multip.<br>with: | Detected errors            |                        |                           |                      | Wrong outputs<br>(out of the tolerance) |   |
|-------------------------|----------------------------|------------------------|---------------------------|----------------------|---|---|
|                         | total of inject.<br>faults | % of detect.<br>errors | by the<br>exec.<br>system | by the<br>assertions | estimate<br>of $\bar{C}$                | $\bar{C}$ upper 99%<br>confidence limit |
| <b>Standard ABFT</b>    | 8645                       | 51.7%                  | 20.6%                     | 31.1%                | 1.8%                                    | 2.14%                                   |
| <b>Robust ABFT</b>      | 9767                       | 50.5%                  | 20,1%                     | 30,4%                | 0.0%                                    | 0.05%                                   |

Table IV - Comparison between Standard and Robust ABFT

### 3. ROBUST ASSERTIONS

The techniques we have used to make ABFT robust are quite general and can be applied to any system where the programmer defines assertions on the output. Let us consider a general block of code that receives some data, which can be validated by one or more input assertions, performs some calculations over this data, and produces results whose correctness can be evaluated by one or more assertions. As in robust ABFT, we propose an end-to-end protection of the data, and gold code protection for all the assertions performed in such blocks of code.

Supposing that there are  $n$  assertions, then each one is executed twice and the magic number is decomposed

into a particular output, the assertions can use the values of the inputs and possibly other internal state variables. With ABFT, the process of encoding the input data via checksums also works as a protection of the values that will be used to check the results. While that codification assures a low probability that errors in the checksum vectors will mask independent errors in the results, when considering general assertions we have no such guarantee. Inputs (and other values) used in assertions must be protected in a similar way to the outputs, and verified at the end, if we want to rely on the results of the assertions.

Thus for a general Robust Assertions technique we propose the following structure, illustrated in figure 3:

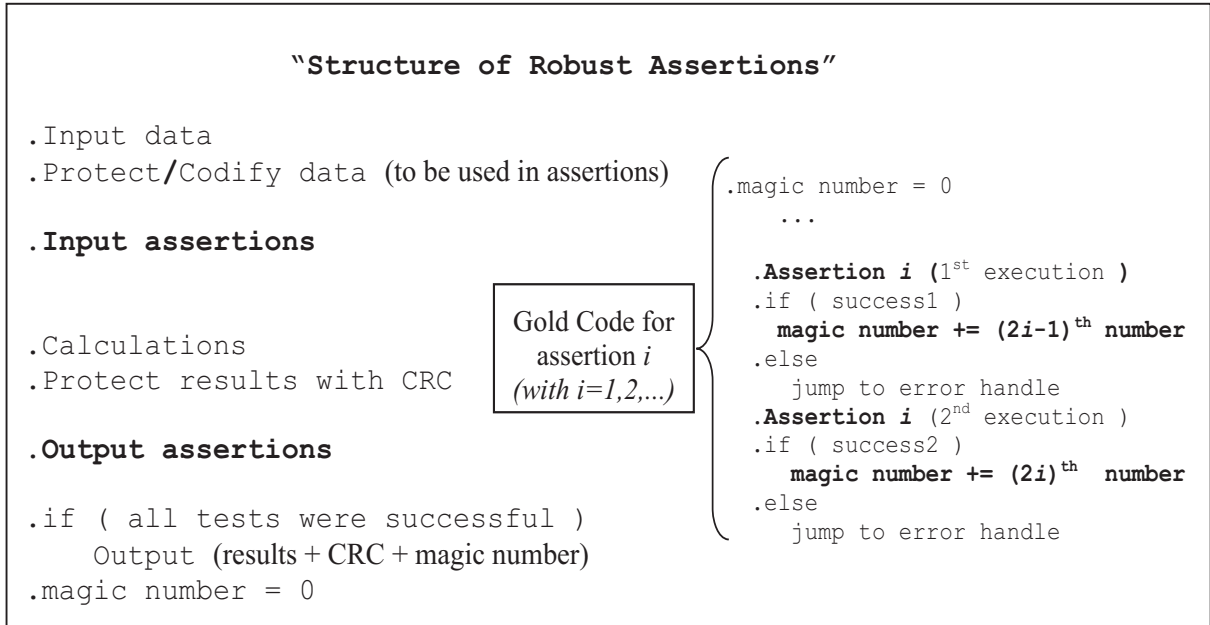


Figure 3 – Robust Assertions



The program, after inputting the data, protects (or codifies) all the data that will be used in any assertion. Next, if applicable, assertions on the input are executed following the gold code structure. Input assertions may include: 1- verifying a magic number in the case that the data was previously verified by a robust assertion; 2- verifying the data integrity in the case that the data was previously protected; 3- other application specific assertions used to validate the inputs.

The program proceeds with the main calculation and, at the end, protects the results with a CRC. Finally, it executes the output assertions following again the gold code structure. Output assertions can be: 1- application specific assertions used to check the correctness of the output; 2- verification of input data (and possibly other values) used in previous assertions.

If the results pass all the tests, then they are output together with the CRC and the magic number. Afterwards, the magic number variable is again set to zero.

It could be argued that, under the single fault assumption that was followed in these experiments, the protection of the correctness test is not relevant, because either the computation is affected or the test, but not both. Unfortunately, we cannot assume that, because we have no error containment regions. In the course of our experiments we have witnessed cases where both were affected. In many such cases the program execution is changed so strongly that the system crashes and does not output anything, but in some cases wrong data does come out. This is particularly true when the main calculation is very simple, as is often the case in feedback control systems. Still, it is true that under a single fault the probability of that event is low. But, since the time cost of the gold code is usually so low, it is worth the additional assurance it gives. This additional coverage may be small but it is relevant, particularly when the system has above 99,9% coverage, as is the case for instance in the experiments summarized in the table III.

There are two additional reasons to protect the acceptance test even under a single fault model. First, assertions can also detect errors due to software faults, and second, assertions can detect if a correctly designed and implemented algorithm cannot solve an ill conditioned problem with the required accuracy [14]. Each of these cases together with a single hardware fault affecting the acceptance test can lead to wrong outputs outside the tolerance margin. Again, although such compound events have low probability, the very low time overhead of the gold code is a small price for the additional assurance it gives.

On the other hand, there are cases where the single fault assumption is not realistic. A clear example is space applications subject to cosmic radiation. There, the sce-

nario where several heavy-ions hit the system during a single execution of a lengthy execution must be considered - see for instance the NASA REE project [14]. Double faults are realistic also in programs that manipulate large amounts of data spending several days or even weeks in the same computation.

If a single fault model is acceptable and the overhead of the gold code is too high, then a simplex execution of the correctness test can be considered, but will have to be validated by detailed fault injection experiments to ensure that the loss of coverage is not significant.

To test the Robust Assertions technique we used a control application, namely a small industrial controller for gas-fired pottery furnaces. On the one hand it is typical of the applications where assertions are common, and on the other hand we had available the full source code of a commercial controller, developed by the authors around 1990 and produced and sold by an industrial firm for some years after that. The "robust assertions" technique was fully applied to that real PID controller and an exhaustive study characterizes its behavior under fault injection. After almost fifty thousands injected faults the upper limit estimated for the non coverage of the robust controller is 0.009% with a confidence level of 99%. The execution overhead is 13.83% for 50 000 iterations and 13.76% for 150 000 iterations. As the main program is very simple, the probability of false alarms due to the introduction of the robust assertions structure is slightly higher than in the ABFT case but, as discussed before, the probability that a fault affects simultaneously the data and the correctness test is also higher justifying the protection of the assertions. We get a system with an error coverage above 99,99% and we think that this result clearly supports the usefulness of the robust assertion technique. For want of space the details of that work are not presented here but they are fully described in the previous conference paper [35].

#### 4. BOUNDED FAILURE

Previous research has shown that it is not possible to build fail-silent systems using only low-cost and low-overhead behavior based error detection methods in simplex systems subject to hardware faults [5] [6]. Although few, some erroneous results do manage to get out. We are then forced to consider that those systems follow the next less restrictive failure model, the Byzantine failure model. Still, usage of that model with those systems makes us feel quite uncomfortable since, after having examined many of those erroneous results, we see that they are not arbitrary, much less malicious. An intermediate failure model might be of interest in many situations. Based on above reported research we propose one new model, the Fail-Bounded model. Subsection 4.1 defines the model and subsection 4.2 justifies its need and applicability.

#### 4.1. DEFINITION

If we can guarantee that the results of a computation in a simplex computer are always filtered by an acceptance test, or assertion, and that the outputs are not corrupted after being tested by that assertion, (in which case we call it a robust assertion), we know that the output, even if wrong, satisfies the invariant tested by that assertion. The ensuing failures are thus not arbitrary, since the error they exhibit has an upper bound that depends on the output assertions used by the programmer. The notion of distance implicit in the concept of "upper bound" is very general; the associated metric is completely dependant on the type of assertion used, and has essentially two values: satisfies the assertions, or does not satisfy the assertions.

This behavior defines a new failure model that we call Fail-Bounded:

**Definition 1:** A system is said to be Fail-Bounded if it either:

- a) produces correct output;
- b) produces wrong outputs, but the error they exhibit has an upper bound defined by output assertions whose execution is guaranteed;
- c) stops producing output (because of a crash or because an error was detected).

The Fail-Bounded failure model is an intermediate model between Fail-Silent and Byzantine Failure:

- The Fail-Bounded model becomes the Fail-Silent model as the error bound becomes zero;
- The Fail-Bounded model becomes the Byzantine model (in the value-domain) as the error bound becomes arbitrarily large;

The assertions we consider here can be of any kind. They can be like the acceptance tests of recovery blocks [36], but they can also be simple likelihood tests. Obviously, the stronger the test the smaller is the distance between an erroneous output and the correct value, and the more usable will be the system. For a further discussion and examples of assertions see for instance [13], [37], [38], [39], [40] and the recent papers on software robustness [41], [42], [43] [44].

#### 4.2. JUSTIFICATION

A certain fuzziness is inevitably associated with the Fail-Bounded model, since we do not guarantee that the outputs are exactly correct, but only that they fall within an interval around the correct value. We contend that this is a realistic and usable assumption in several classes of systems.

The first type of such systems is number crunching. People working with numerical algorithms are already used

to that fuzziness, because of the finite precision of floating point arithmetic. This is exactly what we have seen before for the ABFT case, where the round-off error required us to work with intervals of correctness, not exact values. There is even a branch of mathematics called interval arithmetic, where calculations with intervals are studied (see for instance the chapter on "Interval Arithmetic" in [45]). Clearly, depending on the assertions used, the size of the interval around the correct value where wrong results fall can be close to the round-off interval, or it can be substantially larger. Finding assertions that lead to error intervals sufficiently small to be usable is an open problem for many numerical calculations. For those algorithms for which an ABFT has been developed, or a Result Checker [46] exists, this is essentially a solved problem.

A second class of systems where Fail-Bounded is a natural notion is feedback control, like the control application referred in the previous section [35]. There we show that the physical restrictions of such systems can be easily used to define meaningful assertions, and that the Robust Assertion technique is easy to apply and effective. Indeed, in the control of continuous time systems, a certain level of error is unavoidable, similarly to round-off error in floating point arithmetic. In fact, control algorithms are designed precisely to enable the controlled system to go on working correctly in spite of external disturbances - if these didn't exist, controlling a system would involve no feedback, just a device to inform the controlled system of the desired set point. We claim that, if the error is not too large, an erroneous output of the controller can be considered just as another "external disturbance" that will be compensated for by the feed-back control mechanism. We have studied this question extensively in [33], showing it to be true. Similar results were obtained in [47] [48]. We thus claim that Fail Boundedness fits very well the behavior of feedback control systems, is very usable and easy to implement.

A third class of systems where Fail-Bounded is meaningful is discrete state systems. For this class, work to prove the feasibility of the model is not so advanced. An example that can help clarify the idea is the control of the traffic lights at a street crossing. If the safety restrictions are met (e.g., no two crossing lanes have a green light at the same time, and lights do not change too quickly), it is not relevant if the standard sequence by which each incoming traffic flow gets green is changed once in a while. This means that the system can accommodate changes (errors) in the normal state sequence, as long as the safety restrictions are met.

Another relevant question at this point, having established the relevance of the Fail-Bounded model for several system classes, is its usability. Can dependable systems of practical relevance be built on building blocks that follow this model? In [33] a positive answer is given to

this question for feedback control systems. In the first part of this paper a similarly positive answer is given for floating point calculations. And for distributed systems at large? Both the Fail-Silent model and Fail-Byzantine model have proven their worth since many distributed algorithms have been built on them. Can the same be expected from the Fail-Bounded model? While this is largely an open research problem, we can argue that, in many cases, algorithms meant to be used with the Fail-Silent model can also be applied to systems with Fail-Bounded behavior. In fact, the previous description shows that in many circumstances the output error due to a fault in the system is indistinguishable from other sources of error the system is prepared to deal with, or at least the output deviation is not relevant. If the erroneous outputs resulting from a fault (erroneous in the sense that if fault free the system would have produced different outputs) do not fall outside those intervals, those outputs can be considered correct. Under those conditions, the Fail-Bounded model then becomes the same as the Fail-Silent model, and all algorithms that assume Fail-Silence can be applied to Fail-Bounded systems.

We cannot obviously guarantee that any system will follow the Fail-Bounded model with 100% certitude, but the same is true for all other failure models, except for the Byzantine model. Still, the results presented for robust ABFT and those obtained for the robust control application [35] show that the coverage of the Fail-Bounded assumption, in the sense of [49], can be quite high, thus making it realistic and usable.

## 5. CONCLUSIONS

This paper addressed in the first place the problem of determining the behavior of ABFT techniques under a more general fault model than has been used to calculate its coverage in the original papers, and essentially uncovered two weakpoints: data can be corrupted during input/output and the correctness tests can be incorrectly executed. In the same line, and as has already been pointed out in other works, it is also a problem that the correctness tests cannot be exact due to rounding errors in floating point calculations. We proposed simple ways to solve the first two problems, leading to what we collectively called Robust ABFT, and showed by fault injection that in spite of their simplicity they are very effective. The problem of rounding errors cannot be solved, meaning that we can only speak of the correctness of the result within a certain error tolerance.

We then proposed a generalization of the Robust ABFT technique to Robust Assertions, where we essentially try to guarantee that all data output by a system has been previously filtered by a correctly executed assertion, with high probability. Results obtained with a PID controller subjected to fault injection have proved that such a mechanism is feasible and effective.

This motivated the main proposal of this paper, namely a new failure model called Fail Bounded, somewhere between the Fail-Silent and the Fail-Byzantine models, according to which a system either outputs correct results, or stops because of a crash or upon detection of an error, or outputs results that are wrong but within a certain "distance" of the correct outputs, where this distance is defined by the output assertions whose execution is guaranteed with high probability.

Although we argued that in many circumstances fail-bounded systems can be handled as fail silent systems, it is still an open problem to determine how usable fail-bounded systems can be as building blocks for dependable systems, distributed or not.

## ACKNOWLEDGEMENTS

This work was partially supported by the Portuguese Ministério da Ciência e do Ensino Superior, the FEDER program of the European Union through the R&D Unit 326/94 (CISUC) and the Group of Networks and Multimedia of the Institute of Telecommunications - Covilhã Lab, Portugal, and the project POSI/1625/95/2001 (PARQUANTUM). The present paper is an extended evolution of the conference paper: J.G. Silva, P. Prata, M. Rela and H. Madeira "Practical Issues in the Use of ABFT and a New Failure Model," in 28th Int'l Symposium on Fault-Tolerant Computing, Germany, 1998, pp. 26-35.

## REFERENCES

- [1] A. Campbell, P. McDonald and K. Ray. Single Event Upset Rates in Space. *IEEE Trans. on Nuclear Science*, 39(6): 1828-1835, 1992.
- [2] C. Constantinescu. Impact of Deep Submicron Technology on Dependability of VLSI Circuits. In *Proc. Int'l Conf. on Dependable Systems and Networks*. Pages 205-209, 2002.
- [3] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proc. Int'l Conf. on Dependable Systems and Networks*. Pages 389-398, 2002.
- [4] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton. The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In *Proc. 18th Int'l Symp. Fault-Tolerant Computing*. Pages 246-251, 1988.
- [5] H. Madeira and J. G. Silva. Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking. In *Proc. 24th Int'l Symp. Fault Tolerant Computing Systems*. Pages 350-359, 1994.

- [6] M. Z. Rela, H. Madeira, and J. G. Silva. Experimental Evaluation of the Fail-Silent Behavior of Programs with Consistency Checks. *In Proc. 26th Int'l Symp. Fault-Tolerant Computing*. Pages 394-403, 1996.
- [7] J. G. Silva, J. Carreira, H. Madeira, D. Costa, and F. Moreira. Experimental Assessment of Parallel Systems. *In Proc. 26th Int'l Symp. Fault-Tolerant Computing*. Pages 415-424, 1996.
- [8] A. Mahmood and E. J. McCluskey. Concurrent Error Detection Using Watchdog Processors - A Survey. *IEEE Trans. Computers*, 37(2): 160-174, 1988.
- [9] K. Wilken and J. P. Shen. Continuous Monitoring: Low-Cost Concurrent Detection of Processor Control Errors. *IEEE Trans. on Computer-Aided Design*, 9(6): 629-641, 1990.
- [10] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, J. A. Abraham. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. *IEEE Trans. on Parallel and Distributed Systems*, 10(6): 627-641, 1999.
- [11] G. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson. Use of Time and Address Signatures for Control Flow Checking. *5th Int'l IFIP Conference on Dependable Computing for Critical Applications (DCCA-5)*, ISBN 0-8186-7803-8, IEEE Computer Society Press, February 1998.
- [12] A. Steinger and C. Scherrer. On Finding An Optimal Combination Of Error Detection Mechanisms Based On Results Of Fault Injection Experiments. *In Proc. of the 27th Int'l Symp. on Fault-Tolerant Computing*, IEEE Computer Society Press, 1997.
- [13] A. Mahmood, E. J. McCluskey, and D. J. Lu. Concurrent Fault Detection Using a Watchdog Processor and Assertions. *In Proc. Int'l Test Conference*. Pages 622-628, 1983.
- [14] M. Turmon, R. Granat, and D. S. Katz. Software-Implemented Fault Detection for High-Performance Space Applications. *In Proc. 30th Int'l Conf. on Dependable Systems and Networks (FTCS-30 & DCCA-8)*. Pages 107-116, 2000.
- [15] K.-H. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Computers*, c-33(6):518-528, 1984.
- [16] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Prog. Lang. Syst.* 4(3):382-401, 1982.
- [17] P. Banerjee, J. T. Rahmed, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J.A. Abraham. Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor. *IEEE Trans. Computers*, 39(9): 1132-1144, 1990.
- [18] A. R. Chowdhury and P. Banerjee. Algorithm-Based Fault Location and Recovery for Matrix Computations. *In Proc. 24th Int'l Symp. Fault-Tolerant Computing*. Pages 38-47, 1994.
- [19] Y.-H. Choi and M. Malek, A Fault-Tolerant FFT processor. *IEEE Trans. Computers*, 37(5): 617-621, 1988.
- [20] A. R. Chowdhury and P. Banerjee. Compiler-Assisted Generation of Error Detection Parallel Programs. *In Proc. 26th Int'l Symp. Fault-Tolerant Computing*. Pages-360-369, 1996.
- [21] P. Banerjee and J. A. Abraham. Bounds on Algorithm-Based Fault Tolerance in Multiple Processor Systems. *IEEE Transactions on Computers*, c-35(4): 296-306, 1986.
- [22] B. Vinnakota and N. K. Jha. Design of Algorithm-Based Fault-Tolerant Multiprocessor Systems for Concurrent Error Detection and Fault Diagnosis. *IEEE Trans. Parallel and Distributed Systems*, 5(10): 1099-1106, 1994
- [23] S. Yajnik and N. K. Jha. Graceful Degradation in Algorithm-Based Fault Tolerant Multiprocessor Systems. *IEEE Trans. Parallel and Distributed Systems*, 8(2): 137-153, 1997.
- [24] R. K. Sitaraman and N. K. Jha. Optimal Design of Checks for Error Detection and Location in Fault-Tolerant Multiprocessor Systems. *IEEE Trans. Computers*, 42(7): 780-793, 1993.
- [25] J. Carreira, H. Madeira, and J. G. Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Trans. Software Eng.*, 24(2): 125-135, 1998.
- [26] P. Duba and R. K. Iyer. Transient Fault Behavior in a Microprocessor: A Case Study. *Presented at ICCD*. Pages 272-276, 1988.
- [27] ANSI/IEEE. IEEE Standard for Binary Floating-Point Arithmetic, 1985.
- [28] Motorola. PowerPC 601 Risc Microprocessor user's Manual, 1993.
- [29] A. R. Chowdhury and P. Banerjee. Tolerance Determination for Algorithm-Based Checks Using Simplified Error Analysis Techniques. *In Proc. 23rd Int'l Symp. Fault-Tolerant Computing*. Pages 290-298, 1993.
- [30] P. Prata. High Coverage Assertions. PhD Thesis, Universidade da Beira Interior, Portugal, 180 pages, September 2000.
- [31] D. Powell, M. Cukier, and J. Arlat. On Stratified Sampling for High Coverage Estimations. *In Proc. 2nd European Dependable Computing Conference*. Pages 37-54, 1996.
- [32] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Trans. Computer Systems*, 2(4): 277-288, 1984.
- [33] J. Cunha, R. Maia, M. Z. Rela, J. G. Silva. A Study of Failure Models in Feedback Control Systems. *In*

- Proc. The Int'l Conf. on Dependable Systems and Networks (DSN-2001)*. Pages 314-323, 2001.
- [34] N. Oh, P. P. Shirvani and E. J. McCluskey. Control Flow Checking by Software Signatures. *In IEEE Trans. on Reliability*, 51(1), 2002.
- [35] J. G. Silva, P. Prata, M. Z. Rela and H. Madeira. Practical Issues in the Use of ABFT and a New Failure Model. *In Proc. 28th Int'l Symposium on Fault-Tolerant Computing*. Pages 26-35, 1998.
- [36] B. Randell. System Structure for Software Fault-Tolerance. *IEEE Trans. Software Eng.*, SE-1(2): 220-232, 1975.
- [37] D. M. Andrews. Using Executable Assertions for Testing and Fault Tolerance. *In Proc. 9th Int'l Symp. Fault-Tolerant Computing*. Pages 102-105, 1979.
- [38] B. McMillin and L. M. Ni. Executable Assertion Development for the Distributed Parallel Environment. *In Proc. 12th Int'l COMPSAC*. Pages 284-291, 1988.
- [39] N. Leveson and T. J. Shimeall. Safety Assertions for Process-Control Systems. *In Proc. 13th Int'l Symp. Fault-Tolerant Computing*. Pages 236-240, 1983.
- [40] A. Watanabe and K. Sakamura. Design Fault Tolerance in Operating Systems Based on a Standardization Project. *In Proc. 25th Int'l Symp. on Fault-Tolerant Computing*. Pages 372-380, 1995.
- [41] N. P. Kropp, P. J. Koopman and D. P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. *In 28th Int'l Symposium on Fault-Tolerant Computing*. Pages 230-239, 1998.
- [42] R. A. Maxion and R. T. Olszewski. Improving Software Robustness with Dependability Cases. *In 28th Int'l Symposium on Fault-Tolerant Computing*. Pages 346-355, 1998.
- [43] J. Pan, P. Koopman, D. Siewiorek, Y. Huang, R. Gruber and M. L. Jiang. Robustness Testing and Hardening of Corba ORB Implementations. *In Int'l Conference on Dependable Systems and Networks*. Pages 141-150, 2001.
- [44] C. Fetzer and Z. Xiao. HEALERS: A Toolkit for Enhancing the Robustness and Security of Existing Applications. *In Int'l Conference on Dependable Systems and Networks*, 2003.
- [45] N. Higham. Accuracy and Stability of Numerical Algorithms. SIAM, 688 pages, 1996.
- [46] H. Wasserman and M. Blum. Software Reliability via Run-Time Result-Checking. *Journal of the ACM*, 44(6): 826-849, 1997.
- [47] J. P. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson. Reducing Critical Failures for Control Algorithms Using Executable Assertions and Best Effort Recovery. *In Proc. Int'l Conf. on Dependable Systems and Networks*. Pages 347-356, 2001.
- [48] J. P. Vinter, A. Johansson, P. Folkesson, and J. Karlsson. On the Design of Robust Integrators for Fail-Bounded Control Systems. *In Proc. Int'l Conf. on Dependable Systems and Networks*. Pages 415-424, 2003.
- [49] D. Powell. Failure Mode Assumptions and Assumption Coverage. *In Proc. 22nd Int'l Symp. Fault-Tolerant Computing*. Pages 386-395, 1992.