Detecting and Correcting Errors in Functional Units Performing Composable Operations

Lou Scheffer
Cadence

ABSTRACT

In the operation of a DSM chip, there is the possibility of transient errors. This paper proposes a new way to detect and/or correct such errors. If we must do N identical composable operations, we can detect errors by doing 1 additional similar operation, and both detect and correct errors by performing about $log_2(N)$ additional operations. For example, suppose an algorithm requires performing 1000 FFTs. With one additional FFT, we can verify that all FFTs were performed correctly. With 10 additional FFTs (performed on various linear combinations of the input data) we can detect which, if any, FFT was wrong, and compute the correct answer without re-doing the incorrect computation. This result holds whether the results are computed in one cycle or many, sequentially or in parallel, or in hardware and software.

Categories and Subject Descriptors

J.6 [**Computer Applications**]: Computer Aided Engineering

General Terms

Algorithms, Performance, Design, Verification

Keywords

XXX XXX, XXX vXXX, SXXX tXXX, YXXX

1. INTRODUCTION AND MOTIVATION

In the operation of DSM chips, there is the possibility of transient errors. These are most commonly caused cosmic rays, alpha particles, or neutrons that impinge upon the chip and cause transient data errors or upset the state of one or more flip-flops. This is commonly called Single Event Upset, or SEU. Not surprisingly, this problem is most common in systems exposed to radiation (such as space based systems) but occur (more rarely) even at ground level[6]. As dimensions scale down this problem will become worse since smaller and smaller disturbances can cause these problems, and some sources (such as neutrons) cannot be eliminated by any practical amount of shielding. We would like a way to detect, and if possible correct, such errors. many such methods have been proposed, but they require significant overhead. In this paper we propose a cheaper method of performing such detection and correction.

We start by observing that if we perform N identical operations on different pieces of data, we may be able to detect an error by performing the equivalent of a checksum. The critical property is composability, which allows us to check the results of N operations by doing one additional operation. The classic composable operation is a linear one where $F(a+b) = F(a) + F(b)$. This implies $F(a+b+c) = F(a) + F(b) + F(c)$, and so on. Thus one additional $F()$ operation can be used to check the results of any number of $F()$ operations. More generally, we require two (possibly identical) operations $\oplus$ and $\otimes$ such that $F(a \oplus b) = F(a) \otimes F(b)$. An example of a non-linear but composable function is exp() since $\exp(a+b) = \exp(a) \cdot \exp(b)$.

Are enough operations composable to make this approach worthwhile? The answer depends on the application, of course, but in many cases it seems true. Composable operations include any linear operation and a significant number of other mathematical operations. All linear operations are composable, including such common operations such as FFTs, DCTs, wavelet transforms, and many matrix operations. Speech encoding and adaptive optics are dominated by FFTs, a linear operation. Video encoding is full of DCTs (discrete cosine transforms). MPEG-2 spends up to 35% of its time in DCTs[9, 7]. Decomposition into wavelets is now common in many encodings. Many multimedia operations apply a given digital filter to many samples. All of these are linear operations, and hence composable.

1.1 Previous work

It is well known[1] that almost all commonly used boolean codes can be extended to work when the symbols are real or complex numbers instead of binary digits [5]. This work extends this idea by replacing the channel with arbitrarily complex operations, provided they are composable.

Another closely related work is on Algorithm Based Fault Tolerance (ABFT), introduced by Huang and Abraham[4]. As the name implies, this is intended to protect the execution of a single algorithm by adding 'checksums', usually linear functions of the input. These checksum values are

[1]Among a small circle of specialists, that is.

ICCAD 2003 November 9-13, 2003, San Jose, California, USA.

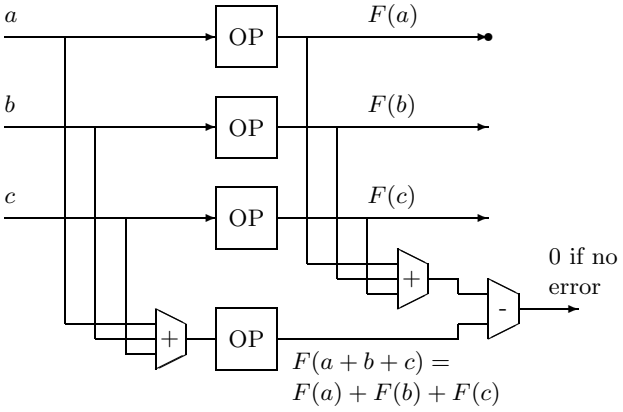

Figure 1: Example of real valued parity check

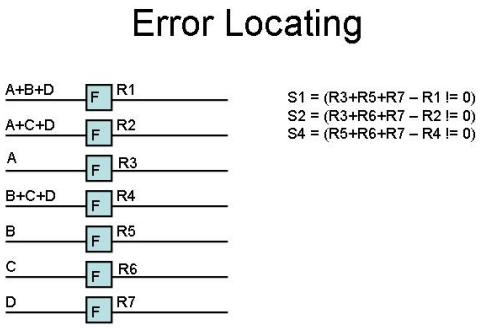

Figure 2: Error Correcting code for Composable Operations

predictably transformed by the operation, and can therefore be used to detect and possibly correct errors. Matrix multiplication was the first algorithm this was applied to, but research is actively extending the set. The REE group at JPL, for example, is looking at techniques for numerical linear matrix multiplication, LU decomposition, QR decomposition, single value decomposition (SVD), and Fast Fourier Transforms (FFT).

Beckmann[2] and Hadjicostis[3] generalize these methods to protect operations that are operations over groups, rings and fields. Since the real numbers form a field, a vector of real numbers under component-wise operations will also be a field. In this way the work of this paper can be viewed in these more theoretical frameworks.

The current work differs from previous work in ABFT in two ways. We try to correct not a single operation, but N identical operations, all working on different data. Furthermore, there is no need to design the checksum for a particular algorithm. This technique works with any algorithm, as long as it is composable.

2. THE IDEA

The first example is a parity check for composable operations. Suppose we have N complicated, but composable, operations, to be done on independent data. In this example we will use FFTs as an example of such an operation, though any composable operation will work. We wish to verify that all N operations were done correctly. We can detect any single error with one additional FFT. We sum all the inputs of all the FFTs, and tranform that. After the FFTs, we add all the spectra and compare the sum to the transform of the sum of the inputs. By linearity, the two results should be equal. This is shown in Figure 1.

If the sum does not match, one of the FFTs was wrong, though we have no idea which one (in fact, all the data might be right, and only the checksum FFT is wrong). In the binary case, we can detect all errors where an odd number of the entries are wrong, and will falsely claim no error in the cases where an even number of entries are incorrect. The linear system will presumably do better than this if the linear operations are real values, because (unlike binary) the odds of two errors cancelling in the checksum should be negligable.

With a little more work we can determine which FFT was bad and what the correct answer is. We need to add $\log_2 N$ check inputs, each picked from a subset of all possible input sums. If we pick them properly, the pattern of wrong answers tells precisely which operations were incorrect. One good encoding is shown in Figure 2. This is based on the Hamming code, with the XOR operations (addition MOD 2) replaced by addition over real or complex numbers. There are 4 data results and 3 check results, any of which might be wrong. The three checksums are evaluated with a wrong answer corresponding to a 1 and a right answer to a 0. (This is called a 'syndrome' in coding theory.) If all three are 0, then the answer is correct (or, less likely, a multiple cancelling error occured). If not all answers match, then these three resulting bits form a number in binary that is the index of the wrong bit.

For example, suppose that in figure 2, FFT 5 gave a wrong result. Then the syndrome computation gives $S1 = 1$, $S2 = 0$, and $S4 = 1$. Interpreting these as a binary number gives 5, which is the index of the bad FFT. Once we know this one is bad, we can recompute it as $R1 - R3 - R7$.

In the binary case, this code will not detect double errors - it will think it is an (incorrect) single error, and fix the wrong thing. The same will happen here. Any possible error must result in one of the eight possible output states - no error, or one of the 7 bits in error. This can be solved, as in the binary case, by adding an overall checksum. The added redundancy allows two or more errors to be (usually) detected. You simply correct the single error as usual and see of the corrected value now satisifies the overall parity checksum. If not, there was more than one error. As in the binary case, this will detect most multi-entry errors as well.

3. USES

The most obvious use is to protect against single event upset. If the application can deal with a certain amount of incorrect results, a checksum may be all that is needed. (Many audio or video applications fall into this class. If the data is known bad, interpolation or some other estimate can be used. If errors are rare this hurts the quality very little.) If the application needs all correct answers, then the more complex error correcting codes can be used.

Another use of this same principle is to protect against errors in distributed systems (Such as Seti@home). In SETI@home version 3.03, processing a work unit involves performing about 31,560 FFTs[1]. In SETI@home, work is distributed to a large number of not necessarily trustworthy computers. They can make unintentional errors, of course, buth there is a worse problem - cheating. Since there is a contest with fame and fortune for the group who does the most work, some people cheat and return a null result without actually doing the work. Since the great majority of work units will return a null result anyway, this is very difficult to spot. The techniques here provide an easy way to verify that the users have at least done the linear portion of the task (all the FFTs), without taking much compute power or bandwidth.

For example, we can calculate a signature by summing all the bins of all the outputs of all the FFTs. This is a linear function of the inputs. So if we want to check 1000 work units for accuracy, we can sum the 1000 inputs, do the FFTs and sums on that. For the jobs done remotely, we each returns the signature. We sum these, and if it matches the signature of the summed inputs, all 1000 were done correctly. If we are willing to do 10 extra FFTs, (each over a different input subset), then if we get an error we can figure out which one was bad, and what the result should have been. With 11 extra FFTs, we can also detect the cases where 2 or more computations are in error.

4. PRACTICAL CONSIDERATIONS

Most of the operationw we might try to protect are not exactly linear. Most floating point operations, for example, involve at least rounding. As a result of these non-linarities, the exact cancellation of Fig. 1 will not happen in practice.

Take, for example, the FFT of the previous examples. This will be slightly non-linear. Therefore we cannot say we have an error if the checksum is not exactly 0. We need to set a threshold, which we expect no normal operation to cross, that defines an error. If the data are independent, then the non-linarities can be expected to be about sqrt(N) times bigger than the non-linearities of a single FFT. If we combine 16 FFts, for example, the moise will be about 4 times bigger. Therefore we will not see an error unless it causes an error in the output that is somewhat more than 4 times the inherent error of a single operation. In practice this means transient errors in the LSBs of computations may be missed since they are comparable in size to errors already committed by the algorithms.

Similarly, if we re-compute the spectrum of the erroneous FFT by taking combinations of the checksum FFTS and the correct values, we would expect errors about sqrt(N) times as big as we would have had if the calculation was error free.

If we are trying to determine if the operation was done at all, as in the SETI@home example, then the numerical situation is much better. For example, a single FFT has a numerical non-linearity of less than 10^{-5}. Therefore we could sum almost any number of such FFTs (up to about 10^{10} of them) and still detect if one of them was missing entirely.

5. EXPERIMENTAL RESULTS

The first experiment is based on Fig. 1. We take the checksum of 15 FFTs by taking a 16th FFT. We use the FFT from [8] with 256 complex points, using single precision floating point. In 1,000,000 trials without errors, the cancelation is good to a level of to a level of about $39 \cdot 10^{-6}$ (of the input size), so we set the threshold for detecting and error to $40 \cdot 10^{-6}$. With this threshold, no false errors were found in 1,000,000 error free runs. Next, we introduced a single error in a arbitrary bit of a arbitrary floating point number after an arbitrary pass of an arbitrary FFT. 1 million examples of such errors were tried, and measured for the detection of errors. The results are shown in figure 3. As expected, all MSB errors were detected and most LSB errors were not.

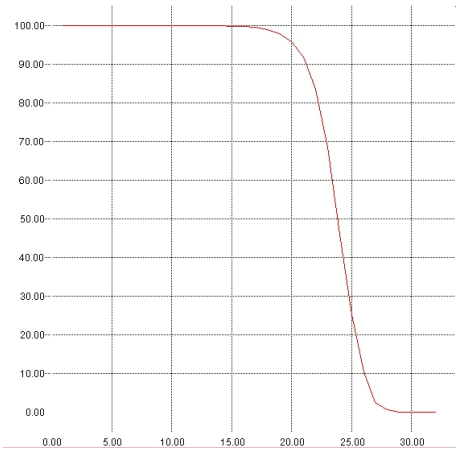

Figure 3: Detection rate vs bit number

Another experiments shows that we can detect one in a million not being done at all. Here we took 1M FFTs, and compared the sum of the FFTs with the FFT of the sum. Omitting even one of the FFTs from the sum results in an easily detectable error.

Further experiments show we can detect an error location and correct the spectrum, that the overhead of these operations are as expected, and that almost all multiple errors are detected. [Note to referees - these experiments are ongoing and much more detail can be provided if the paper is accepted].

6. CONCLUSIONS

This paper has shown how multiple independent (but composable) operations can be protected against errors with relatively low overhead. This protection can be simply the detection of errors, or can include correction of single errors and detection of multiple errors. The operations can be executed in series by a single unit, in parallel by N units, or in other combinations. The technique works in hardware or software.

7. REFERENCES

[1] http://ksetispy.sourceforge.net/manual/commands.html.

[2] P. Beckmann and B. Musicus. A group-theoretic framework for fault-tolerant computation. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP-92*, volume 5, pages 557–560, Mar 1992.

[3] C. N. Hadjicostis. *Coding approaches to fault tolerance*

in combinational and dynamic syst ems. Kluwer Academic Publishers, 2001.

[4] K. Huang and J. Abraham. Algorithm–based fault tolerance for matrix operations. *IEEE Trans. Comp.*, C–33:518–528, 1984.

[5] J. Marshall, T. Coding of real-number sequences for error correction: A digital signal processing problem. *Selected Areas in Communications, IEEE Journal on*, 2:381 –392, Mar 1984.

[6] E. Normand. Single event upset at ground level. In *IEEE Transactions on Nuclear Science*, volume 43, pages 2742–2750, Dec. 1996.

[7] K. Patel, B. C. Smith, and L. A. Rowe. Performance of a software mpeg video decoder. In *Proceedings of the first ACM international conference on Multimedia*, pages 75–82. ACM Press, 1993.

[8] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.

[9] R. Stockel. Feig's scaled 2-d dct now tested with the berkeley mpeg-player. http://rnvs.informatik.tu-chemnitz.de/jan/MPEG/HTML/idct_discussion/Index.html, 1998.