Using reversible computing to achieve fail-safety

Peter G. Bishop
Adelard
Coborn House, 3 Coborn Rd
London E3 2DA, UK
pgb@adelard.co.uk

Abstract

This paper describes a fail-safe design approach that can be used to achieve a high level of fail-safety with conventional computing equipment which may contain design flaws. The method is based on the well-established concept of "reversible computing".

Conventional programs destroy information and hence cannot be reversed. However it is easy to define a virtual machine that preserves sufficient intermediate information to permit reversal. Any program implemented on this virtual machine is inherently reversible. The integrity of a calculation can therefore be checked by reversing back from the output values and checking for the equivalence of intermediate values and original input values. By using different machine instructions on the forward and reverse paths, errors in any single instruction execution can be revealed. Random corruptions in data values are also detected.

An assessment of the performance of the reversible computer design for a simple reactor trip application indicates that it runs about ten times slower than a conventional software implementation and requires about 20 kilobytes of additional storage. The trials also show a fail-safe bias of better than 99.998% for random data corruptions, and it is argued that failures due to systematic flaws could achieve similar levels of fail-safe bias. Potential extensions and applications of the technique are discussed.

1. Introduction

Most practical computer-based safety systems rely on commercial hardware, such as processor chips, and supporting software, like compilers. These components have not been verified in any formal sense, and any inherent flaws could affect the system behaviour in unpredictable ways. A stronger safety case can be made if a fail-safe behaviour can be imposed on this "untrusted base".

In this paper we describe a novel approach to fail-safe design which is based on the concept of "reversible computing". We will describe the reversible computing concept, the implementation of the fail-safe design, and an evaluation of the performance of the technique when applied to a simple reactor trip application. In the final sections we discuss the practical applications of this technique and further areas of work.

2. Reversible computing

Early pioneers in the field of computing like Turing and Von Neumann examined the minimum energy needed for computation and concluded that each elementary operation would require an energy expenditure of at least $kT \ln 2$ where k is the Boltzmann constant and T is the absolute temperature. The argument here is that kT represents the background energy (Brownian motion) and any operation must exceed this to be distinguishable from the background noise. This theory was subsequently overturned by Landauer [7, 8] who showed that this was only true if the process was irreversible. If you could perform the computation, save the result and then reverse the computation to get the original inputs, no energy need be consumed (except for retaining the answer). Basically normal computation increases disorder (i.e. entropy) and is irreversible, but a reversible design does not increase disorder (the reverse action acts like a refrigerator). This fundamental concept was further developed by Bennett [1, 2, 3] who showed it was possible to construct a modified form of Turing machine which was reversible, and hence that any computation is potentially reversible. In a later development Fredkin [6] showed that it was possible to construct a "conservative logic gate" which was reversible but could be used to construct conventional AND and OR functions, so there is a general mechanism for constructing reversible logic circuits. These concepts have been used to minimise the heat dissipation in logic circuits [9, 5].

In our paper we have used the reversible computing concept to implement a novel form of self-checking which can be applied at the software level rather than the circuit level. The reversible computing concept can be illustrated by the following simple example. The "+" function maps the number pair $< 1, 3 >$ to a single value $< 4 >$. It is not possible to reverse this because this output value can be computed from several distinct input pairs (e.g. $< 0, 4 >$, $< 1, 3 >$ or $< 2, 2 >$). In a reversible computation the mapping is bi-directional (i.e. a one to one mapping). This can be achieved by generating "garbage data" as well as the required result. For example we could define a modified function (PLUS) that produces the required result together with one of the input operands (e.g. $< 1, 3 >$ maps to $< 4, 1 >$ where the second value is "garbage data"). The unknown input value (3) can be regenerated by subtracting the garbage data from the sum.

All the basic computing functions can be modified to generate the necessary garbage data to make them reversible. A conventional computing function can be represented as

$$y = f(x)$$

but a reversible function typically has the form:

$$< y, g >= f_r(x)$$

The function $f_r(x)$ produces the same y value as $f(x)$ but also produces the "garbage data" g required for reversal. There is an associated inverse function f_r^{-1} that uses the garbage data and computed value to regenerate the input values:

$$x = f_r^{-1}(y, g)$$

The behaviour of a reversible function is illustrated in Figure 1.

Providing the garbage data is saved, a program constructed from a sequence of reversible functions is also a reversible function. This suggests that the reversible computing concept could be used to implement a powerful and general form of fault detection. If a reversible computer function $f_r(x)$ and its inverse function $f_r^{-1}(y, g)$ are correctly implemented, then we would expect that:

$$x = f_r^{-1}(f_r(x))$$

would hold for any program execution. In other words, by reversing the computation and getting the original input values, we have greater confidence that the computed value y is correct. If the forward and reverse functions are diversely implemented, this should guard against systematic faults in the underlying "machine" (e.g. due to faulty software tools or hardware design flaws like the Pentium divide bug). If the flaw exists in one direction only, the failure should always be detected by a mismatched value. If failures can occur in both directions, it is difficult to calculate the detection probability, but both functions have to fail simultaneously and agree on the same wrong result. To be permanently undetected they have to agree on the same wrong answer for all error-inducing values of x.

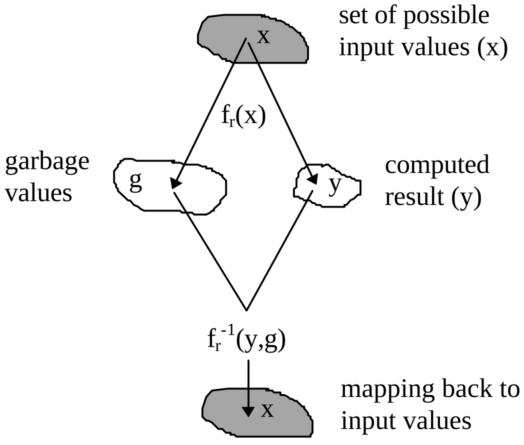

Figure 1. Illustration of reversible computing

The mechanism should also detect typical random data corruption faults that affect any data value used in the computation. It can also guard against simple application programming errors which result in finite precision overflow and underflow since the reverse computation will not be identical.

Reverse computing can be regarded as a form of low-level design diversity which does not suffer the usual problems of diversity (such as consistency checking and voting). It also has the advantage that the set of reversible instructions (the reversible "virtual machine") only needs to be implemented once, and it can then support many different applications. If an application is formally proved down to the fail-safe "virtual machine" level, a strong argument for the complete implementation can be made.

Set against this there are a number of disadvantages. It will be slower, use extra memory storage and may not be appropriate for all computations (especially floating point). With existing floating point hardware it is not possible to define an exact inverse due to round-off problems. For many safety applications however, relatively slow computations using integers are sufficient to implement a safety function, and this is the focus of the current work.

3. A prototype reversible computer

In order to evaluate the approach a simple reversible instruction set, called ARC (A Reversible Computer), has been implemented. A prototype of the reversible instruction set has been implemented in Forth. This language was chosen because it is readily extensible so that reversible operators can be defined in the language and then intermixed with existing Forth instructions (e.g. for declaring and accessing data variables, or running tests). It is also an interactive language so that the new functions can be rapidly tested and debugged. The interactive capability can also be used to inject faults interactively into the "virtual machine" to check its fault detection capability. The language is not optimal in terms of speed, but once the basic concepts have been established it is relatively easy to re-implement the virtual machine using C or even assembler code.

A brief outline of the Forth language will be given, followed by a description of the design of the reversible virtual machine.

3.1. The Forth language

Forth is rather unusual because it is a stack-oriented language which uses Reverse Polish Notation (RPN). This method is used on some calculators; values are put on the stack then evaluated with an operator and the result is put back on the stack. For example an expression such as:

```
        2 + 30 / 5
```

is represented in Forth as:

```
        2 30 5 / +
```

Reference to a variable (e.g. X) places a memory address on the stack, and an explicit operation is needed to extract the value, so to add two variables X and Y the following code is used:

```
        X @ Y @ +
```

where @ is the "load" operator that extracts a value from a memory location. Additional operators can be defined which take their arguments from the stack and return the result(s) on the stack. For example the operator INCREMENT could be defined as follows:

```
: increment
        1 +
;
```

Once defined, this function can be treated like a built-in operator, e.g.: the sequence:

```
        2 increment
```

will leave the value 3 on the stack. The language contains no GOTO instructions. Conditional statements, while loop and fixed loop constructs are provided in the language. Enhanced versions of these control constructs can be defined as new operators.

3.2. Design of the reversible virtual machine

In any reversible computer design, it is essential to store the garbage data. In Bennett's original work on reversible computing [1], a reversible Turing machine was implemented using an additional "tape" to hold the garbage data. The ARC design uses a similar approach (although the "tape" is actually an array implemented in main memory). During forward execution, garbage data is appended to the tape. When the execution is reversed, the tape is "wound back" whenever a reverse instruction consumes a garbage value. The tape should be fully rewound when the program has been fully reversed. The second design issue is how we represent the forward and reverse versions of the program. In some proposed hardware designs [10], only the forward version of the program is required and the direction of execution is reversed when the end is reached. On the reverse path, the instruction is interpreted differently so that it performs the reverse computational function. This approach is space-efficient as the program is no larger than a conventional non-reversible program, but it is difficult to implement in software.

We examined two alternative approaches for representing the program and its inverse. In the first design, each forward instruction places the address of its matching reverse function on an additional "command tape" (e.g. PLUS would store a REVPLUS operator on the command tape). When the forward execution is completed, reversal can be performed by a REVERSE operator which reads and executes the instructions from the command tape. Execution of these commands will also consume data from the data tape, so both tapes should be fully rewound when the reversal is complete. In the second design, the command tape is eliminated and a second "reverse program" has to be written which is a mirror image of the forward program, e.g. a program such as:

```
        Y:= X + 1
```

would be represented by the following sequence of ARC instructions (which are described in more detail later):

```
        X LOAD
        1 REF
          PLUS
        Y STORE
```

The program to reverse this computation can be created by a sequence of matching REV commands written in the reverse order

```
        Y REVSTORE
          REVPLUS
        1 REVREF
        X REVLOAD
```

A STORE instruction destroys the original memory contents so the prior value is saved on the data tape. The matching REVSTORE command restores the original memory contents. The LOAD does not destroy data, but the REVLOAD is an essential command because it checks that the "uncomputed" value on the stack matches the one stored in memory. Thus the REVLOAD command automatically checks that the reverse computation matches the original input value. The REF–REVREF pair performs a similar check on program constants.

Both designs have similar reverse programs, in the first case it is created on the command tape, while in the second case it is an explicit program. The main difference is that the command tape version represents a particular "thread" through the program, so reverse IFTHEN...ELSE...ENDIF structures are not needed. In the "mirror program" design, the IFTHEN...ENDIF structures have to record which path was executed on the data tape. This information is used by a matching REVENDIF...REVELSE...REVIFTHEN structure to determine which portion of the code should be executed in the reverse program. Loop iteration counts are stored in a similar way, so that the correct number of reverse iterations are performed.

The main problem with the command tape concept is the amount of storage required for long computations. Since the "tapes" are implemented as arrays in memory, space is limited. The "mirror program" design requires less dynamic memory as there is no "command tape". In addition, less information needs to be stored on the data tape (such as the memory location for a REVSTORE instruction) as this information is provided by the mirror program.

In practice, either design would be adequate for a simple safety application, and "tape exhaustion" should not occur provided the maximum usage of the tape can be determined in advance. The finite tape length could even be regarded as an advantage in a real-time system because exhaustion of the tape could be designed to enforce a timing constraint and the system could be designed to be fail-safe if tape exhaustion was detected.

3.3. The ARC machine instructions

The ARC machine instructions are relatively limited but, coupled with some "raw" Forth instructions for data declarations, they are sufficient to perform reasonably complex calculations. The ARC instructions are stack-oriented and this makes it very easy to convert high level language expressions into equivalent ARC instructions. Many language translators convert the input into an intermediate Reverse Polish format before generating the machine code, so there is virtually a one-to-one equivalence between the RPN format and equivalent ARC instructions.

All operators use 32 bit integers and include: memory access operators – LOAD, STORE, REF, LOADINDEX, STOREINDEX; arithmetic operators – PLUS, MINUS, INC, DEC, TIMES, DIV, MAXX, MINN; comparison operators – GT, EQ, NE, LT; boolean operators – ANDD, ORR, XORR, NOTT (the rather odd mnemonics in this list were chosen to avoid existing Forth operators). Each command has a "mirror" command with a "REV" prefix (e.g. PLUS and REVPLUS).

Modified versions of the standard Forth operators were used for conditional commands and iteration. The changes were made to ensure the chosen branch is recorded on the tape so that the correct block of code is executed when it is reversed. The flow control commands are:

Conditional execution with an optional ELSE clause:

```
<bool>  IFTHEN
               <true list>
        ELSE
               <false list>
        ENDIF
```

An iterative loop command:

```
<nloops> DOLOOP
               <loopbody>
         ENDLOOP
```

and an infinite loop command:

```
        BEGIN
               <loopbody>
        AGAIN
```

Loop counters are computed explicitly using the reversible instructions (e.g. INC), so that reversing through loops is possible. Standard Forth commands are used to declare the required variables, i.e.:

```
        n CONSTANT x
```

gives a constant n the symbolic name x;

```
        VARIABLE y
```

declares variable y (a 32 bit integer); and

```
        CREATE x n CELLS ALLOT
```

declares an array x with n integer cells (indexed from zero).

To be useful in a real-time context, we also need plant input-output commands such as GETAN, GETDIG, PUTAN and PUTDIG. These have not been implemented in the experimental version.

3.4. Implementation

The commands are relatively easy to implement, as shown in the following implementation of the PLUS operator. The forward command is:

```
: PLUS
        dup pshtape +
;
```

`and the reverse command is:`

```
: REVPLUS
        poptape tuck - swap
;
```

In the PLUS operation, the "dup" command duplicates the top of the stack, the "pshtape" command transfers it to the data tape; and "+" sums the next two values and leaves the result on the stack.

In the REVPLUS operation one of the original operands is retrieved from the tape using "poptape"; the "tuck" command places a copy of it behind the second item on the stack (i.e. z, y becomes: y, z, y). The two top items on the stack are subtracted ($-$), and the "swap" ensures the top two items on the stack (the original operands) are in the right order.

The concept can be re-implemented in any computer language. An implementation in C could for example use in-line procedures or macros, and pass the input and output values through procedure arguments, e.g.:

```
void PLUS( int x,  y, *z )
{       pushtape( y );
        *z = x + y;
};
```

```
void REVPLUS( int z, *x, *y)
{
        *y = poptape( );
        *x = z - *y;
};
```

3.5. Reversing the computation

With the "mirror program" model, an explicit reverse program must be written. This is executed after the forward program. The mirror program can be implemented by hand, but it is relatively simple to generate the mirror program automatically.

With the "command tape" model, the forward commands also store the command "thread" required for reversal. For example, the PLUS command stores the address of REVPLUS on the command tape. An additional command, REVERSE, is required which reads the command tape and executes the stored command thread in the reverse order.

In both designs there is an INIT command which resets any tape to its start position. This should only be invoked at program start-up. Failure to rewind the tape completely on reversal indicates that some error has occurred.

4. Evaluation of the ARC virtual machine

4.1. The trip application

Once developed, the ARC virtual machine was used to implement a relatively simple trip application, where a trip is initiated if any temperature reading exceeds some upper limit. In reality, temperature measurement values would be obtained from an analogue interface, and the trip result would be sent to a digital interface. In the trial application however, the measured temperatures are assumed to be stored in an array (TLIST) and the trip results stored in another array (RESLIST). The ARC trip program is shown below:

```
\ Loop index variable
    VARIABLE X

\ Number of temperature A/D inputs
    500  CONSTANT #TCS
\ A/D offset equivalent to 4 mA
    917  CONSTANT #OFFSET
\ A/D full scale equivalent to 20 mA
    4096 CONSTANT #SCALED
\ Full scale temp (degrees)
    600  CONSTANT #SCALEM
```

```
\ Max Temp in channel ( degrees )
    300  CONSTANT #TLIM

\ Input temperature array (mA)
    CREATE TLIST #TCS cells allot

\ Trip decision array (per input)
    CREATE RESLIST #TCS cells allot

\ ______ TRIPFUNC ______
\
\ TRIP Function pseudo code
\ X:=0
\ DO X=0, #TCS-1
\   IF ( (TLIST[X] - OFFSET)
\        * SCALEM / SCALED ) > TLIM )
\   THEN
\      RESLIST[X]:=1:
\   ELSE
\      RESLIST[X]:=0
\   ENDIF
\   X:= X + 1
\ ENDDO

\ Begin TRIP Function definition
: TRIPFUNC
   0 REF
   X STORE
   #TCS REF DOLOOP
     X LOAD
     TLIST LOADINDEX
     #OFFSET REF
     MINUS
     #SCALEM REF
     TIMES
     #SCALED REF
     DIV
     #TLIM REF
     GT
     IFTHEN
        1 REF
        X LOAD
        RESLIST STOREINDEX
     ELSE
        0 REF
        X LOAD
        RESLIST STOREINDEX
     ENDIF
     X INC
   ENDLOOP
;
\ End of TRIPFUNC definition
```

This is the basic trip function, and it can be tested interactively by changing values in TLIST and then inspecting the results in RESLIST. In a real application the trip function would be executed in an infinite loop, followed by the reverse program, e.g.:

```
INIT            \initialise tape
BEGIN
    TRIPFUNC
    REVTRIPFUNC
AGAIN
```

The BEGIN...AGAIN operators are standard Forth commands that implement an infinite loop.

This is the "mirror program" form where a specific reverse program has to be written. In the "command tape" form, REVTRIPFUNC would be replaced by a generic REVERSE function which reverses the commands stored on the command tape. While the use of REVERSE avoids the need for a specific mirror program, a mirror program does provide an additional on-line program integrity check (since corruptions or systematic errors in either the forward program or the mirror program will be detectable on reversal).

4.2. Integration with other integrity checks

The program can integrated with other integrity checks. Firstly, the program can be tied to a hardware watchdog where alternating signals are sent on the completion of the forward and reverse paths. For example TICK and TOCK functions could be implemented that send alternate signal values to the watchdog hardware. If the reversal fails, the watchdog will trip out.

After each reversal, all variables (including RESLIST) will be restored to their original values. An independent check can be incorporated after reversal to check that the program variables match some specific pattern. For example each variable could have an initial value set by PRESET-WORK-VARIABLES which reflects its location in memory. After reversal the variables can be checked by CHECK-WORK-VARIABLES to check that the initial pattern of values is still present.

We can also exploit our knowledge about the data tape. The tape region might be corrupted by invalid data assignment operations. To check the integrity of the tape operation we can maintain a sumcheck which can be checked by a special CHECK-TAPE-INTEGRITY function prior to reversal. We also know that the tape should be fully rewound, so this can be checked after reversal (e.g. by CHECK-TAPE-REWOUND). So the overall program structure would be:

```
   PRESET-WORK-VARIABLES
   INIT
   BEGIN
        TRIPFUNC
        CHECK-TAPE-INTEGRITY
        TICK
        REVTRIPFUNC
        CHECK-TAPE-REWOUND
        CHECK-WORK-VARIABLES
        TOCK
   AGAIN
```

Note that further run-time checks are incorporated into the basic ARC instructions to detect tape exhaustion during the forward and reverse executions of the program.

4.3. Storage requirements and timing

To evaluate the overall performance and overheads of the different methods, some comparative tests were performed using the two different reversible methods, and a "direct" implementation which is non-reversible. For a 500 channel trip function, run on a 90 MHz Intel Pentium, the storage requirements and execution times shown in Table 1 were obtained.

Version	Tape Used (kilobytes)	Time (millisecs)
Command tape	120	504
Mirror program	20	102
Non-reversible	n/a	10

Table 1. Reversible computing performance

The storage requirements are quite high for the "command tape" version, mainly because the reverse instructions are stored on an additional command tape and this increases for every DOLOOP iteration. Major storage savings could be achieved by reversing each channel computation individually.

All the execution times appear to be adequate for simple safety applications where input/output execution times could well be the dominant factor (e.g. up to 20 milliseconds for an analogue measurement). However, it should be noted that the execution times could increase by a factor of 10 for a slower processor (such as an Intel 80386).

It can also be seen that the "command tape" version is about five times slower than the "mirror program" version, which in turn is about ten times slower than a direct implementation. Similar ratios might be expected if the ARC virtual machine was implemented in other languages and processors. With an optimised ARC virtual machine (e.g. implemented in C or assembler) an Intel 80386 processor might be able to execute the TRIPFUNC program and its reverse in about the same time as the Forth version on a Pentium (i.e. 0.1 seconds). This would leave plenty of spare capacity for more complex real-time applications.

4.4. Response to data corruption

Random data corruption can be catered for by specific hardware checks (such as memory parity checks) and by using redundancy. However data can also be corrupted by flaws in program data flows (e.g. due to errors in the compiler, linker or the processor), and these are potentially more dangerous as they can cause failures in redundant channels. In order to test the response to data corruption, some interactive functions were implemented which could corrupt the values on the tape, and the main program variables (the RESLIST and TLIST arrays). After executing TRIPFUNC, a data location was corrupted and then the execution was reversed. A simple test environment was constructed to automate this process, and 1000 random corruption tests were applied to:

- the input values
- the result values
- the data "tape"

For all 3000 tests it was found that all the inserted corruptions were detected by the checks. This is hardly surprising, as the REVLOAD, REVREF and tape integrity checks should find any single instance of corruption. Compensating multiple corruptions are a possibility, but these are likely to be low probability events.

4.5. Response to computation flaws

It is difficult to estimate the detection probability of computational flaws in operators like PLUS or TIMES because it is hard to establish the behaviour of an unknown fault. To get a rough estimate of the likely detection probability, the detection performance was measured using a corrupted data tape but removing the tape integrity check. The rationale behind this assessment is that a flaw in a computation function will result in some discrepancy between the main computed values and the garbage values on the data tape. The reverse computing functions should detect these if they occur. The responses of the reverse computing operation to 1000 random corruptions of the data tape are shown in Table 2, together with the integrity check that identified the fault.

Detection method	Number detected	Percent detected
REVREF check	518	51.8
REVLOAD check	204	20.4
Boolean value check	139	13.9
CHECK-WORK-VARS	76	7.6
Comparison (REVGT, etc.)	33	3.3
Total	970	97.0

Table 2. Initial fault detection performance

Around 97% of the corruptions were detected. The REVLOAD, REVREF and preset variable checks found around 80% of the injected errors. These are the direct result of uncomputing from the output values to the input values. The boolean and comparison checks are intermediate checks (e.g. for legal boolean values, correct operation of AND, OR, etc.). A closer inspection was made of the remaining 3% of undetected errors, and it was found that the TIMES / REVTIMES pair was the major culprit. The lack of detection can be explained as follows. On the forward path the result is computed and one of the operands is put on the tape, e.g. 5×1001 produces the result 5005 and one of the operands, e.g. 1001, is stored on the tape. On the reverse path one of these values is corrupt, for example 1001 becomes 1000. The reverse calculation divides 5005 by 1000 which yields the correct result of 5 for the other operand (assuming integer division). It is clear that the reverse operation is incomplete as it discards data; we know that a divide operation has a remainder and that this remainder should always be zero. An exact divisor check was added to the REVTIMES operation.

The remaining five detection failures were associated with the STOREINDEX operation which computes the wrong address. On reversal, REVSTOREINDEX uncomputes the original array index—this involves a divide by four as each integer occupies four bytes. Like the REVTIMES operation, this divide operation failed to check that the remainder was zero, so small changes to the address would not alter the uncomputed index. This explains the 5 cases where reversal failed to detect the corruption. A check for exact division was added to all indexing operations.

The tests were repeated with the exact divisor checks incorporated in REVTIMES and the indexing instructions and an extended test was performed using 58 007 random corruptions. All cases of corruption were detected. The results are summarised in Table 3.

Detection method	Number detected	Percent detected
REVREF check	20185	34.8
REVLOAD check	11927	20.6
REVTIMES check	12229	21.1
Boolean value check	8251	14.2
CHECK-WORK-VARS	3960	6.8
Comparison (REVGT, etc.)	936	1.6
Exact index check	519	0.9
Total	58007	100.0

Table 3. Fault detection (exact divisor checks)

The exact divisor checks occur at intermediate points in the calculation and will therefore detect errors that would otherwise be trapped at the input values, i.e. by REVREF and REVLOAD. It can be seen that REVTIMES detects far more than the missing 3% and there is a matching fall in the number detected by REVREF.

5. Discussion

5.1. Fail-safe bias

The main intent of this design approach is to detect failures due to systematic design faults rather than random hardware faults. Normally, random hardware faults would be revealed by specific hardware checks (such as memory parity and memory bound limits) and channel redundancy. Nevertheless the test procedure makes use of random corruptions and it is encouraging to note that all 58 007 corruptions were detected by the method. This was observed even when the tape integrity check on the "garbage" data was omitted. This suggests that individual channels will exhibit a high fail-safe bias of perhaps 99.998% for random internal failures.

The realism of the fail-safe performance in response to systematic faults is more debatable, as it depends on the nature of the postulated fault. In practice the failures would have to be infrequent otherwise they would be detected during the normal verification and validation tests, so the transient corruptions used in the tests could well be a reasonable representation of the behaviour of such faults, but it would be desirable to assess the response to a range of postulated hardware and compiler faults. Nevertheless the current test results indicate that the fail-safe bias could be 99.998% or better.

It could be argued from a theoretical point of view that any failures due to a single flaw in an N-to-N mapping should be 100% detectable as the converse mapping will still be N-to-N and should expose any failure. Set against this, there could be faults in the implementation (such as the omission of the exact divisor check) which could reduce the achieved level of fail-safe bias. In practice therefore, a more comprehensive evaluation is required to support a claim for a high fail-safe bias.

5.2. Run-time overheads

Although the storage and timing overheads are significant, reversible computing appears to be a practical proposition for the straightforward computations found in safety applications. Other software fail-safety techniques (such as vital coded processing [4]) can involve a hundredfold increase in execution time, so the tenfold increase of our technique should not be a major limitation. In addition, the speed could be further improved by implementing the ARC instruction set in assembler code or C, and the storage requirements can be further reduced by reversing specific sub-computations.

5.3. Alternative implementation options

There are many alternative strategies for implementing reversible computing which could be further explored. For example, the existing implementation relies on a software comparison of the uncomputed results. This could be replaced with a fail-safe hardware checker that maintained a stack of input values and then compared these with uncomputed values.

It would also be feasible to remove the mirror program completely. The program could be implemented as a sequence of byte codes (similar to the Java virtual machine), and this could be interpreted by a forward interpreter and a reverse interpreter which works back from the end. This is likely to be less efficient than an explicit mirror program but it would be relatively easy to implement directly in hardware where the overheads would be much lower.

It would also be possible to use separate processors to hold the forward and reverse programs, with the "garbage values" and final output values being passed to the second processor for reversal. The uncomputed values derived by the reverse processor could then be compared with the original input values using a hardware comparator. For extra assurance, diverse virtual machines and computer hardware could be used for the two processors.

5.4. Functional limitations

It is important to realise that reversible computing is not a panacea for ensuring fail-safety; it has limitations and the technique should be used in conjunction with other methods to ensure the safety of the overall system:

- It only checks the integrity of the low-level instructions. Separate methods are needed to ensure that the application software performs the intended function (e.g. through validation, formal methods, etc.).

- There is no exact reversal for floating point. Any floating point reversible machine would have to test if the result was "close" to the original value. This might be turned to advantage if we are concerned about algorithmic stability; a stable algorithm would have reverse results close to the original inputs, while an unstable algorithm would not.

- There is no protection against faults in the input-output hardware, so additional input-output integrity checks are required. For example, diverse measurements and application-level credibility checks could be used.

6. Summary of results

The reversible computing concept looks very promising. The main technical results of our research study are that:

1. It is strongly fail-safe and can protect against both random and systematic faults in the underlying compiler and processor hardware.

2. The tests performed indicate a fail-safe bias of better than 99.998%; however this may be over-optimistic as the tests did not simulate a complete range of credible systematic faults in the hardware and compiler.

3. The instruction set is capable of handling quite complex applications and there is scope for extension to floating point operations.

4. The approach is generic. The same reversible instruction set can be used on many different applications, and the instruction set can be implemented on any computing hardware and compiler technology.

There are some limitations to the method:

1. A reversible program runs about ten times slower than a conventional one, but it can be optimised for higher performance. The same virtual machine concepts can be readily implemented in other languages such as C or assembler code, or directly in hardware.

2. Significant storage is needed for the "garbage data" needed for reversal, but exhaustion of the storage space can be a useful mechanism for detecting runaway programs and timing overruns.

There is considerable scope for further development of the concept, e.g. hardware support for reversible computing, extension to floating point, and code generators to automate the construction of reversible programs.

7. Acknowledgements

This work was funded by the UK (Nuclear) Industrial Management Committee (IMC) Nuclear Safety Research Programme under Scottish Nuclear contract PP/74851/HN/MB with contributions from British Nuclear Fuels plc, Nuclear Electric Ltd, Scottish Nuclear Ltd and Magnox Electric plc.

References

[1] C.H. Bennett, "Logical Reversibility of Computation", IBM Journal of Research and Development, vol. 6, pp. 525–532, 1973

[2] C.H. Bennett, "The Thermodynamics of Computation, a Review", International Journal of Theoretical Physics, vol. 21, pp. 905–940, 1982

[3] C.H. Bennett, "Notes on the history of reversible computation", IBM Journal of Research and Development, vol. 32, pp. 281–288, 1988

[4] P. Chapront, "Vital Coded Processor and Safety Related Software Design", Proceedings SAFECOMP 92, Pergamon Press (Zurich), 1992

[5] J.S. Denker, S.C. Avery, A. G. Dickinson, A. Kramer and T.R. Wik, "Adiabatic Computing with the 2N-2N2D Logic Family", International Workshop on Low Power Design, pp. 183–187, 1994

[6] E.F. Fredkin and T. Toffoli, "Conservative Logic", International Journal of Theoretical Physics, vol. 21, no. 3/4, pp. 219–253, 1982

[7] R. Landauer, "Irreversibility and Heat Generation in the Computing Process", IBM Journal of Research and Development, vol. 5, pp. 183–191, 1961

[8] R. Landauer, "Uncertainty Principle and Minimal Energy Dissipation in the Computer", International Journal of Theoretical Physics, vol. 21, no. 3/4, pp. 283–297, 1982

[9] R.C. Merkle, "Towards Practical Reversible Logic", Physics and Computation, pp. 227–228, October 1992

[10] J. Storrs Hall, "A Reversible Instruction Set Architecture and Algorithms", Physics and Computation, pp. 128–134, November 1994