Microelectronics Reliability 54 (2014) 1050–1055

Contents lists available at ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel

Algorithm transformation methods to reduce the overhead of software-based fault tolerance techniques

José Rodrigo Azambuja [a,*], Gustavo Brown [b], Fernanda Lima Kastensmidt [a], Luigi Carro [a]

[a] Instituto de Informática, PPGC and PGMICRO at Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
[b] Facultad de Ingeniería at Universidad de la República, Montevideo, Uruguay

ARTICLE INFO

Article history:
Received 29 July 2011
Accepted 22 November 2013
Available online 22 December 2013

ABSTRACT

This paper introduces a framework that tackles the costs in area and energy consumed by methodologies like spatial or temporal redundancy with a different approach: given an algorithm, we find a transformation in which part of the computation involved is transformed into memory accesses. The precomputed data stored in memory can be protected then by applying traditional and well established ECC algorithms to provide fault tolerant hardware designs. At the same time, the transformation increases the performance of the system by reducing its execution time, which is then used by customized software-based fault tolerant techniques to protect the system without any degradation when compared to its original form. Application of this technique to key algorithms in a MP3 player, combined with a fault injection campaign, show that this approach increases fault tolerance up to 92%, without any performance degradation.

1. Introduction

Future technologies will be much more unreliable [1] and, at the same time, the performance gap between memory and processors will not get any smaller [2]. Memories have long been protected against multiple fabrication defects [3,4]. Hence, thanks to their regularity, memories would be a natural fabric to help one cope with unreliable technologies. Although the idea of bringing computation to memory is old [5,6], it never quite succeeded. However, as we move from an era where single defects, high reliability and high yield were present, to a situation with multiple defects and low yield in the logic, the idea of using high reliable memories as a substitute to traditional computation gets more appealing.

In this paper we present a framework for algorithm transformation with the purpose of achieving reliable fault tolerant designs and, at the same time, improve performance. We show that memory can be used as a direct replacement of computations, thus decreasing the area of unreliable hardware that cannot be easily corrected or protected [7]. Furthermore, the same strategy that favors reliability also favors parallelism. The main idea is to analyze a given algorithm and, using induction variables analysis [8,9] and other related tools like memorization [10], replace most of the computations a processor performs by accesses to some tables of precomputed values stored in memory. Our aim is to transform the algorithm in such a way that the computations left are just applications of simple functions over the input data and the precomputed data. By simplifying the amount of computations that must still be done by the processor, software-based fault tolerance can be better applied, and hence no performance penalties are incurred, but fault tolerance improves by 92%.

Many of the algorithms for data processing used nowadays allow for the transformations here proposed. We will focus on two of the key algorithms which are part of the MP3 [11] coding scheme, namely the modified cosine discrete transformation (MDCT) [12] and the Huffman coding algorithm [13], along with the discrete Fourier transformation (DFT) [14] widely used in signal processing. Finally, we will discuss how the proposed fault tolerant strategy can be deployed in this real life application.

2. Related work

Enhancing reliability has become one of the key issues for current and future hardware designs. Several research trends on this subject are described in [1,15]. Aside from the ongoing efforts on fault avoidance [16,17], current fault tolerance techniques rely on space or time redundancy to provide fault tolerance [18,19] which, for TMR, triplicates the amount of space/time required.

In the 70s Stone [5] described a technique to use memory as an alternative to classical computation. However, it has never gained substantial attraction, as it posed restrictions on the way the program running on such devices should be written. Later, with the introduction of Computational RAM, a new architecture to bring

* Corresponding author. Tel.: +55 51 3308 7036.
E-mail addresses: jrfazambuja@gmail.com, jrazambuja@inf.ufrgs.br (J.R. Azambuja), gbrown@fing.edu.uy (G. Brown), fglima@inf.ufrgs.br (F.L. Kastensmidt), carrofglima@inf.ufrgs.br (L. Carro).

0026-2714/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.microrel.2013.11.011

computation to memory was proposed [6]. It allows a dual use of memories; memory modules can be seen either as traditional DRAMs, or as independent SIMD systems which are amenable for parallel applications. Even though the performance improvement with this technique looks promising, it appears that more research is needed to develop applications that take advantage of it.

The use of static analysis tools like induction variables analysis is very common in the compiler construction area [9,20] as it allows one to improve the performance of the compiled code. It has also been used as a tool for code optimization targeted to VLSI designs [21,22]. Memorization, on the other hand, relates to a dynamic optimization technique used primarily to compute any given function only once, and return a cached value any time it is required again. Although usually a software based technique, it has also been incorporated in hardware based solutions [23,24]. Our work relies on these tools to analyze and transform a given algorithm, but now focusing on reliability enhancement and fault tolerance as a major goal. Nonetheless, as we later show, the same tools that help one improve reliability also favor performance.

Using memories to help one to achieve high reliability designs is a common task nowadays [4,25]. The regularity found on memories, the use of error correcting codes and small extra logic added to cope with spare memory rows and columns allow one to efficiently protect them against multiple faults [3]. Furthermore, with the introduction of magnetic and ferroelectric RAMs, the soft error rate of such devices dumped near zero [7]. This is why we believe one should take advantage of the regular structure of memories (that ease low cost ECC introduction) to better use them at the software level, increasing global reliability, without compromising performance.

3. Proposed algorithm transformation method

The framework for algorithm transformation proposed in this paper consists of rounds of analysis/transformation steps which are repeated until no more transformations are feasible. In every round, parts of the algorithm which apply some computations are extracted to memory in the form of precomputed indexed data structures. Care must be taken to assure that the optimized algorithm yields the same results as the original algorithm, and at the same time the time/space required by the optimized algorithm does not exceed that of other fault-tolerant approaches like triple modular redundancy.

The extraction of computation to memory is basically addressed by the use of precomputed tables, which in turn are indexed by variables related to the algorithm itself (typically updated by the algorithm's inner loops). Thus, to keep the time required to compute the algorithm constrained, care has to be taken just for the precomputation of the data structures which will be put in memory. On the other hand, space requirements will need much more attention as the size needed to hold the precomputed data might grow too large, whenever the number of indices or their range become too large.

The first step is to analyze the algorithm to find some portions that fit in the description above. Those portions typically compute some values based on input values derived from the algorithm itself (indices of inner loops and other variables), or the algorithm's input data, provided that the range of these inputs are not too large. This step might be performed manually or automatically with the aid of static analysis tools (induction variables analysis) usually found on modern compilers. The next step involves the rewriting of the identified portions as accesses to data structures stored on memory. Finally the data structure must be populated with the precomputed values prior to the execution of the algorithm. These steps might be performed repeatedly until no more portions of the algorithm are susceptible of such transformations.

After all the transformations are applied, two things must be taken care of to improve the reliability of the hardware it will operate on. On one hand, the data structures which hold the precomputed values have to be protected. This can be done using any of the existing memory protection techniques usually found on literature. Note that at this point the hardware designer has quite some flexibility regarding the level of protection (i.e. guard against multiple defects, etc.) depending on the chosen the protection scheme and, contrary to traditional TMR implementations, space/time requirements grow logarithmically with the fault tolerance protection.

On the other hand, even as we move computations to memory, there is still the need to compute something (the memory address for one thing), and the hardware involved in these computations must also be protected. This can be accomplished by applying traditional fault tolerance techniques and by taking into account that, as the computation needed to execute the algorithm gets smaller, the hardware involved could be simpler and well protected, without loss of performance compared to the original algorithm.

4. Applying algorithm transformation methods to case study algorithms

Not every problem is amenable to the transformations we propose to apply. For example, the simple scalar code $A = x.y$ where x and y are 16 bits variables would require a giant and slow memory, and hence the granularity of the proposed approach is important.

Furthermore, once one chooses a problem to optimize, one has to select a proper algorithm that solves it. For now, we will focus on algorithms which are heavily based on matrix operations. These algorithms usually are built over the application of some functions over the internal loop indices, and the actual input data generally fulfills our requirements of memory space constraints. For the experiments reported in this paper, we manually transformed the algorithm using the approach described in previous sections, so that most of the complex operations are already precomputed in memory, and we leave simple operations that handle large range dynamic data (input or temporary) to be computed online using the precomputed data. The resulting algorithms allow a fault-tolerant hardware implementation via the protection of the precomputed data structures held in memory. For our case study, we focused on modules which are responsible for more than 70% of the execution time of an MP3 player and other signal processing problems.

4.1. Case study 1: MDCT algorithm optimization

We first work on the MDCT problem, which is defined as [12]:

$$X_k = \sum_{n=0}^{2N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}+\frac{N}{2}\right)\left(k+\frac{1}{2}\right)\right],\ 0 \le k < N$$

where for MP3 coding, N is either 12 or 18. The naïve implementation is described in Fig. 1. Note that in the second term the cosine

```
input: vector x (size 2N), matrix M
output: vector X (size N)
for k = 0 to N-1
  for n = 0 to 2N-1
    X(k) = X(k) +
    + x(n)*cos(pi/N*(n+1/2+N/2)*(k+1/2))
  next n
next k
```

Fig. 1. Naïve MDCT implementation.

```
input: vector x (size 2N), matrix M
output: vector X (size N)
for k = 0 to N-1
  for n = 0 to 2N-1
    X(k) = X(k) + x(n)*M(k, n)
  next n
next k
```

Fig. 2. Optimized MDCT implementation.

calculation performed does not depend on the actual input data, but rather solely on the indices of the two loops.

Thus, we may precompute the result of all those operations and gather them in a matrix $M_{N\times 2N}$, where $M(i,j) = \cos\left[\frac{\pi}{N}\left(i+\frac{1}{2}+\frac{N}{2}\right)\left(j+\frac{1}{2}\right)\right]$. The optimized code shown in Fig. 2 performs just a series of accumulate-multiply operations.

4.2. Case study 2: Huffman algorithm optimization

We will focus on another of the algorithms used by an MP3 coder, the Huffman code [13]. The Huffman coding problem is a variable length entropy encoder which derives an optimal weighed path length of the code given an input alphabet and a set of frequencies for each input symbol. Formally, let $A = \{a_1, a_2, \ldots, a_n\}$ be the input symbol alphabet of size n, and $W = \{w_1, w_2, \ldots, w_n\}$ be the set of symbol weights of the input source. Then, let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of codewords derived for that input by the algorithm. The goal of the algorithm is to find such a set C so that the weighed path length of C is optimal compared to any other code constructed based on the sets A and W. The algorithm so described can be generated in $O(nlog(n))$ sequential time. However, in this format the algorithm is not amenable neither to parallelization nor to have its behavior moved to memory with precomputed tables. Thus, we will work over another solution described in [26]. Moreover, instead of constructing the Huffman code we optimize the construction of the *height bounded subtree* [26] from which the actual Huffman code can be derived.

Our starting point is the algorithm described in Fig. 3. All the arithmetic is done using the tropical semiring [27]. Initially two matrices S and A are initialized with appropriate values using the non-decreasing vector of input probabilities. After that, three operations are applied sequentially $log_2(N)$ times. The resultant matrix A contains the set of average codeword length, from which the actual Huffman code can be derived.

We begin our optimization by applying two simple optimizations. First we use a table of precomputed values for the $log_2(N)$ function, as it depends only on the upper bounded input value N. Then we collapse all three operations (tropical_product, matrix_sum and matrix_min) onto a unique operation, tropical_square_minsum depicted in Fig. 4. This way, all the overhead of performing the loops is done only once.

Manually applying an induction variable analysis we note that the computation of the resulting value in each location depends on whether $min + S(j,i) < A(j,i)$, if it is not the case then the value for TSM(j,i) will be $A(j,i)$. Thus, we may apply a further transformation to S so as to avoid the if-then-else. In the final implementation, shown in Fig. 5, the initialization of the S matrix and the initial value for min are slightly modified to accomplish this.

However, there is still a comparison and an addition over the tropical semiring that, at this point, cannot be further optimized without the use of a huge memory, because its input indices over a precomputed data structure have a large range. Therefore we conclude our transformations and stop at this point. The resulting algorithm is again suitable for a fault tolerant hardware implementation. Aside from moving part of the computations to memory, we also removed some of the inner loops contained in the initial algorithm which leads to a faster implementation.

```
input: vector p probabilities (size N)
output: matrix A with height bounded
subtree

S,A = initialize
h = ceil(log2(N));
for i = 0 to h
   A_prod = tropical_product(A, A)
   A_h_right = matrix_sum(A_prod, S)
   A = matrix_min(A, A_h_right)
next i

procedure initialize: S, A
for j = 0 to N-1
  for i = 0 to N-1
    if (j <= i)
      S(j,i) = sum(p(k), j<=k<=i)
    else
      S(j,i) = INFINITE
      A(j,i) = INFINITE
  next i
  A(j,j) = 0
next j

procedure tropical_product(A,B):Product
for j = 0 to N-1
  for i = 0 to N-1
    min = INFINITE
    for k = 0 to N-1
      if A(j,k-1) + B(k,i) < min
          min = A(j,k-1) + B(k,i)
    Product(j,i) = min;
  next i
next j

procedure matrix_sum(A,B):Sum
for j = 0 to N-1
  for i = 0 to N-1
    Sum(j,i) = A(j,i) + B(j,i)

procedure matrix_min(A,B):Min
for j = 0 to N-1
  for i = 0 to N-1
    if A(j,i) > B(j,i)
      Min(j,i) = B(j,i)
    else Min(j,i) = A(j,i)
```

Fig. 3. Naïve Huffman implementation.

```
input: vector p probabilities (size N)
output: matrix A with height bounded
subtree

S,A = initialize
h = Log2Table(N);
for i = 0 to h
   A = tropical_square_minsum(A, S)
next i

procedure tropical_square_minsum(A,
S):TSM
for j = 0 to N-1
  for i = 0 to N-1
    min = INFINITE
    for k = 0 to N
      if A(j,k-1) + B(k,i) < min
          min = A(j,k-1) + B(k,i)
    data = min + S(j,i)
    if dato < A(j,i)
      TSM(j,i) = data
    else TSM(j,i) = A(j,i)
   next i
next j
```

Fig. 4. Initial optimized Huffman implementation.

```
procedure initialize: S, A
for j = 0 to N-1
  for i = 0 to N-1
    if (j < i)
      S(j,i) = sum(p(k), j<=k<=i)
    else
      S(j,i) = 0
      A(j,i) = INFINITE
  next i
  A(j,j) = 0
next j

procedure tropical_square_minsum(A,
S):TSM
for j = 0 to N-1
  for i = 0 to N-1
    min = A(j,i)
    for k = 0 to N
      if A(j,k-1) + B(k,i) < min
          min = A(j,k-1) + B(k,i)
      TSM(j,i) = min + S(j,i)
   next i
next j
```

Fig. 5. Final optimized Huffman implementation.

4.3. Case study 3: DFT algorithm optimization

We will now shift our attention to another signal processing tool, the *Discrete Fourier Transform* (*DFT*) [14], which is broadly used in digital signal processing applications. Although there exists fast algorithms to compute the transform (Fast Fourier Transform) we will focus on the DFT as it can scale well with parallel processing, which the FFT cannot.

The DFT is defined as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}kn}, \quad 0 \le k < N$$

where in our case we use $N = 16$, thus computing the 16-point complex DFT. Fig. 6 shows de initial naïve implementation, where instead of using complex exponentiation we use sines and cosines.

As with the case of the MDCT, we can precompute the cosines (and sines) used with respect to the indices of the for loops involved in the computation. Also, the computation of the DFT requires a division of every term by N. We can also move that division to the precomputed tables, according to the following matrices $M\ \cos_{N\times N}$, and $M\ \sin_{N\times N}$, where $M\ \cos(j,i) = \frac{\cos[(-ij2\pi)]}{N}$ and $M\ \sin(j,i) = \frac{\sin[(-ij2\pi)]}{N}$.

The transformed algorithm, shown in Fig. 7, uses this precomputed matrices to compute the DFT with a series of additions and subtractions.

4.4. Implementation notes

We developed implementations of the three algorithms discussed for both versions (naïve and optimized algorithms). Table 1 shows a comparison of the results obtained for each implementation. For the MDCT algorithm, the naïve version relies on a table of

```
input: vector x, y (size N)
output: vector x2, y2 (size N)
for j = 0 to N-1
  for i = 0 to N-1
    cos = cos(-i * (pi * 2* j / N));
    sin = sin(-i * (pi * 2* j / N));
    x2[j] += (x1[i]*cos - y1[i]*sin)/N;
    y2[j] += (x1[i]*sin + y1[i]*cos)/N;
  next i
next j
```

Fig. 6. Naïve DFT implementation.

```
input: vector x, y (size N),
       matrix Mcos, Msin
output: vector x2, y2 (size N)
for j = 0 to N-1
  for i = 0 to N-1
    x2[j] += x1[i]*Mcos[j][i]-
             y1[i]*Msin[j][i];
    y2[j] += x1[i]*Msin[j][i] +
             y1[i]*Mcos[j][i];
  next i
next j
```

Fig. 7. Optimized DFT implementation.

Table 1
Comparison of naïve and optimized algorithm implementations.

	Naïve	Optimized
MDCT		
Program size (number of instructions)	60	19
Memory size (bytes)	1600	2592
Execution time (ms)	2.05	0.64
Huffman		
Program size (number of instructions)	104	76
Memory size (bytes)	3072	3136
Execution time (ms)	1.59	1.49
DFT		
Program size (number of instructions)	81	56
Memory size (bytes)	3456	2304
Execution time (ms)	1.20	0.76

400 cosine values evenly spaced, while the optimized algorithm relies on a table of precomputed cosine values which are the exact values that the algorithm uses. The optimized algorithm uses 60% more memory to hold the precomputed cosines table, but the program itself can be implemented with 66% less code and the execution time is reduced by 68%. In the case of the Huffman algorithm, aside for a small lookup table used in the optimized algorithm to find a logarithm, both versions utilize the same amount of memory to hold the auxiliary matrices, although the data stored in memory is slightly different. The program size of the optimized algorithm shows a 25% reduction and the execution time shows a 6% improvement. For the DFT algorithm, the naïve versión relies on two tables, one holding 400 cosine values evenly spaced and the other for a 400 sine values. The optimized algorithm does not use these tables. Instead it uses two precomputed matrices that already hold the information needed to compute the DFT. Therefore the memory requirements decrease in the optimized version as the memory required to hold the precomputed matrices is inferior than the size of the cosine and sine tables used in the naïve version. Choosing a careful layout for the memory structures (vectors and matrices of precomputed values) results in simple yet efficient programs. The final versions of both programs resemble the application of a simple operation over a matrix. One level of fault tolerance is achieved by protecting the data structures held in memory. The actual protection scheme used might be selected according to the level of reliability the application needs to meet.

Those parts of the algorithm not suitable for transformation (the for-loops, multiply-accumulate and minima's search) which are left will be protected using software-based fault tolerant techniques. Table 1 shows a comparison between the naïve and optimized implementations, while Table 2 presents the instruction count for each implemented version.

5. Applying software-based techniques to case study algorithms

Fault tolerance techniques based on software can provide high flexibility, low development time and low cost for computer-based

dependable systems [28–30]. Such techniques offer fault tolerance by exploiting information redundancy, control flow analysis and comparisons to detect errors during the program execution. For that, software-based techniques use additional instructions in the program code to either recompute instructions or to store and to check suitable information in hardware structures.

Software-based techniques increase the execution time and memory occupation, since instruction replication is inserted in the program code, comparing the replicated data stored in the data memory and executing code interpolated with the original program. Considering software-based fault tolerance, the memory overhead is not an issue, since the memory can be protected with ECC techniques. On the other hand, the execution time can be an issue, depending on the application. In this section, software-based techniques will be adapted to the optimized algorithms in order to increase their execution time up to the naïve version execution time, which can be seen on Table 1. The hardening transformation will be performed using a tool called HPCT [31].

5.1. Case study 1: Optimized MDCT algorithm hardening

According to Table 1, the optimized MDCT algorithm executes in 0.64 ms. Therefore, its design space allows us to increase its runtime up to 2.2 times, while maintaining the original execution time. The program code uses 7 registers and executes a program code with a total of 19 instructions.

In order to harden the optimized MDCT, we transformed the code using three different software-based techniques, called signatures, variables and inverted branches, described in [32]. The first technique divides the program code into basic blocks and associates a unique identifier to each one. The unique identifier is then set at the beginning of each basic block, and verified upon its exit by instructions inserted on the original code. The second technique duplicates all instructions (except branch instructions) over replicated registers. It then verifies register values with their replicas whenever a register is read by any instruction. By duplicating the instructions and registers, the variables technique replicates the whole data-flow and is then capable of detecting all faults upsetting the microprocessor's data path. The last technique replicates the branch instructions, by inserting a replicated branch after the original instruction and an inverted branch in the target address of the original instruction. Both replicated branch instructions have an error subroutine as target address.

Table 3 shows that even applying all three techniques, the execution time remained lower than the original, with a 25% reduction, from 2.05 ms to 1.53 ms.

5.2. Case study 2: Optimized Huffman algorithm hardening

The optimized Huffman algorithm offers a smaller design space than the MDCT, since the reduction in execution time from the original was 6.7%, and therefore not all techniques could be applied. The same three techniques used in case study 1 were applied separately to the optimized Huffman algorithm, resulting in a higher execution time overhead than the allowed 6.7% for all techniques. This means that one technique should be chosen and customized in order to fit the allowed overhead in execution time.

In this case, we chose as a starting point the variables technique, due to its better fault tolerance rates [31] and due to the fact that it can be easily customizable to protect a selected group of registers, decreasing the overhead in execution time proportionally to its fault tolerance rate. We started protecting a group of one register and added new registers until the execution time reached the allowed 1.59 ms.

The execution time of the optimized and protected Huffman algorithm reached 1.59 ms when the group of protected registers had 5 registers, from a total of 19 used in the program code. Table 3 shows the results for the optimized and protected Huffman algorithm compared to the naïve and optimized versions.

5.3. Case study 3: Optimized DFT algorithm hardening

The optimized DFT algorithm presented a reduction of 36% in execution time, restricting the application hardening through software-based techniques. The same three techniques applied in the other case studies were applied in the DFT algorithm, resulting in higher execution times than the naïve version. Therefore, one technique should be chosen and customized to fit the allowed overhead, as we performed on case study 2.

As in case study 2, we started the hardening by applying the variables technique [31], because of their easy customization and high detection rates. We started by hardening a group of registers and added new registers until the execution time reached the higher value lesser than the naïve version, which is 1.20 ms.

The execution time of the optimized and protected DFT algorithm reached 1.18 ms when the group of protected registers had 6 registers, from a total of 10 used in the program code. Table 3 shows the results for the optimized and protected DFT algorithm compared to the naïve and optimized versions.

6. Fault injection experimental results

The chosen case-study microprocessor is a five-stage pipeline microprocessor based on the MIPS architecture, but with a reduced instruction set. The miniMIPS microprocessor is described in [33].

In order to perform the fault injection campaign, 10 thousand faults were randomly generated for each program, considering the execution time and a list of every signal of the microprocessor description (including registered signals). SEU and SET types of faults were injected directly in the microprocessor VHDL code by using ModelSim XE/III 6.3c, one fault per program execution. SEUs

Table 3
Comparison of naïve, optimized and protected algorithm implementations.

	Naïve	Optimized	Optimized and protected
MDCT			
Program size (number of instructions)	60	19	109
Memory size (bytes)	1600	2592	5184
Execution time (ms)	2.05	0.64	1.53
Huffman			
Program size (number of instructions)	104	76	128
Memory size (bytes)	3072	3136	6272
Execution time (ms)	1.59	1.49	1.59
DFT			
Program size (number of instructions)	83	58	102
Memory size (bytes)	3456	2304	4608
Execution time (ms)	1.20	0.76	1.18

Table 4
Results for set and seu fault injection campaign in the MDCT, Huffman and NEWAPP algorithms for 10 thousand faults.

	(I) Naïve	(II) Optimized	(III) Optimized and protected
MDCT			
Incorrect results	1913	1412	149
Huffman			
Incorrect results	1056	1291	840
DFT			
Incorrect results	2655	2931	1826

were injected in registered signals, while SETs were injected in combinational signals, both during one and a half clock cycle. The fault injection campaign is performed automatically. At the end of each execution, the results stored in memory were compared with the expected correct values. If the result did not match, the fault was classified as a wrong result.

For each of the three case study algorithms, three software implementations were tested: (I) Naïve version, (II) Optimized version and (III) Optimized and Protected version. Each of them was upset with 15 thousand faults. Results of the fault injection campaign are presented in Table 4.

The quantity of faults that caused an error in the system for the MDCT algorithm (I) was reduced in 26% by optimizing it (II), and in 92% by optimizing and protecting it with software-based techniques (III), aside from the 25% reduction in execution time (Table 4). The Huffman algorithm presented an increase of wrong answers when optimized (II) and compared to the original (I), due to the use of a bigger number of registers. On the other hand, a reduction of 20% on the number of wrong answers was achieved when combining the optimization and the protection through software-based techniques (III), while maintaining the original execution time of 1.59 ms (Table 4). For the DFT algorithm the number of errors of the optimized version (II) compared to the original (I) increased by 22% and the execution time dropped almost 40%. However, when comparing the original (I) with the optimized and protected (III) the number of incorrect results decreased over 30% with the same execution time.

7. Conclusion and future work

This work presents a framework for algorithm transformation which leverages on static analysis tools like induced variables analysis and memorization usually found on modern compilers to derive a modified algorithm which makes use of precomputed data structures stored in memory to be used instead of traditional computation. These techniques let us create fault-tolerant hardware designs by protecting the precomputed memory segments with any of the available and well established memory protection schemes used nowadays. The ratio of reliability-overhead sought may be tuned by adjusting the parameters of the ECC used to protect the memories.

As an effect of the transformations derived from our approach the final algorithms show performance improvements, which are then used to protect the program code with software-based techniques. The application of customized software-based techniques is able to increase the fault tolerance from 20% to 92%, varying according to the performance gain due to the optimization methods.

We showed the application of this framework with two simple algorithms used on the MP3 coding scheme and also for a algorithm used in signal processing. Even though at this point the algorithms have been optimized with the simple application of induction rules, one of our future work concerns the automatic definition of the granularity of operations that must be memorized, since this has a direct impact on the size of the final memory. Moreover, we plan to apply other software-based techniques, and balance them with some protection at the hardware level, to achieve 100% fault coverage with minor area, performance and power overhead.

References

[1] Constantinescu C. Trends and challenges in VLSI circuit reliability. IEEE Micro 2003;23(4):14–9. July.
[2] McKee SA. Reflections on the memory wall. Proc Conf Comput Front 2004:162.
[3] Chakraborty K, Mazumder P. Fault-tolerance and reliability tech-niques for high-density random-access memories, ed. Prentice Hall PTR, ISBN 978-0130084651; 2002.
[4] Koren I, Koren Z. Defect tolerance in VLSI circuits: techniques and yield analysis. Proc IEEE 1998;86(9):1819–38. September.
[5] Stone HS. A logic-in-memory computer. IEEE Trans Comput 1970;C-19(1):73–8. January.
[6] Elliott D, Stumm M, Snelgrove WM, Cojocaru C, et al. Computational RAM: implementing processors in memory. IEEE Des Test Comput 1999;16(1):32–41. January.
[7] Rhod EL, Lisbôa CA, Carro L. A low-SER efficient core processor architecture for future technologies. In: Proc Conf Des, Autom Test Eur. April; 2007.
[8] Wolfe M. Beyond induction variables. In: ACM SIGPLAN'92 Conf Program Lang Des Implement; 1992. p. 162–74.
[9] Van Engelen RA. Efficient symbolic analysis for optimizing compilers. In: Proc Int Conf Comp Constr; 2001. p. 118–32.
[10] Donald M. Memo functions and machine learning. Nature 1968;218:19–22.
[11] Brandenburg K, Stoll G. ISO-MPEG-1 audio: a generic standard for coding of high quality digital audio. J Aud Eng Soc 1994;42(10):780–94. October.
[12] Painter T, Spanias A. Perceptual coding of digital audio. Tech report 85287-7206, department of electrical engineering, telecommunications research center. Arizona State University, Tempe, Arizona.
[13] Huffman DA. A method for the construction of minimum-redundancy codes. In: Proc I.R.E; 1952. p. 1098–102.
[14] Oppenheim A, Schafer R. Discrete-time signal processing. 2nd ed., Prentince Hall. p. 541–99.
[15] Stott E, Sedcole P, Cheung P. Fault tolerant methods for reliability in FPGAs. In: Proc Int Conf Field Program Logic Appl. September; 2008. p. 415–20.
[16] Wada Y et al. A 128Kb SRAM with soft error immunity for 0.35 mm SOI-CMOS embedded cell arrays. In: Proc IEEE Int SOI Conf; 1998. p. 127–8.
[17] Cha H, Patel JH. Latch design for transient pulse tolerance. In: Proc IEEE Int Conf Comput Des; 1994. p. 385–8.
[18] Berg M. Fault tolerance implementation within SRAM based FPGA designs based upon the increased level of single event upset susceptibility. In: Int On/Line Test Symp; 2006.
[19] Durand S et al. FPGA with self/repair capabilities. International work-shop on field programmable gate arrays; 1994. p. 1–6.
[20] Aho A, Sethi R, Ullman J. Compilers: principles, techniques, and tools. Addison Wesley Publishing Company, Reading; 1986.
[21] Gupta S, Miranda M, Catthoor F, Gupta R. Analysis of high-level address code transformations for programmable processors. In: Proc Conf Des, Automat Test. Europe; 2000. p. 9–13.
[22] Baradaran N, Diniz P. A compiler approach to managing storage and memory bandwidth in configurable architectures. ACM Trans Des Autom Electron Syst 2008;13(4):1–26.
[23] Citron D, Feitelson D. Hardware memoization of mathematical and trigonometric functions, technical report. Hebrew University of Jerusalem, March; 2000.
[24] Sodani A, Sohi G. Dynamic instruction reuse. Proc Int Symp Comput Arch 1997:194–205.
[25] Quach N. High availability and reliability in the titanium processor. IEEE Micro 2000;20(5):61–9. September.
[26] Atallah MJ, Kosaraju SR, Larmore LL, Miller GL, Teng S-H. Constructing trees in parallel. Proc Symp Paral Algor Arch 1989:499–533.
[27] Pin JE. Tropical semirings. Gunawardena J. (editor), Idempotency. Cambridge University Press; 1998. p. 50–69.
[28] Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M. Soft-error detection using control flow assertions. In: Proc IEEE Int Symp DFT in VLSI Syst; 2003. p. 581–88.
[29] Huang KH, Abraham JA. Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput 1984:518–28.
[30] Oh N, Shirvani PP, McCluskey EJ. Control flow checking by software signatures. IEEE Trans Reliab; 2002. p. 111–12.
[31] Azambuja JR, Sousa F, Rosa L, Kastensmidt FL. The limitations of software signature and basic block sizing in soft error fault coverage. In: Proc IEEE Latin–American Test Workshop; 2010.
[32] Azambuja JR, Pagliarini S, Rosa L, Kastensmidt FL. Exploring the limitations of software-based techniques in see fault coverage. J Electron Test 2011.
[33] Hangout LMOSS, Jan S. The minimips project. opencores.org/projects.cgi/web/minimips/overview; 2010.