Algorithm Transformation Methods to Reduce Software-only Fault Tolerance Techniques' Overhead

José Rodrigo Azambuja¹, Gustavo Brown², Fernanda Lima Kustensmidt¹, Luigi Carro¹

¹ Instituto de Informática – PPGC – PGMICRO
Universidade Federal do Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil
{jfazambuja, mascarpato, flklima} @ inf.ufrgs.br

² Facultad de Ingeniería
Universidad de la República
Av. 18 de Julio 1968, Montevideo, Uruguay
Gbrown @ fing.edu.uy

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 an MP3 player, combined with a fault injection campaign, show that this approach increases fault tolerance up to 92%, without any performance degradation.

Keywords—algorithm transformation; fault tolerance; software-only techniques

I. INTRODUCTION

Future technologies will be much more unreliable and, at the same time, the performance gap between memory and processors will not get any smaller [1]. Memories have long been protected against multiple fabrication defects [2-3]. Hence, thanks to their regularity, memories would be a natural fabric to help one cope with unreliable technologies, but 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 [4]. Furthermore, the same strategy that favors reliability also favors parallelism. The main idea is to analyze a given algorithm and, using induction variables analysis [5-6] and other related tools like memoization [7], 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%.

II. 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 [8]. Aside from the ongoing efforts on fault avoidance [9-10], current fault tolerance techniques rely on space or time redundancy to provide fault tolerance [11-12] which, for TMR, triplicates the amount of space/time required.

The use of static analysis tools like induction variables analysis is very common in the compiler construction area [6, 13] 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 [14-15]. Memoization, 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 [16-17]. 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 [3, 18]. 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 [2]. Furthermore, with the introduction of magnetic and ferroelectric RAMs, the soft error rate of such devices dumped near zero [4]. 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.
III. 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.

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 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 found on literature. Note that at this point the hardware designer has quite some flexibility regarding the level of protection 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, and the hardware must also be protected. This can be accomplished by applying traditional fault tolerance techniques.

IV. 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 heavily based on matrix operations, which 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.

A. Case Study 1: MDCT Algorithm Optimization

We first work on the MDCT problem, defined as [19]:

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

where for MP3 coding, N is either 12 or 18. Note that in the second term the cosine 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

\[ M(i,j) = \cos \left[ \frac{\pi}{N} \left( i + \frac{1}{2} + \frac{N}{2} \right) \left( j + \frac{1}{2} \right) \right]. \]

B. Case Study 2: Huffman Algorithm Optimization

We will focus on another of the algorithms used by an MP3 coder, the Huffman code [20]. The Huffman coding problem is a variable length entropy encoder which derives an optimal weighted 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 weighted 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 \( O(n \log n) \) 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 [21]. Moreover, instead of constructing the Huffman code we optimize the construction of the height bounded subtree [21] from which the actual Huffman code can be derived.

All the arithmetic is done using the tropical semiring [22]. Initially two matrices \( S \) and \( A \) are initialized with appropriate values using the non-decreasing vector of input probabilities. Then, 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 using 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, reducing the loop overhead. 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, the initialization of the S matrix and the initial value for min are slightly modified.

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.

C. Implementation Notes

We developed implementations of the two algorithms discussed for both versions (naïve and optimized algorithms). Tab. I shows a comparison between the naïve and optimized implementations, while Tab. II presents the instruction count for each implemented version. For the MDCT algorithm, 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 Huffman algorithm case, both versions utilize the same amount of memory to hold the auxiliary matrices, although the data stored in memory is slightly different. Program size of the optimized version shows a 25% reduction and execution time shows a 6% improvement.

### TABLE I. NAIVE AND OPTIMIZED ALGORITHM IMPLEMENTATIONS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Naïve</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDCT</td>
<td>60</td>
<td>19</td>
</tr>
<tr>
<td>Memory size (bytes)</td>
<td>1600</td>
<td>2592</td>
</tr>
<tr>
<td>Execution Time (ms)</td>
<td>2.05</td>
<td>0.64</td>
</tr>
<tr>
<td>Huffman</td>
<td>104</td>
<td>76</td>
</tr>
<tr>
<td>Program size (instructions)</td>
<td>3072</td>
<td>3136</td>
</tr>
<tr>
<td>Execution Time (ms)</td>
<td>1.59</td>
<td>1.49</td>
</tr>
</tbody>
</table>

### TABLE II. INSTRUCTION COUNT OF ALGORITHM IMPLEMENTATIONS

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Naïve</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>23059</td>
<td>5257</td>
</tr>
<tr>
<td>Branches</td>
<td>667</td>
<td>667</td>
</tr>
<tr>
<td>Memory access</td>
<td>1314</td>
<td>1314</td>
</tr>
<tr>
<td>Total</td>
<td>27983</td>
<td>7885</td>
</tr>
<tr>
<td>Huffman</td>
<td>8255</td>
<td>8552</td>
</tr>
<tr>
<td>Branches</td>
<td>4532</td>
<td>3843</td>
</tr>
<tr>
<td>Memory access</td>
<td>2972</td>
<td>2449</td>
</tr>
<tr>
<td>Total</td>
<td>16797</td>
<td>15525</td>
</tr>
</tbody>
</table>

V. 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 [23-25]. 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 Tab. I. The hardening transformation will be performed using a tool called HPCT [26].

A. Case Study 1: Optimized MDCT Algorithm Hardening

According to Tab. I, the optimized MDCT algorithm executes in 0.64ms. 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 [26]. The first technique verify the program’s execution flow, while the second duplicates all instructions (except branch instructions) over replicated registers and check their consistencies and the third 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. Tab. III shows that even applying all three techniques, the execution time remained lower than the original, with a 25% reduction, from 2.05ms to 1.53ms.

B. 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.

We chose as a starting point the variables technique, due to its better fault tolerance rates [26] 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 1.59ms, at 5 registers from a total of 19. Tab. III shows the results for the optimized and protected Huffman algorithm compared to the naïve and optimized versions.

### TABLE III. COMPARISON OF NAIVE, OPTIMIZED AND PROTECTED HUFFMAN ALGORITHM IMPLEMENTATIONS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparison</th>
<th>Naïve</th>
<th>Optimized</th>
<th>Optimized and Protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDCT</td>
<td>Program size (instructions)</td>
<td>60</td>
<td>19</td>
<td>109</td>
</tr>
<tr>
<td>Memory size (bytes)</td>
<td>1600</td>
<td>2592</td>
<td>5184</td>
<td></td>
</tr>
<tr>
<td>Execution Time (ms)</td>
<td>2.05</td>
<td>0.64</td>
<td>1.53</td>
<td></td>
</tr>
<tr>
<td>Huffman</td>
<td>Program size (instructions)</td>
<td>104</td>
<td>76</td>
<td>128</td>
</tr>
<tr>
<td>Memory size (bytes)</td>
<td>3072</td>
<td>3136</td>
<td>6272</td>
<td></td>
</tr>
<tr>
<td>Execution Time (ms)</td>
<td>1.59</td>
<td>1.49</td>
<td>1.59</td>
<td></td>
</tr>
</tbody>
</table>

VI. 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 [27].
In order to perform the fault injection campaign, 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 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 both case study algorithms, three software implementations were tested: (I) Naive 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 Tab. IV.

TABLE IV. RESULTS FOR SET AND SEU FAULT INJECTION CAMPAIGN IN THE MDCT AND HUFFMAN ALGORITHMS FOR 10 THOUSAND FAULTS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(I) Naive</th>
<th>(II) Optimized</th>
<th>(III) Optimized and Protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDCT</td>
<td>1913</td>
<td>1412</td>
<td>149</td>
</tr>
<tr>
<td>Huffman</td>
<td>1056</td>
<td>1291</td>
<td>840</td>
</tr>
</tbody>
</table>

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 (Tab. IV). The Huffman algorithm presented an increase in 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 execution time of 1.59ms.

VII. CONCLUSION AND FUTURE WORK

This work presents a framework for algorithm transformation which leverages on static analysis tools like induced variables analysis and memoization 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. 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. Even though at this point the algorithms have been optimized with simple application of induction rules, one of our future work concern the automatic definition of the granularity of operations that must be memoized, 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


