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.