A Data Recomputation Approach for Reliability Improvement of Scratchpad Memory in Embedded Systems

Hossein Sayadi[1], Hamed Farbeh[2], Amir Mahdi Hosseini Monazzah[1], and Seyed Ghassem Miremadi[3]
Department of Computer Engineering
Sharif University of Technology
Tehran, Iran 11155-1639
[1]{hsayadi, ahosseini}@ce.sharif.edu, [2]farbeh@mehr.sharif.edu, and [3]miremadi@sharif.edu

Abstract—**Scratchpad memory (SPM) is extensively used as the on-chip memory in modern embedded processors alongside of the cache memory or as its alternative. Soft errors in SPM are one of the major contributors to system failures, due to ever-increasing susceptibility of SPM cells to energetic particle strikes. Since a large fraction of soft errors occurs in the shape of Multiple-Bit Upsets (MBUs), traditional memory protection techniques, i.e., Error Correcting Code (ECCs), are not affordable for SPM protection; mainly because of their limited error coverage and/or their high overheads. This paper proposes a novel algorithm that efficiently protects SPM with high error correction capability and minimum overheads. This proposed data recomputation algorithm recomputes the correct value whenever an error is detected in the SPM. The simulation results show that the proposed algorithm significantly reduces the vulnerability of SPM from 91.7% to 8.4%. Moreover, the proposed algorithm imposes no area overhead and no hardware modification, meanwhile its performance overhead is less than 1%.**

Keywords-Reliability; Embedded Systems; Error Correction; Data Recomputation; Scratchpad Memory

I. INTRODUCTION

Radiation-induced soft errors in modern embedded processors are the major source of system failures [1], [2]. Among various processor components, memory cells occupy more than 60% of the chip area and are known as the most vulnerable component to soft errors [3]. Scaling-down the transistor feature size and decreasing the critical charge in today's CMOS technology further increase the vulnerability of memory cells to soft errors. The reasons for this increasing rate of soft errors are: 1) an ever increasing probability of bit-flips in the case of a particle strike has been reported, as the memory cells can now be affected by lower energetic particle; 2) the probability of affecting more than a single cell in the case of a particle strike has extremely increased, making *Multiple-Bit Upsets* (MBUs) as important as *Single Event Upsets* (SEUs) [4]. Consequently, soft errors in on-chip memories are the major reliability concern in embedded processors.

A dominant fraction of on-chip memory in a wide range of embedded processors constitutes of software managed Scratchpad Memory (SPM). Predictability, better performance, and lower energy consumption have made SPM as an attractive alternative to conventional cache memory in these processors [5], [6], [7], [8]. Due to the high vulnerability of SPM to soft errors and high contribution of faulty SPM in system failures [9], providing a fault-tolerant SPM against soft errors is of decisive importance in embedded system designs [10], [11], [12].

As SPM operates in single clock cycle latency and is accessed almost every clock cycle [6], the overheads of traditional memory protection techniques, e.g., Single Error Correction-Double Error Detection (SEC-DED) code is an overkill for embedded processors. Moreover, these techniques have limited error correction capability; thus, correcting soft errors in the shape of MBU requires multiple SEC-DED codes per word, namely *interleaved SEC-DED*, which further increases the overheads. Previous studies on SPM reliability improvement tried to reduce the overheads of traditional Error Correcting Codes (ECCs) or improve their error correction capability [9], [10], [11], [12].

All of the previous SPM protection techniques can be categorized into two main approaches: 1) selective employment of ECCs [11], and 2) providing a duplication for SPM entries [9], [10], [12]. The main deficiency of these techniques is that all of them incur at least one of the following three disadvantages: 1) significant overheads are imposed to correct MBUs [9], [12], 2) a large fraction of SPM contents remains unprotected [10], and 3) both hardware and software modification is required [11], [12].

In this paper, a recomputation-based method is proposed to correct soft errors in SPM. The main idea underlying this method consists of four stages: 1) identifying the vulnerable data in SPM by analyzing the program behavior and data access patterns, 2) determining how vulnerable data are generated and consumed at runtime, 3) generating the corresponding recomputation subroutine and inserting corresponding control instruction in appropriate code location for each vulnerable data, and 4) linking the recomputation subroutines and the source code to produce the modified source code. The proposed algorithm utilizes parity code to detect errors in SPM and call the recomputation subroutine whenever an error is detected in SPM lines. To the best of our knowledge, there has been no prior effort to improve the reliability of SPMs through data recomputation. Therefore, this approach is the first recomputation-based error correction method in scratchpad memories. The proposed approach requires no hardware modification and is applicable to COTS (Commercial Off-The-Shelf) processors. Moreover, it significantly improves the reliability of SPM with negligible performance overhead.

The remainder of this paper is organized as follows. Previous work is briefly discussed in Section II. In Section III, we explain the basic concepts of data recomputation and then we present the details of the proposed data recomputation approach. In Section IV, the simulation system setup and results are given. Finally, Section V presents the conclusion of this study.

II. Related Work

The purpose of this paper is to protect SPM against soft errors using a data recomputation algorithm. Therefore, we categorize the previous studies into two main categories: 1) previous work on reliability improvement in SPMs, 2) data recomputation methods in SPMs. First, we briefly discuss the previous studies that focused on reliability improvement of SPMs. Then, the previous data recomputation techniques on SPMs are presented. It is notable that the aim of all previous data recomputation methods was to improve the performance and/or energy consumption of SPM; and, this paper is the first work that proposes to employ data recomputation for error correction and reliability improvement for SPM. Hence, reviewing the previous data recomputation efforts is beneficial to introduce the concept of data recomputation and present the main contributions of this paper.

A. Improving the Reliabilty of SPM

As mentioned earlier, there are few studies related to the reliability enhancement of SPMs. In [10], a compiler-based data duplication technique is proposed. The main idea of this technique is to identify a subset of live blocks and duplicate them in dead SPM blocks under the control of compiler. To minimize the performance overhead, this duplication is applied only to a subset of live blocks. The SPM lines are protected by parity code and when an error is detected, it will be replaced by the duplication for error correction, if a duplication exists. This technique cannot correct all detected errors because of the existence of data blocks without any duplication. Moreover, the algorithm provided no solution for updating the duplication and no experimental result has been reported to prove its applicability.

The work in [12] tried to protect distributed SPMs in multicore processors against radiation-induced soft errors. This method, namely *Embedded RAIDs-on-Chip* (E-RoC), is based on providing an architecture similar to RAID (Redundant Array of Independent Disks) architecture used in hard-disk drives, for the distributed SPM. In this method, the contents of SPMs are duplicated in other SPM modules based on a RAID-like mechanism. However, the SPM management module and the extra SPM accesses significantly increase the energy consumption of the SPM. To reduce the energy consumption overhead, the paper employed the aggressive voltage scaling. However, reducing the operating voltage not only leads to reduction of operating frequency, i.e., performance reduction, but also the vulnerability of SPMs against soft errors will exponentially increase.

The research performed in [9] employed a duplication technique to improve the reliability of Instruction-SPM. This technique stores a backup copy of the Instruction-SPM contents in the main memory and supposed that the SPM contains the read-only instruction, which remained unchanged during the program execution. To indicate the location of duplications in the main memory, a translation table is considered. Applying this method to Data-SPM imposes significant energy consumption and performance overheads. This is due to the fact that updating the duplications of data in main memory significantly increases the number of accesses to main memory; and, this imposes two orders of magnitude more energy consumption and latency in comparison to accessing SPM.

The work in [11] suggested a hybrid architecture for SPM that employs STT-RAM memory cells alongside of SRAM memory cell in SPM configuration. In this technique, the SPM space is partitioned into two parts: a *non-volatile* part consists of STT-RAM cells and a *volatile* part consists of SRAM cells. Due to immunity of STT-RAM cells against soft errors, allocating the STT-RAM part to vulnerable data can improve the reliability of the system. The main limitation of this technique is that it requires the modification of both hardware and software. Moreover, there may exist vulnerable data that allocated to SRAM part, which threaten the reliability of the SPM.

B. Data Recomputation Algorithms for SPM

The first research on data recomputation is presented in [13] at the title of "rematerialization". In this effort, the cost of recomputing a register value is compared with the cost of storing and reloading it from main memory and if the recomputation cost is lower, the corresponding register value will be recomputed (rematerialized). The work in [14] presented a data recomputation algorithm to enhance the performance of the processor in a non-uniform memory architecture which consists of one processor in the centre and SPM banks around the processor. The main idea of this compiler-based algorithm is to regenerate the value of a SPM that is far from the processor, by accessing its constructing data elements in the closer SPMs. The recomputation process will be executed if the cost of accessing to closer SPMs and recomputing the data is less than directly accessing to the farther SPM and reading the data.

In [15], a data recomputation technique is proposed to decrease write activities in main memory constructed by Non-Volatile Memory (NVM). This technique prevents unnecessary write operations in main memory and tries not to save the intermediate generated data in the NVM-based main memory. In this way, whenever the corresponding data needs to be accessed in later steps, it will be recomputed by accessing to its constituent necessary operands from main memory and executing the instruction required to generate the data. Since the latency and energy consumption of reading data from NVMs is significantly lower than writing data, this data recomputation technique leads to improve the performance and energy consumption of the system by minimizing the number of write operation.

The research in [16] is based on a data recomputation algorithm that is employed to reduce energy consumption of memory banks architecture. This effort utilizes energy reduction technique by deactivating unnecessary memory banks in order to reduce energy consumption of the system. Moreover, whenever a data access occurs to a low-power memory bank, this algorithm checks if the corresponding data can be recomputed by other elements of active banks or not. If this condition satisfied, the low power banks will not be turned on for accessing the data and the recomputation process will be executed by accessing to the data operands located in active banks.

It is obvious that the goal of all the above data recomputation studies is to improve the performance and/or energy consumption of the system. None of the previous researches tried to improve the SPM reliability via data recomputation.

III. Proposed Approach and Motivational Examples

In this section, we present a data recomputation technique for improving reliability of SPM. This paper is the first effort of using data recomputation to correct errors in SPM and can be the start point for a new error correction approach. In the following subsections, we first present three examples to illustrate the basic concepts and various aspects of the data recomputation mechanism and then our data recomputation algorithm will be explained in details.

A. Motivational Examples

To introduce the proposed data recomputation approach, it is needed to first clarify the general concepts of data recomputation algorithms presented in the literature. As mentioned, the goal of all the previous data recomputation algorithms is to improve the performance and/or energy consumption of the system by minimizing the number of accesses to off-chip memory. In this section, we show how the previous studies used the concept of data recomputation to reduce the number of off-chip memory accesses and how this paper utilizes this concept to improve the reliability of the SPM. To this aim, three sample codes are considered in Fig. 1, Fig.2, and Fig. 3. Each figure consists of two parts: a) the primary code without data recomputation, b) the modified version of the code after applying a data recomputation algorithm presented in the literature. In these examples, three different scenarios, which data recomputation can be applied to recompute a data instead of accessing it through memory, are considered.

In Fig. 1(a), A[i] is generated in the first loop and then it is used in the computations of E[i] in the second loop. It is assumed that arrays B, C, and E are located in SPM and array A is located in the off-chip memory. To reduce the off-chip memory accesses, data recomputation algorithms suggest replacing A[i] in the second loop with its producer elements, i.e., B[i]*C[i], instead of accessing A[i] through memory. This modification is illustrated in Fig. 1(b). Now, we consider the Fig. 1(a) from the reliability point of view and assume array A is located in SPM space. Suppose that a soft error occur in one

```
for (i=0;i<=n;i++){
        ...
   A[i]=B[i]*C[i];
        ...
     }
for (i=0;i<=n;i++){
        ...
   E[i]=4*A[i];
        ...
   }          (a)
```

```
for (i=0;i<=n;i++){
        ...
   A[i]=B[i]*C[i];
        ...
     }
for (i=0;i<=n;i++){
        ...
   E[i]=4*(B[i]*C[i]);
        ...
   }       (b)
```

Figure 1. Sample code segments 1: a) Primary code, b) Modified code segment after recomputing A[i]

```
for(i=0;i<=n;i++){

    A[i]=B[i]*C[i];
        ...
   E[i]=4*A[i];
        ...
    }
         (a)
```

```
for (i=0;i<=n;i++){

    A[i]=B[i]*C[i];
        ...
   E[i]=4*(B[i]*C[i]);
        ...
  }
         (b)
```

Figure 2. Sample code segments 2: a) Primary code, b) Modified code segment after recomputing A[i]

```
   A[0] = 1;
   A[1] = 2;
     ...
   for (i=0;i<=n;i++){
         ...
    A[i]= A[i-1]*A[i-2];
         ...
     }
         (a)
```

```
 Nothing for previous
 data recomputation
     algorithms



         (b)
```

Figure 3. Sample code segments 3: a) Primary code, b) No recomputation method in previous studies

element of array A, namely A[i], in the time interval between computing A in the first loop and consuming it in the second loop. The faulty data, i.e., A[i], can be corrected by re-executing the operations that compute A[i]. In this case, the error will be corrected by re-executing A[i]=B[i]*C[i]. The opportunity of correcting soft errors using the producing elements of the erroneous data is the main motivation of this paper. Based on this opportunity, we propose a data recomputation-oriented algorithm that is capable to correct the erroneous data.

The proposed method benefits from the producer elements of A[i] in the first loop, and it replaces the A[i] with its corresponding producer, whenever an error occurs at any element of A. It should be noted that in the proposed algorithm, the program is not rolled back to a point that the total elements of A is produced; instead, only the i^{th} faulty element of A is recomputed, which is correspond to multiplying B[i] with C[i]. In the next subsection, the mechanism of recognizing the producers of A[i] will be discussed; the details of the operations needed to recompute A[i] will be presented on the next subsection, as well.

The scenario presented in Fig. 2(a) includes a loop in which an element is produced, namely A[i], and this element is consumed by one of the next operations of the loop. Same as the scenario in Fig. 1(a), arrays B, C, and E are located in SPM and array A is in off-chip memory. The data recomputation algorithms in previous studies suggest replacing A[i] in E assignment with producer elements of A[i], i.e., B[i]*C[i], instead of directly accessing A[i]. This is illustrated in Fig. 2(b). From reliability point of view, considering the scenario of Fig. 2(a) and assuming that array A is located in SPM, our proposed error correction algorithm uses the producers of A[i] to recover from error by recomputing A[i] when an error is detected in accessing A[i]. It is notable that in this case, the proposed algorithm will not rolls back to the beginning of the loop to re-execute all the loop operations; instead, only the operations needed to regenerate the erroneous element of array A, namely A[i], will be re-executed.

Considering the scenario depicted in Figure 3(a), the current element of array A, namely A[i], is generated by the previously produced elements of A, namely A[i-1] and A[i-2]. Data recomputation algorithms presented in previous studies had no idea about this situation. In this study, the question that arises here is that whether our data recomputation approach is capable of correcting errors in this scenario or not. In the next subsection, a novel approach is presented, which enables the recomputation in such situations. It will be shown that the proposed method not only covers the situations considered in previous data recomputation studies, but also it is capable to recompute the faulty data that previous algorithms are unable to recompute.

B. Proposed Approach

In this section, the proposed data recomputation algorithm for error correction in SPM is explained in detail. The focus of this algorithm is on the data arrays accessed in loops of the program. This is because of that the SPM space is allocated to the most frequently accessed data and arrays are most frequently accessed data of the program [9], [11]; Moreover, in most of the SPM allocation algorithms and all of the data recomputation algorithms in the literature, SPM space is only allocated to the arrays. However, the proposed algorithm is not restricted to data arrays and can be applied to other data structures. The basic idea of the proposed algorithm is to analyze the program behavior, extract some information related to how the data are produced in the code, and insert some instructions to monitor the execution of the program; then, whenever a soft error occurs and an erroneous data is read, a subroutine will be called to perform the operations that recomputes the erroneous data and corrects the error.

The details of the proposed method are presented in Algorithm 1. The inputs of the algorithm are primary code in which the SPM is allocated to data arrays and the profiling information used for SPM allocation containing information about the structure and the behavior of the program, e.g., the list of functions and arrays in the program, the location of data arrays in main memory and SPM, and the SPM allocations information. The output of the algorithm is the modified version of the code that is augmented with instructions and subroutines capable of monitoring the runtime behavior of the program and correcting the erroneous SPM data.

Algorithm 1: The Proposed Data Recomputation Algorithm

```
1: Input: Primary code, Profiling information
2: Output: Modified code

3: for all arrays mapped to SPM
4:      Determine the vulnerable data arrays
//      All vulnerable arrays: A = {A_1, A_2, ..., A_n}
..........................................................................
//Determining the vulnerable arrays that can be recomputed
5: for each vulnerable array A_i
6:      Analyze the assignment statements computing the array
7:      Analyze the assignment statements consuming the array
8:      Determining whether the array elements can be
        recomputed using its producers?
// All vulnerable arrays that can be recomputed: B = {B_1, B_2, ...,
B_m}
..........................................................................
//Modify the code and insert the appropriate functions and
        instructions for data recomputation
9: for each vulnerable array B_i
10:     generate its corresponding recomputation_subroutine
11:     Insert data_indicator_instructions (DRIs) in corresponding
        code lines
12:     update the recomputation_handler_subroutine to insert the
        recomputation_subroutine in the list of available functions
..........................................................................
..........................................................................

// Error correction procedure in the case of error detection
a1: call recomputation_handler_subroutine
a2:             find the corresponding recomputation_subroutine
a3:             if recomputation_subroutine exist
                        //erroneous data can be recomputed
a4:             call the subroutine to recompute the data and
                correct the error
a5: re-execute the last program instruction, which read the
erroneous data
```

The algorithm consists of three major steps to generate the modified version of the code. These steps are as follows:

- **Step 1**: The vulnerable data arrays are determined among all data arrays mapped to SPM. Since all data are primarily located in the main memory and are transferred to SPM on demand, there is a copy of each SPM entry in the main memory. For data arrays that have not been modified in SPM, i.e., *clean data* or *read-only data*, their copy can be used for error correction [9]. Hence, we need to apply the error correction method only to modified data array, i.e., *dirty data* or *vulnerable data*. The vulnerable data are identified in this step according to accessing pattern of data (lines [3-4]).
- **Step 2**: for each vulnerable data, the producing and consuming pattern of the data is analyzed and it is determined whether the pattern matches with one of the patterns that the algorithm can make a recomputation subroutine to recover the data in faulty condition (lines [5-8]). Not all dirty data in SPM may necessarily have a recomputation subroutine; for

now, the proposed algorithm can support the most general cases of data access patterns. However, the algorithm will be improved to support a wider range of data access patterns for data recomputation.

- **Step 3**: the code is modified for all vulnerable data arrays that can be recomputed. For each data that is selected for recomputation, a *recomputation subroutine* is generated that contains the operations required to generate an element of the corresponding array. The inputs of this subroutine determine for which array element the recomputation operation should be performed. The instructions required to update some runtime information needed for recomputation subroutine, named *data indicator instructions (DRI)*, are inserted in appropriate code locations. The algorithm consists a head subroutine, called *recomputation handler subroutine*, which manage the recomputation subroutines of all data arrays and call the corresponding recomputation subroutine whenever an error correction operation is required. This manager subroutine should be updated whenever a new recomputation subroutine is generated. Lines [9-12] of the algorithm indicate these processes.

According to the proposed algorithm, the modified version of the scenario presented in Fig. 3(a) is illustrated in Fig. 4(a) and Fig. 4(b), which is capable to recompute the faulty elements of array A whenever an error is detected. The error may be detected in the operation that produces the current element of array A, namely A[i]. This error is detected when reading one the producers of A[i], i.e., A[i-1] and A[i-2]. For each of these two elements, data can be reproduced (recomputed) using the operations in recomputation subroutine depicted in Fig. 4(b). The scenarios depicted in Fig. 1(a) and Fig. 2(a) can be modified according to the proposed algorithm, as well.

IV. Experimental Evaluation

A. System Setup

In this section, the experimental evaluation of the proposed data recomputation algorithm is presented. To evaluate the proposed approach, *FaCSim*, a cycle-accurate ARM processor simulator is employed [17]. The SPM size and its latency are considered 4KByte and 1 clock cycle, respectively. To represent the efficiency of our approach, experiments are performed on a set of workloads from MiBench benchmarks suite [18]. The SPM management scheme is considered similar to the one presented in [11].

B. Simulation Results

1) Reliability: We evaluated the reliability of SPM by measuring the vulnerability of data residing in the SPM. Based on Architectural Vulnerability Factor (AVF) of cache memory presented in [19], we propose the SPM version of vulnerability factor as (1):

$$VF_{SPM} = \frac{\sum residency\ time\ of\ vulnerable\ data\ in\ SPM}{Total\ Execution\ Time\ \times M} \quad (1)$$

```
A[0] = 1;
A[1] = 2;
   ...
DRI;
for (i=0;i<=n;i++){
  ...
  A[i]= A[i-1]*A[i-2];
   ...
 }
```

(a)

```
Recomp_Func(j)
 {
if (j=0) then
  A[j]=1;
else if (j=1) then
  A[j]=2;
else
  A[j]= A[j-1]*A[j-2];
 }
```

(b)

Figure 4. Modified code with the proposed data recomputation algorithm a) Inserting data recomputation indicator, b) Recomputation subroutine for A[i-1] and A[i-2]

Where VF_{SPM} is the Vulnerability Factor of SPM in normal operation and M is the total data size in the SPM.

Note that the above-mentioned formulation indicates the vulnerability factor of SPM without applying the proposed data recomputation approach. To evaluate the reliability of SPM after performing our data recomputation algorithm, the vulnerability factor of SPM (VF_{SPM-R}) can be determined as (2):

$$VF_{SPM-R} = \frac{\sum residency\ time\ of\ unrecomputed\ data}{Total\ Execution\ Time\ \times M} \quad (2)$$

Where VF_{SPM-R} is the vulnerability factor of SPM after applying the recomputation algorithm. As can be seen in the VF_{SPM-R} equation, to calculate the vulnerability factor of SPM after applying the proposed recomputation approach, we need to specify the life time of data residing in the SPM that remain unprotected without our recomputation algorithm.

Fig. 5 depicts the simulation results for vulnerability of different benchmarks. The results show promising vulnerability reduction from 91.7% in the baseline SPM to 8.4% for the proposed algorithm. As can be observed from the figure, the vulnerability of FFT reduced from 75.2% to 25.1%. Applying the proposed algorithm to Sha and Dijkstra provides significant reduction in the vulnerability of SPM by decreasing the vulnerability from 99% and 97% to 0%, respectively. This considerable improvement indicates that Sha and Dijkstra get fully protected using our data recomputation algorithm.

2) Performance: In the next set of experiments, the performance overhead of the recomputation approach is measured. Recall that the propsoed algorithm is based on inserting some redundant instructions into the original program to perform the recomputations and protect the data in SPM against soft errors. A subset of these redundant instructions is executed only when an error is detected, while the other subsets of instructions are executed at the normal system operation. The subset of redundant instructions that execute at the normal system operation can degrade the performance of the system.

Fig. 6 illustrates the normalized performance overhead of proposed algorithm in comparison to the non-protected SPM. As depicted, the worst-case performance overhead is experienced in Dijkstra, which is almost 2%. The performance overheads of FFT and Sha are about 0.5% and 0.3%, respectively. Accordingly, the average performance overhead of the proposed algorithm is less than 1%. In contrast with the 83.3% vulnerability reduction of the SPM, the performance

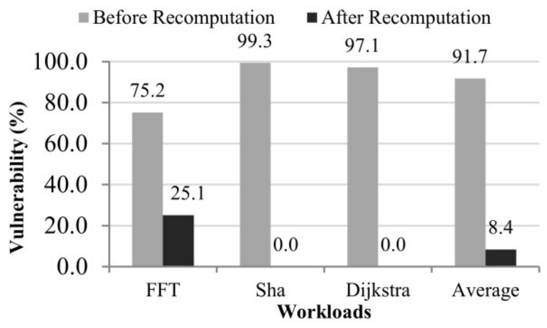

Figure 5. Vulnerability results with and without proposed data recomputation algorithm for different benchmarks

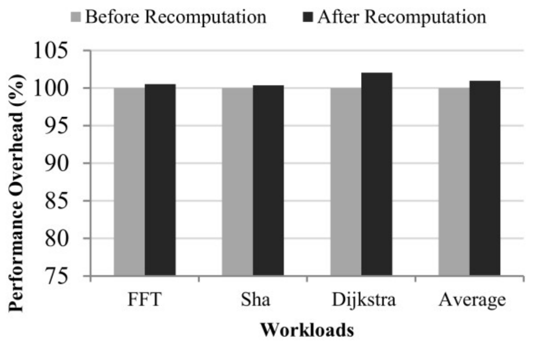

Figure 6. Performance overhead of proposed data recomputation algorithm

overhead of the proposed algorithm is negligible. On the other hand, since the proposed algorithm manipulates the program codes, considering to the appropriate size for the SPM, it imposes no area overhead to the system and no hardware modification is required. Moreover, since the hardware structure remains unchanged, the proposed approach has no effect on the power consumption of the processor, as well.

V. CONCLUSION

In this paper, a data recomputation approach was proposed to protect SPM against soft errors. This recomputation-based algorithm recomputes the erroneous data whenever an error occurs in the SPM. According to the simulation result, by applying proposed data recomputation technique the vulnerability of the SPM to soft error is 83.3% lower than the baseline SPM. In addition to providing enhanced error correction capabilities, our approach imposes no area overhead and no hardware modification to the system, which make it a promising approach for reliability improvement of SPM in COTS applications. Furthermore, the performance overhead of our approach is less than 1%.

REFERENCES

[1] R. Baumann, “Radiation-induced soft errors in ad-vanced semiconductor technologies,” IEEE Transactions on Device and Materials Reliability, vol. 5, no. 3, pp. 305–316, September 2005.

[2] R. Jeyapaul and A. Shrivastava, “Smart cache cleaninig: energy efficient vulnerability reduction in embedded processors,” Proc. IEEE International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 11), pp. 105-114, Taipei, October 2011.

[3] International Technology Road-map for Semiconductors (ITRS), “ERD ERM 2010 final report memory assessment,” Final report, 2010.

[4] A. Dixit and A. Wood, “The impact of new technology on soft error rates,” Proc. IEEE International Reliability Physics Symposium (IRPS 11), pp. 5B.4.1-5B.4.7, USA, April 2011.

[5] P. Marwedel, Embedded systems design, Second edition, Springer, 2010.

[6] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel, “Reducing energy consumption by dynamic copying of instructions on to on-chip memory”, Proc. IEEE International Symposium on Systems Synthesis (ISSS 02), pp. 213-218, USA, October 2002.

[7] D. P. Volpato, A. K. I. Mendonca, L. C. V. Dos Santos, and J. L. Güntzel, “A post-compiling approach that exploits code granularity in scratchpads to improve energy efficiency,” Proc. IEEE Computer Society Symposium on VLSI (ISVLSI 10), pp.127-132, Greece, July 2010.

[8] A. Janapsatya, S. Parameswaran, and A. Ignjatovic, “Hardware/software managed scratchpad memory for embedded system”, Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD 04), pp. 370-377, USA, November 2004.

[9] H. Farbeh, M. Fazeli, F. Khosravi, and S. G. Miremadi, “Memory mapped SPM: protecting instruction scratchpad memory in embedded systems against soft errors,” Proc. European Dependable Computing Conference (EDCC 12), pp. 218-226, Romania, May 2012.

[10] F. Li, G. Chen, M. Kandimer, and I. Kolcu, “Improving scratchpad memory reliability through compiler-guided data block duplication,” Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD 05), pp.1002-1005, USA, November 2005.

[11] A. M. Monazzah, H. Farbeh, S. G. Miremadi, M. Fazeli, and H. Asadi, “FTSPM: A Fault-Tolerant ScratchPad Memory”, Proc. Dependable Systems and Networks (DSN) , pp. 1-10, Hungary, June 2013.

[12] L. A. D. Bathen and N. D. Dutt, “E-RoC: embedded RAIDs-on-chip for low power distributed dynamically managed reliable memories,” Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE 11) , pp. 1-6, France, March 2011.

[13] P. Briggs, K. D. Cooper, and L. Torczon, “Rematerialization,” Proc. Conference on Programming Language Design and Implementation, USA, 1992.

[14] H. Koc, M. Kandemir, and E. Ercanli, “Exploiting large on-chip memory space through data recomputation,” Proc. System-on-Chip Conference (SOCC 10), pp. 513-518, USA, September 2010.

[15] J. Hu, W. Tseng, Xue, C. J. Xue, Q. Zhuge, Y. Zhao, and E. H. Sha, “Write activity minimization for nonvolatile main memory via scheduling and recomputation,” IEEE Transactions on Computer-Aided Design of Integrated Circiuts and Systems (TCAD 11), vol. 30, no. 4, April 2011

[16] H. Koc, O. Ozturk, M. Kandemir, and E. Ercanli, “Minimizing energy consumption of banked memories using data recomputation,” Proc. International Symposium on Low Power Electronics and Design (ISLPED 06), pp. 358-361, Germany, October 2006.

[17] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, S. Y. Han, “FaCSim: a fast and cycle-accurate architecture simulator for embedded systems,” Proc. ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 89-99, USA, June 2008.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T . Mudge, and R. B. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” Proc. International Workshop of the Workload Characterization (WWC 01), pp. 314, USA, December 2001.

[19] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, “A systematic methodology to compute the architec-tural vulnerability factors for a high-performance microprocessor,” Proc. IEEE/ACM International Symposium. on Micro-architecture (MICRO 03), pp. 29-40, 2003.