Multiple-Bit Upset Protection in Microprocessor Memory Arrays Using Vulnerability-Based Parity Optimization and Interleaving

Michail Maniatakos, *Member, IEEE*, Maria K. Michael, *Member, IEEE*, and Yiorgos Makris, *Senior Member, IEEE*

***Abstract*— We propose a technology-independent vulnerability-driven parity selection method for protecting modern microprocessor in-core memory arrays against multiple-bit upsets (MBUs). As MBUs constitute over 50% of the upsets in recent technologies, error correcting codes or physical interleaving are typically employed to effectively protect out-of-core memory structures, such as caches. Such methods, however, are not applicable to high performance in-core arrays, due to computational complexity, high delay, and area overhead. Therefore, we investigate vulnerability-based parity forest formation as an effective mechanism for detecting errors. Checkpointing and pipeline flushing can subsequently be used for correction. As the optimal parity tree construction for MBU detection is a computationally complex problem, an integer linear program formulation is introduced. In addition, vulnerability-based interleaving (VBI) is explored as a mechanism for further enhancing in-core array resiliency in constrained, single parity tree cases. VBI first physically disperses bitlines based on their vulnerability factor and then applies selective parity to these lines. Experimental results on Alpha 21264 and Intel P6 in-core memory arrays demonstrate that the proposed parity tree selection and VBI methods can achieve vulnerability reduction up to 86%, even when a small number of bits are added to the parity trees.**

***Index Terms*— Architectural vulnerability factor (AVF), interleaving, memory array, modern microprocessor, optimization, parity.**

I. Introduction

RECENT radiation-induced soft error rate (SER) scaling trends show that, while the single-bit SER for static RAMs (SRAMs) continues to decrease and the error rate for sequential and static combinational devices has not changed, the multibit SER has increased dramatically [1]. In the presence of a multiple-bit upset (MBU), two or more physically adjacent SRAM bits are upset by a single neutron particle [2]. During an MBU, multiple bit errors in a single word, as well as single bit errors in multiple adjacent words, can be introduced [3]. As contemporary memory structures exhibit an increasing multibit failure rate, the importance of MBU analysis has been highlighted in several recent publications [4], [5]. Dixit and Wood [4] emphasize that MBUs have become much more frequent due to shrinking feature sizes, and MBU protection in microprocessors has become a necessity. MBU statistics are necessary to assess the effectiveness of error correction schemes, with recent experiments demonstrating that MBUs can affect up to eight adjacent cells [6].

Considering both single-bit upset (SBU) and MBU becomes particularly important when assessing vulnerability of modern microprocessors, as they typically include numerous in-core memory arrays to support high-performance execution. In addition to the use of SRAMs for large memory structures, such as the instruction and data caches, limited power budget also dictates the use of SRAM-based structures for various in-core memory arrays, such as the instruction queue or the register allocation table [7]. A concrete example of power savings in a microprocessor is the use of content addressable memory (CAM)/RAM-based structures [8], [9] instead of latch-based memories. These structures are built using SRAM technology, with a typical CAM cell consisting of two SRAM cells [9], and achieve power savings of 36% on average [10]. As SRAMs come at the cost of increased susceptibility to single and multiple bit errors, counter measures against radiation-induced errors need to be put in place.

Typical methodologies for MBU protection include physical interleaving [11], [12], error detection codes [13], and error correcting codes (ECCs) [14], [15]. Interleaving refers to the creation of logical checkwords from physically dispersed locations of the memory array, forcing the MBUs to appear as multiple single-bit errors, instead of a single multibit error. Checkwords are generated based on a specified ECC scheme, thus interleaved memories rely on the presence of advanced ECCs. However, while applying ECC protection to out-of-core memories (such as caches) is the state-of-the art method for resiliency enhancement, generation of checkwords at core clock speed is challenging, and comes at the cost of very high area overhead [16].

Error detection methods, however, that build on parity bits, are far less complex and may still constitute a feasible solution. While parity offers only detection capabilities, it is sufficient for in-core memory arrays of modern microprocessors as

Manuscript received November 5, 2013; revised March 22, 2014 and September 19, 2014; accepted October 21, 2014. Date of publication November 11, 2014; date of current version October 21, 2015.

M. Maniatakos is with the Department of Electrical and Computer Engineering, New York University Abu Dhabi, Abu Dhabi 129188, United Arab Emirates (e-mail: michail.maniatakos@nyu.edu).

M. K. Michael is with the Department of Electrical and Computer Engineering, University of Cyprus, Nikosia 1678, Cyprus (e-mail: mmichael@ucy.ac.cy).

Y. Makris is with the Department of Electrical Engineering, University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: yiorgos.makris@utdallas.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2014.2365032

other correction mechanisms, such as pipeline flushing and checkpoint restoring, can be applied after the fault has been detected. Yet, blindly applying parity across the board not only incurs significant area, power, and delay overhead but may also reduce the achieved coverage. Furthermore, parity protection of certain bits is unnecessary, as they may, ultimately, have a low probability of affecting the application outcome. Instead, similar to the low-overhead parity selection optimization methods that were introduced in different contexts in [17] and [18], judicious parity construction is necessary to minimize vulnerability. As we discuss in Section II, optimal parity selection for MBU detection is not straightforward and simple heuristics yield suboptimal results. Thus, toward a comprehensive solution, in Section III, we formulate the problem as an integer linear program (ILP). Section IV presents a method for rearranging the position of certain bit fields, called vulnerability-based interleaving (VBI), to enhance the efficiency of the ILP solution for one parity tree. Section V describes the modeling language, the solver, and the in-core memory arrays used in this paper and the results are presented in Section VI, followed by the conclusion in Section VII.

II. Selective Parity

While parity is a potentially viable option for protecting in-core memory arrays, adding all bits to a single parity tree is not a good idea for the following two reasons.

1) In-core memory arrays in modern microprocessors are typically quite wide, to store all the information needed for out-of-order instruction execution. For example, the information appended to an instruction word in the Alpha 21264 ranges between 160 and 290 bits [19]. Since up to 32 instructions can be in-flight, the microprocessor employs several large in-core memory arrays to support the pipelined execution engine. Hence, adding parity trees for all the bits in each word of these memory arrays would incur significant overhead in terms of area, power consumption, and delay.
2) More importantly, such a parity tree would only detect MBUs causing an odd number of errors. Therefore, the parity scheme would fail to detect 2- and 4-bit MBUs, which constitute a significant portion of current MBU distributions.

Evidently, connecting only a carefully selected subset of bits to the parity tree might yield better overall protection from MBUs. Moreover, not all bits in such words are equally critical. Indeed, since the type of information stored by each bit is known in advance, we can characterize *a priori* its relative importance and vulnerability. For this purpose, we can use the architectural vulnerability factor (AVF) of each bit, which was first introduced in [20] and which reflects the probability that a bit flip will cause a system-visible error. Consider, for instance, the word of a sample 8-bit memory array, shown in Fig. 1, and let us assume that bit i_0 has an AVF of 0.5. In this case, only half of the faults in this particular bit will affect the end user. Similarly, let us assume that bit i_2 has an AVF of 0, therefore no faults affecting it can produce a visible error. An example of such a case could be a memory array storing information related to branch prediction, which is used to populate the branch history table. The impact of a fault affecting bit i_2 would only result in a different prediction and, thus, a possible performance penalty (or gain), but not to a system-visible error.

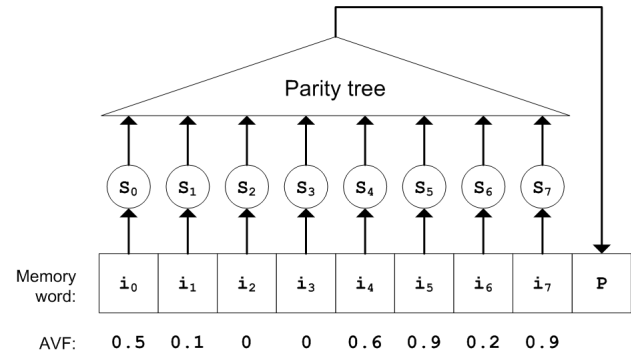


Fig. 1. Example of parity selection for protecting memory words.

By considering the vulnerability of each bit, we can select the most appropriate subset of bits to add to the parity tree, effectively introducing an AVF-driven parity optimization method. In our example, since bits i_2 and i_3 have an AVF of 0, including them in the parity tree is unnecessary. In other words, a parity tree including the remaining 6 bits would be equally effective as a parity tree including all 8 bits, yet it would incur less area, power, and delay overhead. In addition, note that bit i_6, which has a very low AVF of 0.2, is adjacent to bit i_7, which has a very high AVF of 0.9. This implies that MBUs, which affect both of these bits will be masked. Leaving i_6 out of the tree will enable detection of such MBUs that have a high probability of becoming visible to the system, at the cost of allowing single errors on i_6 to propagate (with low probability) to the system level. In other words, careful AVF-driven selection of bits to include in the parity tree can be beneficial both in terms of overhead and in terms of coverage.

Nevertheless, any such single parity tree will continue to be ineffective in detecting MBUs that affect an even number of bits among those connected to the tree. To alleviate the problem, a possible solution is the addition of *multiple* parity trees. In our previous example, if a 2-bit wide upset affects bits i_0 and i_1, it will propagate undetected; however, if i_0 and i_1 are connected to different parity trees, an error affecting both will be detected separately by the two parity bits. Adding more parity trees comes at the cost of an extra parity bit, which needs to be stored per word. However, assuming that the total number of bits connected to the parity trees is the same, it does not require additional XOR gates. It speeds up parity computation since the depth of each tree is smaller than that of a single tree. We also note that the maximum number of trees that one should consider does not exceed the maximum width of the expected MBUs. Indeed if, for example, a single event upset affects at most two adjacent bits of a memory word, addition of a third parity tree is superfluous since all the pairs of potentially erroneous bits can be split into the two trees. The same holds true when there are no adjacent bits with high AVF, essentially implying that a very small number of parity trees will suffice.

To summarize the problem, given: 1) the expected MBU distribution; 2) the vulnerability of the bits; 3) the

Algorithm 1 Simple Algorithm

```
Input: AVF_i, b, t
/* S is a k × t matrix, with (S_{i,r} = 1) ⇒ i-th bit is
   connected to r-th tree                                  */
Output: S
/* Next tree to assign                                     */
1 nexttree = 0;
/* Sort bits by vulnerability and store the first b        */
2 bits = Sort(AVF)[0:b];
/* Sort again based on index, so bits will be in order     */
3 bits = Sort(i);
4 for i = 0; i < b; i++ do
5 |   S_{bits[i],nexttree} = 1;
6 |   nexttree = ((nextree + 1) % t);
7 end
```

maximum number b of bits that the budget allows to connect to parity; and 4) the number t of available parity trees, we seek to choose which bits to add to each of the parity trees to maximize the protection of the memory word from MBUs.

A. Simple Algorithm

A straightforward algorithm for selecting, which bits to add to each of the parity trees is shown in Algorithm 1. The main idea behind this greedy heuristic is to select and place the b most vulnerable bits to the t parity trees in a round-robin fashion. In the example shown in Fig. 1, for $b = 5$ and for $t = 2$, bits $\{i_0, i_4, i_5, i_6, i_7\}$ will be selected, with $\{i_0, i_5, i_7\}$ connected to the first parity tree and $\{i_4, i_6\}$ to the second.

This simple algorithm, however, may yield suboptimal results, especially since it does not consider the distribution of MBU faults. For instance, a 4-bit wide upset affecting bits $\{i_4, i_5, i_6, i_7\}$ will go undetected as both parity trees will experience an even number of errors. Given the high AVFs of the corresponding bits, this MBU will most likely affect the user. However, if i_6 was omitted from the trees, this particular MBU would be detected, although the word would now be vulnerable to SBUs affecting bit i_6. As the latter has a very low AVF (0.2), its exclusion can give better results than the previous configuration. Evidently, a wider range of solutions need to be explored to obtain the optimal subset.

Given a word of k bits and a budget of b bits to connect to parity trees, the size of the solution space for a single parity tree is $\binom{k}{b}$. In a commercial design, such as the Alpha 21264 that has a $k = 219$-bit instruction queue memory array, this implies that with a budget of $b = 44$ bits (20%), the number of possible solutions is $\binom{219}{44} \approx 3.47e46$. This space increases dramatically as more trees are added. This huge solution space, in combination with the inability of simple heuristics to provide an optimal solution to this cover-like problem (as further demonstrated in Section VI), pinpoint the need for a more general solution. To this end, in the following section, we formulate vulnerability-based parity selection as an ILP and we use dedicated ILP solvers for approximating the optimal solution.

III. Formulation of Parity Optimization ILP

In this section, we demonstrate the necessary steps to formulate the parity selection optimization problem as an ILP. We first introduce the ILP formulation in Section III-A, and then explain the derivation of the cost function, first in an intuitive nonlinear form in Section III-B, which we then linearize in Section III-C.

A. ILP Formulation

The goal of the parity optimization problem is to minimize the vulnerability of the in-core memory array. Since we add parity per memory word, the developed cost function will refer to each individual word. Thus, to formulate the cost function to be minimized, called memory word vulnerability factor (MWVF), we define the following ILP.

Given the parameters:

1) k: number of bits in the memory word;
2) V_i: AVF of bit i, $V_i \in [0, 1]$, $i \in \{1, 2, \ldots, k\}$;
3) d: maximum MBU distance, defined by specified fault model;
4) P_j: probability of a j-wide BU, defined by fault model, $P_j \in [0, 1]$, $j \in \{1, 2, \ldots, d\}$, $\sum_{j=1}^{d} P_j = 1$;
5) t: the number of parity trees, $t \geq 1$;
6) b: maximum number of bits to be added to the parity trees, $1 \leq b \leq k$.

Solve for:

1) $S_{i,r} \in \{0, 1\}$;
2) $y_{i,j,m,r} \in \{0, 1\}$;
3) $x_{i,j,m,r} \in \{0, 1, \ldots, \lfloor j/2 \rfloor\}$;
4) $z_{i,j,m} \in \{0, 1, \ldots, t-1\}$;
5) $w_{i,j,m} \in \{0, 1\}$ in the domain $i \in \{1, 2, \ldots, k\}$, $j \in \{1, 2, \ldots, d\}$, $m \in \{1, 2, \ldots, j\}$, $r \in \{1, 2, \ldots, t\}$.

Minimize cost function

$$\sum_{i=1}^{k} V_i \sum_{j=1}^{d} \frac{P_j}{j} \sum_{m=1}^{j} w_{i,j,m} \tag{1}$$

subject to constraints:

1) $\sum_{i=1}^{k} \sum_{r=1}^{t} S_{i,r} \leq b$;
2) $\sum_{n=1}^{j} S_{i-j+m+n-1,r} = 2x_{i,j,m,r} + y_{i,j,m,r}$;
3) $\sum_{r=1}^{t} (1 - y_{i,j,m,r}) = t * w_{i,j,m} + z_{i,j,m}$.

B. Formulating Cost Function

Let us now explain how this cost function was derived and why it reflects the choice and distribution of bits to parity trees, which minimizes vulnerability of an in-core memory word to MBUs. This vulnerability, which we termed MWVF, is defined as the sum of the individual MBU vulnerabilities of all bits in the word. The vulnerability of each individual bit is defined as the product of the bit AVF (V_i) multiplied by the probability that a j-wide MBU will affect bit i. Thus, the initial formulation of MWVF is the following:

$$\sum_{i=1}^{k} V_i * (\text{probability of a } j\text{-wide MBU affecting bit } i). \tag{2}$$

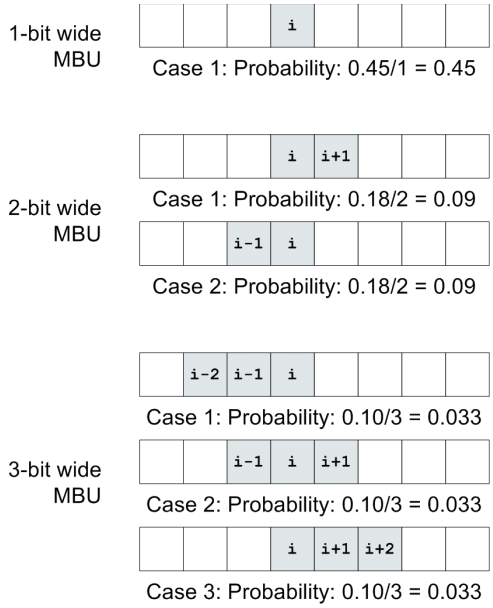


Fig. 2. Bitwise probability distribution in j-wide MBUs.

Given an MBU distribution, bit i might be part of a 1-bit wide MBU (SBU), a 2-bit wide MBU, and a 3-bit wide MBU. For example, given that bit i has two neighboring bits, a 2-bit wide MBU might affect the pair $\{i-1, i\}$ or the pair $\{i, i+1\}$. For the presented formulation, we assume that MBUs affect bits horizontally; vertical MBUs are dealt with by observing that each memory word is protected by a separate parity bit.

The MBU fault model defines how the MBUs manifest to the memory array. For instance, an MBU distribution could be [1: 0.45, 2: 0.18, 3: 0.10, 4: 0.27]. This distribution indicates that with 0.45 probability, the MBU will affect 1 bit, with 0.18 probability it will affect 2 bits, and so on. In case of a 2-bit wide MBU, a fault that includes bit i should be analyzed separately in case $i-1$ and i bits are affected, and in case i and $i+1$ are affected. This distinction is essential, because the vulnerability factor changes according to whether bit $i-1$ or $i+1$ is included in a parity tree.

In this paper, we make the assumption that the two cases have equal probability. Given the MBU distribution described earlier, we assume that the probability of a 2-bit wide MBU affecting bits $i-1$ and i is 0.09 (0.18/2), and the probability of an MBU affecting bits i, $i+1$ is also 0.09. Therefore, in case of a j-wide MBU, the probabilities are distributed equally among all cases, as shown in the example presented in Fig. 2.

Equation (2) is expanded to reflect the different cases of MBU manifestation

$$\sum_{i=1}^{k} V_i \sum_{j=1}^{d} \frac{P_j}{j} \sum_{m=1}^{j} (j\text{-wide MBU affects bit } i). \tag{3}$$

In the simple case of one parity tree ($r = 1$), a j-wide MBU will affect bit i if an even number of these j bits are protected by parity. For example, in case bits $i-1$, i, and $i+1$ are affected by an MBU, the error will be detected if $S_{i-1}+S_i+S_{i+1}$ is an odd number, implying that 1 or 3 of these bits are protected by parity. In case 0 or 2 bits are protected, the error will be masked, and will eventually affect the user with probability $V_i * P_3/3$.[1]

Therefore, in case the sum of S of the j affected bits is even, the corresponding case should be set to 1, otherwise it should be set to 0. Equation (3) now becomes

$$\sum_{i=1}^{k} V_i \sum_{j=1}^{d} \frac{P_j}{j} \sum_{m=1}^{j} \left(1 - \left(\left(\sum_{n=1}^{j} S_{i+m+n-j-1}\right) \bmod 2\right)\right). \tag{4}$$

Note that the inclusion of the *mod* operator converts the problem to nonlinear. In the following section, we present the transformations needed to linearize it.

Equation (4) assumes that only one parity tree is used. To account for multiple trees, (4) is extended to the following:

$$\sum_{i=1}^{k} V_i \sum_{j=1}^{d} \frac{P_j}{j} \sum_{m=1}^{j} \prod_{r=1}^{t} \left(1 - \left(\left(\sum_{n=1}^{j} S_{i+m+n-j-1,r}\right) \bmod 2\right)\right). \tag{5}$$

The expression $(1 - ((\sum_{n=1}^{j} S_{i+m+n-j-1,r}) \mod 2)$ is 0 when, in case of a j-wide MBU, there is at least one case (out of the j cases) that the error will be detected by parity. Therefore, the addition of the product operator ensures that the contribution of the particular case to the total vulnerability will be 0 if there is at least one parity tree that detects the corresponding fault. For instance, if only parity tree two out of three trees in total detects the tested case, the product will be $1 * 0 * 1 = 0$, implying that the error is detected by the current configuration of $S_{i,r}$ and will not contribute to the total MWVF.

Equation (5) shows the cost function that we want to minimize. Since we indicate that b out of the k bits will be added to the parity trees, the following constraint is added:

$$\sum_{i=1}^{k} \sum_{j=1}^{t} S_{i,j} \leq b. \tag{6}$$

Note that this constraint does not preclude the inclusion of a bit to multiple trees.

C. Converting Function to Linear

The inclusion of the *mod* operator, as well as the *product* operator, makes this optimization problem nonlinear. In this section, we introduce two transformations to convert it back to linear.

To remove the *mod* operator, the expression $((\sum_{n=1}^{j} S_{i+m+n-j-1,r}) \mod 2)$ is rewritten as $(y_{i,j,m,n,r})$, and the following constraints are added:

$$\sum_{n=1}^{j} S_{i-j+m+n-1,r} = 2x_{i,j,m,r} + y_{i,j,m,r}$$
$$y_{i,j,m,r} \in \{0, 1\}, x_{i,j,m,r} \in \{0, 1, \ldots, \lfloor j/2 \rfloor\}. \tag{7}$$

The variables $x_{i,j,m,r}$, $y_{i,j,m,r}$ are added to the solver. Equation (7) implies that $y_{i,j,m,r}$ will be 0 when

[1] i can be part of 3 different MBUs, as shown in the 3-bit MBU example of Fig. 2, so the probability that bits $i-1$, i, and $i+1$ are affected is $P_3/3$.

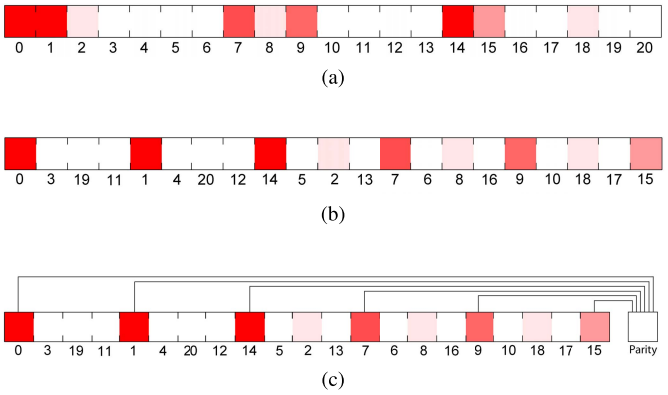


Fig. 3. Example of VBI. (a) Initial order of bit lines. (b) Order of bit lines after VBI. (c) Post-VBI word line including parity bit.

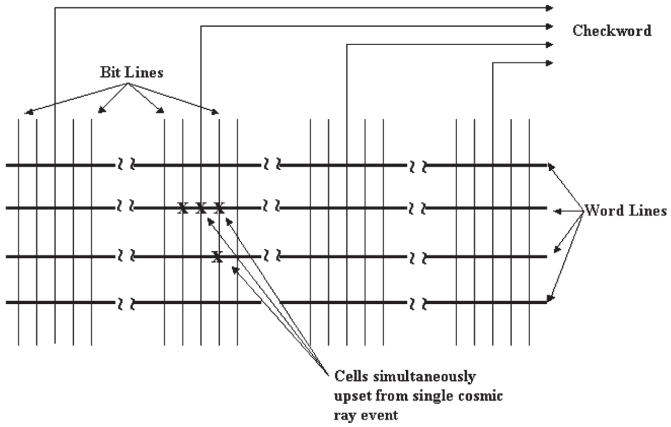


Fig. 4. Physical interleaving [11].

$\sum_{n=1}^{j} S_{i-j+m+n-1,r}$ is an even number, otherwise it will be 1. This effectively replaces the *mod* operator, and our cost function now becomes

$$\sum_{i=1}^{k} V_i \sum_{j=1}^{d} \frac{P_j}{j} \sum_{m=1}^{j} \prod_{r=1}^{t} (1 - y_{i,j,m,n,r}). \tag{8}$$

The final step of converting the cost function to linear is the removal of the product operator. Similar to the previous operation, we replace $\prod_{r=1}^{t}(1 - y_{i,j,m,n,r})$ with $w_{i,j,m}$, adding the following constraints:

$$\sum_{r=1}^{t}(1 - y_{i,j,m,r}) = t * w_{i,j,m} + z_{i,j,m}$$
$$z_{i,j,m} \in \{0, 1, \ldots, t-1\}, w_{i,j,m} \in \{0, 1\}. \tag{9}$$

Since the term $z_{i,j,m}$ is a positive number smaller than t, $w_{i,j,m}$ will be 1 *iff* $\sum_{r=1}^{t}(1 - y_{i,j,m,r}) = t$. The latter implies that all y terms are 0, thus there is no parity tree detecting the corresponding fault. If at least one tree detects the fault, its y term will be 1 and w will be forced to 0.

Therefore, the final optimization function that we feed to the ILP solver is the following:

$$\sum_{i=1}^{k} V_i \sum_{j=1}^{d} \frac{P_j}{j} \sum_{m=1}^{j} w_{i,j,m} \tag{10}$$

given the constraints [6], [7], [9].

IV. Vulnerability-Based Interleaving

There are cases where the ILP solver has limited flexibility when selecting, which bits to add to the parity trees. For example, if critical bits are clustered and only one parity tree is available, then the solver will add only one of them to the formed parity tree, limiting the vulnerability reduction achieved. Fig. 3(a) shows this case: bits 0, 1, and 2 are critical, but the solver will have to select only one to add to the single parity tree.

One parity tree is the most common case in modern memories, as the area footprint is minimal. Thus, to increase the flexibility of the ILP solver during parity optimization, we also propose a method for interleaving individual bit cells based on their probability of affecting instruction execution. Current interleaving techniques [11] include every bit in the protection scheme, regardless of the bit criticality. This is essential for out-of-core memories, where there is no *a priori* information about the contents of the bits. In the case of in-core memories, on the contrary, this information is fixed (e.g., bit 0 is the valid bit and bit 1 is the issued bit). Furthermore, techniques, such as [21] rely on spares to introduce a two-level redundancy. Spares are not an option for microprocessor arrays due to cost. Other techniques, such as 2-D error coding [12], incur prohibitive latency because of both horizontal and vertical refresh of the parity bits.

Our proposed method rearranges the position of certain bitlines in the stored word. Throughout this section, bitlines refer to the columns of a memory array (vertical lines), and word lines refer to the rows (horizontal lines) (Fig. 4). VBI aims to improve design resiliency by exploiting the fact that important bitlines are usually adjacent, rendering the memory array more susceptible to multiple bit errors. Experimental results presented in Section VI-E confirm this observation. Thus, by physically dispersing the critical bitlines and using the ILP formulation to select the optimal set of bits to add to the parity trees, VBI greatly improves the resiliency of a given design.

An example of VBI appears in Fig. 3. A common layout of the information stored in a typical out-of-order instruction queue consists of a bit for the validity of the instruction, a bit indicating whether the instruction has been issued to the functional units, bits storing the branch target of potential branch instructions, bits storing the instruction operands, and bits storing the predicted branch direction. Evidently, as also demonstrated for the Alpha 21264 microprocessor in Section VI-E, critical bits are clustered: for example, the *issue* and *valid* bits are usually the first bits stored in the instruction queue, and will certainly result in instruction corruption. However, the PC branch target information, contained in several adjacent bits up to 64 in modern microprocessors, is rarely used, thus the likelihood of errors in these bits affecting instruction execution is very low. Fig. 3(a) shows the initial layout of bit fields in the stored word, where darker coloring implies higher probability to affect workload output.

Algorithm 2 Sample VBI Algorithm

```
1  Assume A_MxN is the target memory MxN array;
2  Assume VF_MxN is the vulnerability factor for each bit
   of the array;
3  Assume AVBI_M is the output of the VBI algorithm;
4  Assume B is the percentage of bit-lines to be protected;
   /* Find bit-line vulnerability                          */
5  for i = 0; i < M; i++; do
6      VFcol[i] = 0;
7      for j = 0; j < N; j++; do
8          VFcol[i] += VF[i][j];
9      end
10 end
   /* Sort columns by vulnerability                        */
11 Sort(VFcol, key=VFcol[i]);
   /* Find the order to place the bits                     */
12 diffs = [(0, M-1)] // Table containing the range tuples
13 placed = [0, M-1] // List of placement sequence
14 while not empty(diffs) do
       /* Traverse the list and find maximum difference    */
15     maxdiff = 0;
16     for item in diffs do
17         diff = item[1] - item[0];
18         if diff > maxdiff then
19             maxdiff = diff;
20             maxdiffloc = diffs.index(item);
21         end
22     end
       /* Find median, this will be the next position      */
23     median = (diffs[maxdiffloc][1] -
       diffs[maxdiffloc][0])/2 + diffs[maxdiffloc][0];
       /* If still a range of values, split it in the median
          and append it to the table                       */
24     if diffs[maxdiffloc][1] - diffs[maxdiffloc][0] > 1 then
25         diffs.append((diffs[maxdiffloc][0], median));
26         diffs.append((median, diffs,[maxdiffloc][1]));
27         placed.append(median);
28     end
29     del diffs[maxdiffloc] // Remove split range from
         table
30 end
   /* Place the bits in the new order                      */
31 for i = 0; i < M; i++; do
32     AVBI[placed[i]] = VFCol[i];
33 end
```

A possible rearrangement based on VBI appears in Fig. 3(b). More critical bit fields are spread throughout the word, thus minimizing the probability that an MBU will affect more than one of them. With the addition of selective parity protection, as shown in Fig. 3(c), the vulnerability of the memory array is expected to decrease, as the critical bitlines are protected and the probability of an MBU affecting more than one critical bitline is very low (and dependent on the interleaving distance).

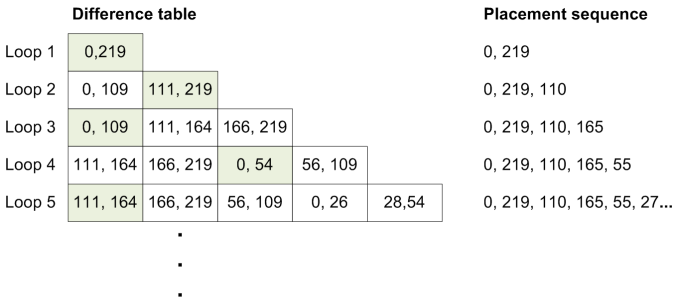


Fig. 5. Selecting the new placement of bits for VBI.

A. VBI Algorithm

The proposed method for rearranging the bitlines appears in Algorithm 2. The inputs of the algorithm consist of the memory array to be protected, the percentage *B* of bitlines to be protected with parity (given a specific resiliency budget), as well as a vulnerability figure for the individual cells of the memory array. These vulnerability figures are obtained through fault simulation, using the MBU fault model defined in Section V-D. The first step of the algorithm is to calculate the individual vulnerability factor for each vertical bitline, by summing the vulnerability factors of each bit in this bitline. Then, the algorithm ranks the list of bitlines in decreasing order of vulnerability.

The next step is to place the most critical bit fields as far from each other as possible. Intuitively, the most critical bit field will be placed in the first bit of the memory word and the second most critical to the end. The third, to be as far from the two, will be placed in the middle. For example, in a 219-bit memory word, the order to place the first three bit fields would be 0, 219, and 110. Then, the next bit field should be placed as far apart from the placed ones as possible, so that is either the median of the range [0, 109] or [111, 219]. The selection should be 165, as AVF(219) <= AVF(0). Fig. 5 shows the first steps of the application of the algorithm to the aforementioned 219-bit memory word. The green boxes denote the next range that will be split. Therefore, steps 12–30 of the algorithm are generating this sequence. The last step assigns the sorted list of bit fields to the generated sequence. While this assignment may slightly increase the routing overhead of the design, the experimental results presented in Section VI-F indicate that application of VBI incurs minimal power and no area overhead.

V. Experimental Setup

In this section, we present the experimental setup, which we used to demonstrate the effectiveness of the proposed vulnerability-based parity optimization methods.

A. Modeling Language

To model the optimization problem, we used GNU MathProg [22]. MathProg is a modeling language intended for describing mathematical programming models, and it is a subset of a modeling language for mathematical programming. The reason behind the choice of MathProg is that it

automatically translates the model description into internal data structures, which are then used to generate mathematical programming problem instances and can also be used by appropriate solvers to find a solution to the given problem. Given the nature of the optimization problem described in this paper, which includes thousands of variables and constraints, MathProg allows us to use variable domains that expand during translation, and thus to avoid the tedious effort of manually generating all the statements.

B. ILP Solver

The solver used to obtain the numeric solution to the optimization problem is solving constraint integer problems (SCIPs) [23]. Constraint integer programming is a generalization of mixed-integer programming, thus it can also be used for solving ILPs. The techniques employed by SCIP, such as Linear Programming (LP) relaxation of the problem, strengthening the LP by cutting plane separator, and analyzing infeasible subproblems to infer global knowledge, are necessary to approximate the optimal solution in a reasonable time, as memory and computational time rise exponentially as more integer variables are added.

In particular, for the presented optimization problem, we first obtain the LP relaxation of the problem by dropping the integrality restrictions of the variables, effectively providing a dual bound on the objective value. This relaxed solution is used for the bounding step of the branch-and-bound algorithm, supported by cutting plane separation. The cut separators are general purpose cuts that usually include complemented integer rounding and Gomory cuts.

Since SCIP cannot directly read the GNU MathProg language, we used `glpsol` [22] to convert the input program from MathProg to the CPLEX LP format, and then used SCIP to solve the optimization problem.

C. In-Core Memory Arrays

Two in-core memory arrays are used in this paper: The instruction queue of the Alpha 21264 instruction scheduler [24], and a memory array of the reservation station (RS) of the Intel P6 architecture.

The Alpha processor incorporates all the features present in current commercial microprocessors, such as aggressive out-of-order scheduling, a deep 12-stage pipeline, and superscalar execution. The instruction queue incorporates 32 memory words, each with a size of 219 bits. The 219 bits include bit fields of various importance, such as the critical *valid* and *issue* bits, or the less important *tag* of the program counter, which is only used for branches. The information stored in the instruction queue is shown in Table I. The first 32 bits contain the instruction word, fetched from the instruction cache. The fetch unit appends information about the location and the target of the instruction and feed the renaming logic. During the rename stage of the pipeline, several fields are added to the instruction in-flight, i.e., functional unit destination, renamed registers, and branch information. When the instruction reaches the scheduler, the reorder buffer id as well as the *issue* and *valid* bits are appended before the instruction is stored in the queue for execution.

TABLE I
BIT FIELDS OF THE INSTRUCTION STORED IN THE INSTRUCTION QUEUE

Instruction after fetch		Instruction after rename		Instruction after issue	
Name	**Bits**	**Name**	**Bits**	**Name**	**Bits**
ARCH_DEST	4:0	NEEDS_DEST_REG	161	ROBID	216:203
OP_FUNC	11:5	BRANCH	162	ISSUED	217
OPLIT	12	SIMPLE	163	VALID	218
LIT	15:13	COMPLEX	164		
SRCB	20:16	MULTIPLY	165		
SRCA	26:21	MEMORY	166		
OPCODE	31:26	WRITES_TO_DEST	167		
FETCH_PC_IN	95:32	LOAD	168		
FETCH_PCTAG_IN	159:96	USES_SRCA	169		
VALID_FETCH	160	USES_SRCB	170		
		ARCH_DEST	175:171		
		CONDITIONAL_BR	176		
		UNCONDITIONAL_BR	177		
		INDIRECT_BR	178		
		MEM_OP_SIZE	180:179		
		MEM_UN_OR_CMOV	181		
		DEST_PHYS	188:182		
		OLD_DEST_PHYS	195:189		
		SRCA_PHYS	202:196		
		SRCB_PHYS	209:203		

TABLE II
TOTAL NUMBER OF VARIABLES AND CONSTRAINTS FOR VARIOUS NUMBERS OF PARITY TREES

Number of trees	Alpha Instruction queue		P6 Reservation Station	
	Variables	Constraints	Variables	Constraints
1 parity tree	8,979	4,381	3,936	1,921
2 parity tree	13,578	6,571	5,952	2,881
3 parity tree	18,177	8,761	7,968	3,841
4 parity tree	22,776	10,951	9,984	4,801

The P6 architecture is the basis of modern Intel microprocessors. We paper an in-core memory array of the 36-slot RS, which is the instruction scheduler of the P6 out-of-order cluster. Due to a nondisclosure agreement with Intel Corporation, we do not present implementation details or absolute vulnerability factors of this in-core array. However, this is not required for this paper, as the main focus is the *relative vulnerability reduction* achieved by selecting the optimal percentage and distribution of bits to include in the parity trees.

Hierarchical statistical fault injection, as described in [25], was employed to extract the individual AVFs of the two in-core memory arrays. A three-level hierarchical design is used: scheduler, out-of-order cluster, and full chip. As accurate vulnerability analysis relies on the use of real-life applications, several benchmarks from the SPEC suite were utilized.

The number of variables and constraints for the individual ILPs appears in Table II. The large number of variables and constraints highlight the need for using an efficient ILP solver to obtain the optimal solution in a reasonable time.

D. MBU Fault Model

Typically observed fail bit patterns, as shown in Fig. 6, indicate that MBUs do not manifest as multiple bit flips spread across rows or columns; instead, they are clustered in double stripes perpendicular to the wordlines and manifest as force-to-0 or force-to-1 effects. This is attributed to the battery effect described in [26]. Fig. 7 shows a highly compact layout of bit cells widely used in the design of such arrays. Since the

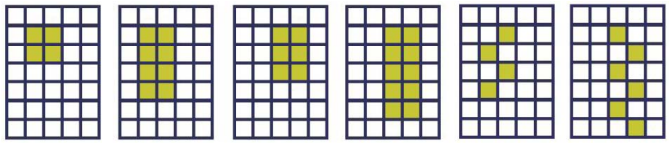


Fig. 6. Typically observed fail bit patterns [6].

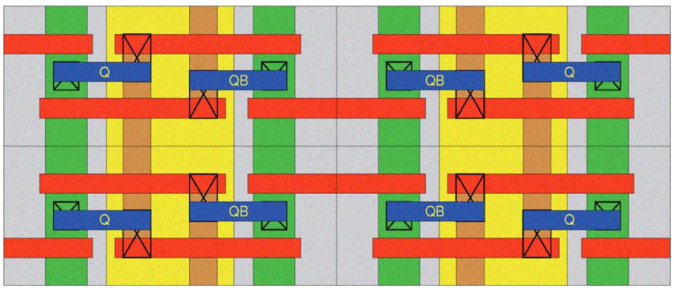


Fig. 7. Compact mirror layout of arrays [27].

TABLE III
MBU DISTRIBUTION FOR THE THREE DIFFERENT FAULT MODELS

MBU width	Fm90	Fm65	FmNew
1-bit	0.95	0.45	0.20
2-bits	0.04	0.18	0.16
3-bits	0.00	0.10	0.16
4-bits	0.01	0.27	0.16
5-bits	0.00	0.00	0.16
6-bits	0.00	0.00	0.16

p-well is shared among every pair of columns, in case a particle strikes and causes charge collection, the generated charge raises the potential of the bulk and turns on a parasiting bipolar transistor. Hence, the circuit node is shorted to the bulk and the contents of the cell are flipped. There is also a probability that parasitic bipolar transistors in neighboring cells sharing the same p-well will turn on, effectively generating an MBU. Depending on the node hit by the particle, the value of the cell may or may not change. For example, in case node Q is struck when the bit is holding 0, the bit cell will not be affected; the same applies when node QB is struck when the bit has a value of 1. Consequently, about 50% of the upsets will not result into bit flips.

Of course, particle effects will not manifest exclusively as 1, 2, 3, or 4 BUs, but rather as a distribution of MBUs of different cardinality. Thus, we define three different fault models that are used in this paper, namely, `Fm90`, `Fm65`, and `FmNew`. The percentile MBU distribution for these three fault models appears in Table III. `Fm90` and `Fm65` are the two realistic distributions of faults taken from [6], while `FmNew` represents an estimation of future vulnerability of memories to MBUs and reflects a uniform distribution of upsets that are up to 6-bits wide.

While Fig. 7 shows a layout of 6T cell, the proposed methodology is *independent of the underlying technology*. Therefore, the presented methodology can be directly applied to 8T or 10T cell designs. Similar to error detection and correction mechanisms that rely on *a priori* knowledge of expected behavior of particle strikes, such as AVF [20] and scrubbing [28], the proposed methodology relies on a representative distribution of MBUs. The key parameter of the distribution is the maximum width of the MBUs, as the ILP solver attempts to place the most critical bits apart. The designer can estimate the maximum radius of MBU strikes based on the density of the SRAM cells and typical radiation experiments [29]. In case of uncertainty, the designer might choose a pessimistic MBU radius.

Finally, as the proposed methodology assumes uniform device-level vulnerability, the designer needs to take precautions to decrease the effect of process parameter variations. These process variations are typically caused by limitations of the fabrication process and variation in the number of dopant atoms in the channel of the short channel devices [30]. An extensive analysis of the impact of SRAM cell process variations appears in [31]. A plethora of solutions, trading-off area, power, and speed, has been proposed. Solutions include body-biasing schemes [32], dual-supply voltage schemes [33], short wordline and bitline pulse schemes [34], and wordline signal rise-time calibration [30].

VI. RESULTS

In this section, we discuss the results of the vulnerability-based parity optimization and interleaving methods. Section VI-A presents the MWVF reduction achieved for various configurations of protected bits, parity trees, and fault models, while Section VI-B discusses the overhead of the presented configurations. Section VI-D compares the quality of the solutions obtained by the ILP solver to the simple algorithm presented in Section II. Finally, Section VI-E demonstrates the effectiveness of VBI and Section VI-F reports the corresponding overhead.

A. MWVF Reduction for Various Configurations

1) Optimal Selection of Parity Bits: Fig. 8(a) and (b) shows the MWVF reduction obtained by adding an increasing number of bits to the parity trees, for the Alpha instruction queue and the P6 RS, respectively, for each of the three fault models. Zero bits indicates that no parity is added. The axis values are omitted for the P6, as vulnerability estimates for the Intel microprocessor cannot be disclosed. However, it is clear from the graphs of both designs that a careful construction of parity trees can lead to a significant vulnerability reduction. For example, adding 77 bits of the Alpha memory array to the parity tree reduces vulnerability by 93%. This, in turn, allows the designer the select the optimal number and distribution of bits to the parity trees to meet reliability goals.

We note that, in case the desired fault coverage for the Alpha in-core memory array is 100%, a configuration of 99 out of the 219 bits, split among two parity trees, offers complete immunity to faults.

2) Effect of Adding Parity Trees: As expected, for the `Fm90` fault model, adding more parity trees has very little effect on the achieved vulnerability reduction. This happens because, in this case, only 5% of the MBUs affect more than 1 bit. However, as a rich set of MBUs is introduced using the `Fm65` and `FmNew` fault models, it is clear that for both microprocessors, one parity tree has limited potential for

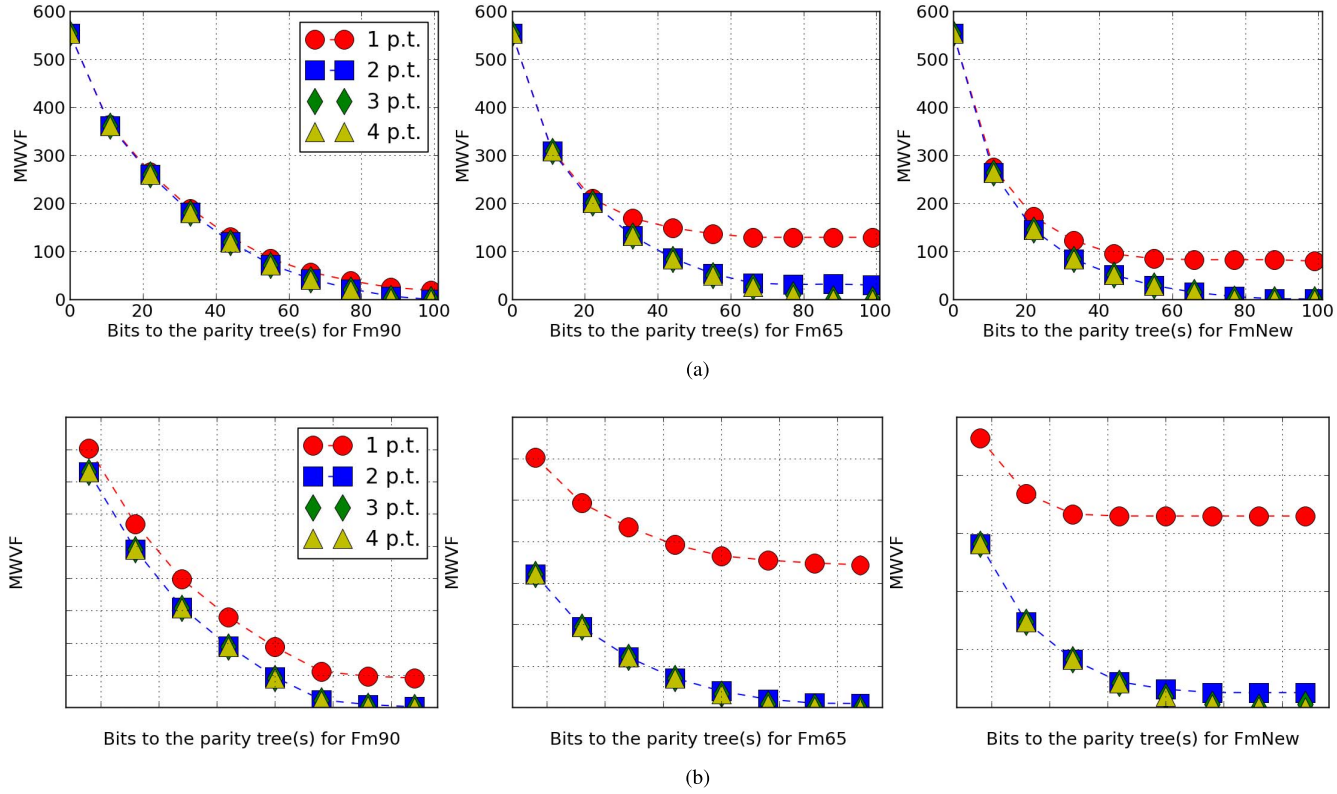


Fig. 8. MWVF reduction for different configuration of parity trees. (a) Alpha 21264 instruction queue. (b) Intel P6 reservation station structure.

TABLE IV
OVERHEAD FOR DIFFERENT PARITY SCHEMES FOR THE ALPHA 21264 INSTRUCTION QUEUE

Percentage of protected bits	Number of protected bits	1 Parity tree			2 Parity trees		
		Logic overhead	Delay overhead	MWVF reduction	Logic overhead	Delay overhead	MWVF reduction
5%	11	0.02%	2.27%	34.29%	0.13%	1.70%	42.41%
10%	22	0.20%	6.70%	52.16%	0.31%	4.38%	53.06%
15%	33	1.85%	8.15%	65.88%	**1.96%**	**5.36%**	**67.32%**
20%	44	3.99%	8.40%	76.53%	**4.10%**	**5.30%**	**78.33%**
25%	55	5.53%	8.85%	84.83%	**5.65%**	**5.60%**	**87.00%**
30%	66	6.79%	9.02%	89.72%	**6.91%**	**5.99%**	**92.23%**
35%	77	8.01%	9.05%	93.14%	8.12%	5.84%	94.40%
40%	88	8.76%	9.08%	95.48%	8.87%	5.96%	98.73%
45%	99	10.66%	9.22%	96.57%	10.76%	6.01%	100%

efficient MBU protection. This difference is more apparent in the P6 in-core memory array, where inclusion of a second parity tree leads to an immediate additional MWVF reduction of 50%, even for a very small numbers of bits added to the parity trees. Furthermore, the overall MVWF reduction achieved using only one parity tree saturates after a certain point. For the Alpha instruction queue, addition of more than 66 bits to the single parity tree does not decrease the vulnerability of the structure; however, 44 bits split into two parity trees offers better protection to MBUs than 66 or more bits in a single parity tree (80 versus 140 MWVF). This key observation highlights the necessity of formulating the parity selection problem as an ILP, as the optimal selection and distribution of bits to parity trees is not straightforward.

Another key observation, concerning the number of parity trees, is that adding more than two parity trees does not offer significant MWVF reduction, even in the presence of 6-bit wide MBUs (`FmNew` model). Since three and four parity trees significantly increase area overhead, this observation allows us to limit the number of parity trees to two. Evidently, this holds true for the selected distribution of faults. As the distribution of MBUs may change dramatically in future nodes, the ILP will be able to handle and identify the need for more than two parity trees.

B. Parity Overhead

Table IV shows the area and delay overheads of protecting the Alpha 21264 instruction queue for a different number of bits added to the parity trees. The instruction queue was synthesized using synopsys design compiler.

1) Area Overhead: Using Table IV, we can select the most desired parity tree configuration to protect against MBUs. As expected, the area overhead increases linearly as more bits

are added to the parity tree, which in turn increases the number of XOR gates required. However, adding a second parity tree adds very small area overhead, as only a flip-flop is added per memory word.

2) Delay Overhead: Similarly, in terms of delay overhead, increasing the depth of the parity tree increases the time required to calculate the word parity. However, adding two parity trees has a considerable advantage, as the depth of the XOR trees decreases and the delay overhead is significantly reduced. Therefore, since in the previous section, we identified two parity trees as sufficient, possible candidates for the most cost-effective resiliency enhancement should be selected among the solutions involving 33, 44, 55, or 66 bits and two parity trees (shown in boldface in Table IV), which offer an MWVF reduction of 67%, 78%, 87%, and 92%, respectively.

3) Recovery Overhead: Error recovery relies on architectural-level microprocessor mechanisms, since parity only offers error detection. Typical server configurations include advanced checkpoint and recovery mechanisms [35], [36]. In case the microprocessor lacks checkpointing mechanisms, popular solutions, such as ReVive [37] and SafetyNet [38], can be seamlessly integrated to the architecture. ReVive adds a 6.3% execution time overhead, while SafetyNet adds no latency to the common case of 99.9% of instructions, by pipelining the checkpoints. The recovery overhead is in the range of a few clock cycles for implementations that checkpoint at the architectural boundary [35], and in range of milliseconds for ReVive and SafetyNet. As in the case of ECCs, the cumulative recovery overhead is dependent on the failing rate of the in-core memories.

C. Comparison With ECC

To highlight the advantages of using selecting parity for vulnerability enhancement, we compare the proposed methodology with traditional ECCs. Typical ECC used in modern microprocessors include Hamming Codes [39] and Hsiao codes (odd-weight-column codes) [15]. Since L2 cache pipelines have high latency, they provide time for deep check-bit generation logic supporting large bundle sizes with low area overhead, e.g., for a 256-bit bundle and a 10-bit syndrome using a Hsiao code, the bit area overhead is 3.9%. For fast microprocessor arrays, however, a lightweight error detection and correction scheme (LEDAC [16]) offers single-cycle detection (assuming a single-cycle read-write), at the price of increased overhead.

Specifically, when LEDAC is applied to the microprocessor array under consideration, it can provide a 100% MWVF reduction at an 88.9% overhead and a two cycle latency (since the in-core memory arrays studied do not offer single-cycle read-write). These results emphasize the advantages of the proposed methodology, offering a 100% MWVF (for two parity trees) at 10.76% overhead.

With regards to recovery latency, ECCs offer various trade-offs between latency and area overhead. Hsiao codes, in particular, are capable of zero-cycle error correction at the expense of added delay to the critical path. This critical path timing penalty, however, may be prohibitive in multigigahertz clock frequencies at which a typical modern microprocessor operates. For example, according to [40], to protect 99 bits, nine check bits are required (as compared with two check bits for parity trees), with a corresponding delay of being 23 equivalent gates. Nevertheless, the designer could identify the most vulnerable bits using the proposed method and use Hsiao codes—instead of the simple parity proposed herein—to immediately correct errors, at the cost of a sizeable increase in area and critical path timing.

D. ILP Solution Evaluation

In this section, we discuss the quality of the solution obtained by the ILP solver, as compared with the simple algorithm described in Section II. Fig. 9(a) and (b) shows the MWVF reduction achieved by the solutions obtained by the solver (ILP-) and the simple algorithm (ALG-), for the `Fm65` and `FmNew` fault models, and for the Alpha and P6 in-core memory arrays, respectively.

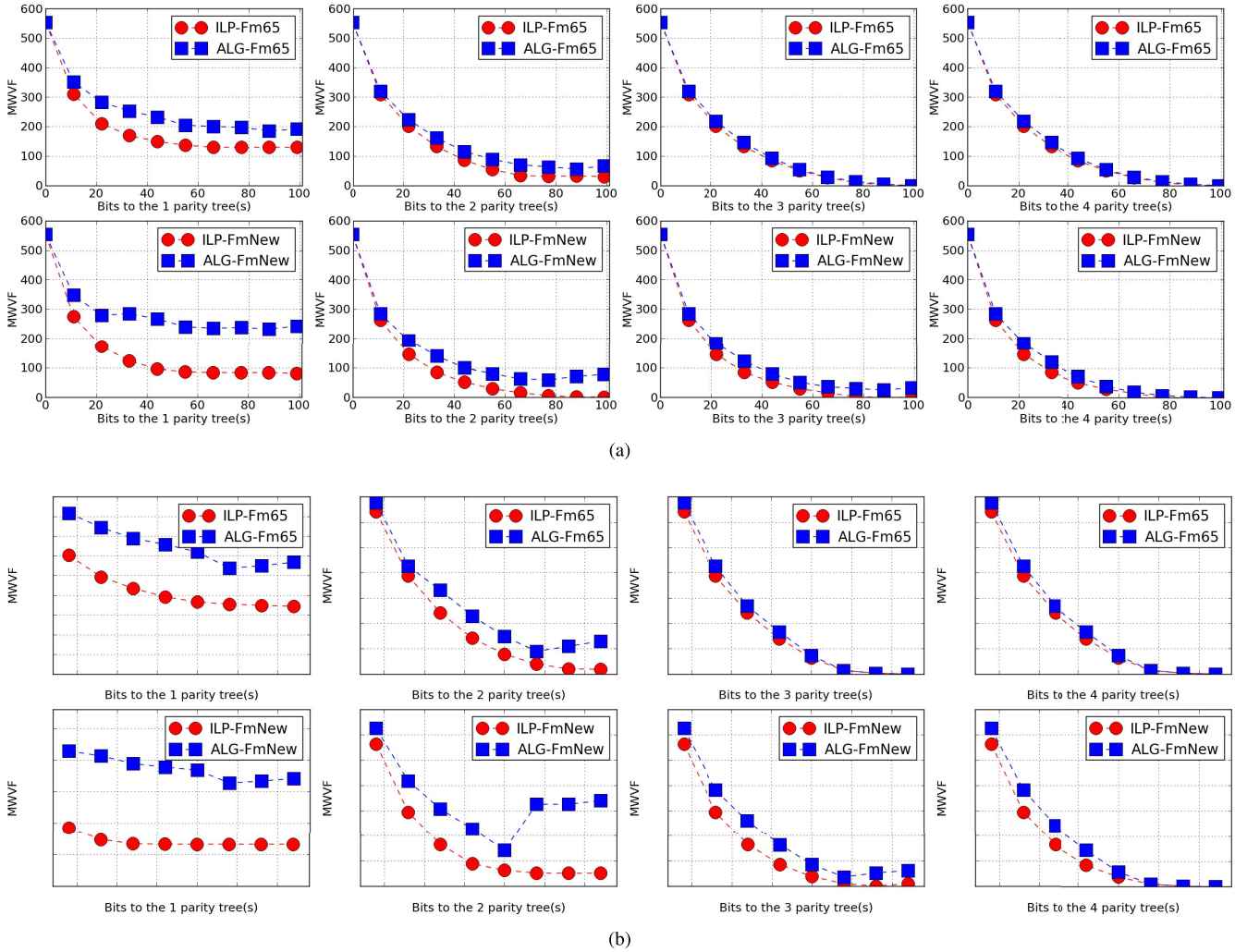
As expected, the solution obtained by the ILP solver is always better than that of the simple algorithm for all configurations of parity trees. Moreover, the simple algorithm yields very poor results when selecting the subset of bits to add to one or two parity trees, for both the Alpha and the P6 structures and for all fault models. For example, the average MWVF difference between the two obtained solutions for the `FmNew` for one parity bit in the Alpha memory array is $\approx$200 units. This is attributed to the effect presented in Section II, where exclusion of a bit from the trees can lead to better protection from MBUs. The simple algorithm produces good results only in the case of three or four parity trees. However, since we demonstrated in the previous section that more than two parity trees add cost without providing much in the way of MWVF reduction, the ILP formulation is necessary to maximize the return on investment with regards to resilience enhancement.

Furthermore, the simple algorithm exhibits an interesting artifact when more than 30% of parity bits are included in one or two parity bits; adding more bits *increases* the vulnerability of the in-core memory arrays. Indeed, adding more parity bits to one tree increases the density of protected bits; thus, blindly adding them to the tree increases the probability of error masking, as more errors resulting in an even number of bit flips are introduced.

E. VBI Application

As mentioned in Section IV, usage of one parity tree, which is the preferred method in parity application, can severely limit the quality of the solution obtained by the ILP formulation. This is evident in Fig. 8(b), where addition of a second parity tree provides an additional 50% reduction for the P6 structure. Therefore, we first apply the VBI algorithm presented in Section IV-A to rearrange the bit fields in the Alpha and the P6 structures and then use the ILP formulation to approximate the optimal bit selection for a single parity tree.

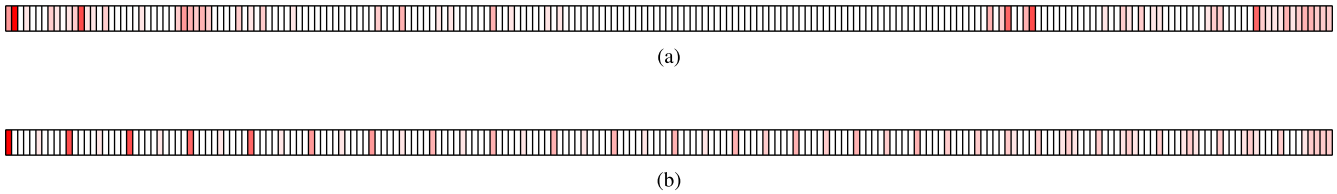
Fig. 10(a) shows the initial order of bitlines, before application of VBI, for the Alpha 21264 instruction queue memory word. Darker color indicates higher vulnerability factors.



(a)

(b)

Fig. 9. MVWF reduction of ILP solution compared with simple heuristic algorithm. (a) Alpha 21264 instruction queue. (b) Intel P6 RS structure.



(a)

(b)

Fig. 10. Pre- and post-VBI orders of bitlines for the Alpha 21264 instruction queue. (a) Initial order of bitlines in the instruction queue. (b) Order of bitlines after VBI.

Fig. 10(b) shows the new order of bit fields, after dispersing the bitlines using the VBI algorithm, assuming a parity protection budget of $B = 10\%$ (i.e., 22 bits). As expected, the most critical bitlines are placed at a maximum distance, based on the given percentage of protected bits.

The vulnerability reduction achieved by VBI deployment appears in Fig. 11. Fig. 11(a) shows the results of applying VBI to the Alpha queue for one parity tree and the three fault models, while Fig. 11(b) shows the corresponding results for the P6 structure. In the case of the latter, it is clear that VBI greatly improves the reliability of the solution, for any given fault model. The relative MWVF reduction is more than 85% for the Fm65 and the FmNew fault models.

In the case of Alpha structure, however, VBI improves the reliability of the structure when more than 22 bits (10%) are added to the parity tree. This result is attributed to the fact that the initial clustering of critical bit fields can be more effective in MWVF reduction, as only one parity bit can significantly reduce vulnerability, since a very few parity bits are added to the tree. VBI spreads the bit fields along the memory word, so it is harder to protect many of them using a very few bits. However, as more than 10% bit fields are added to the parity tree, VBI-based solutions outperform solutions achieved by non-VBI placement. Since the expected use of selective parity involves adding more than 10% for an effective solution, VBI, coming at virtually no extra cost, can significantly improve the solution obtained by the ILP solver.

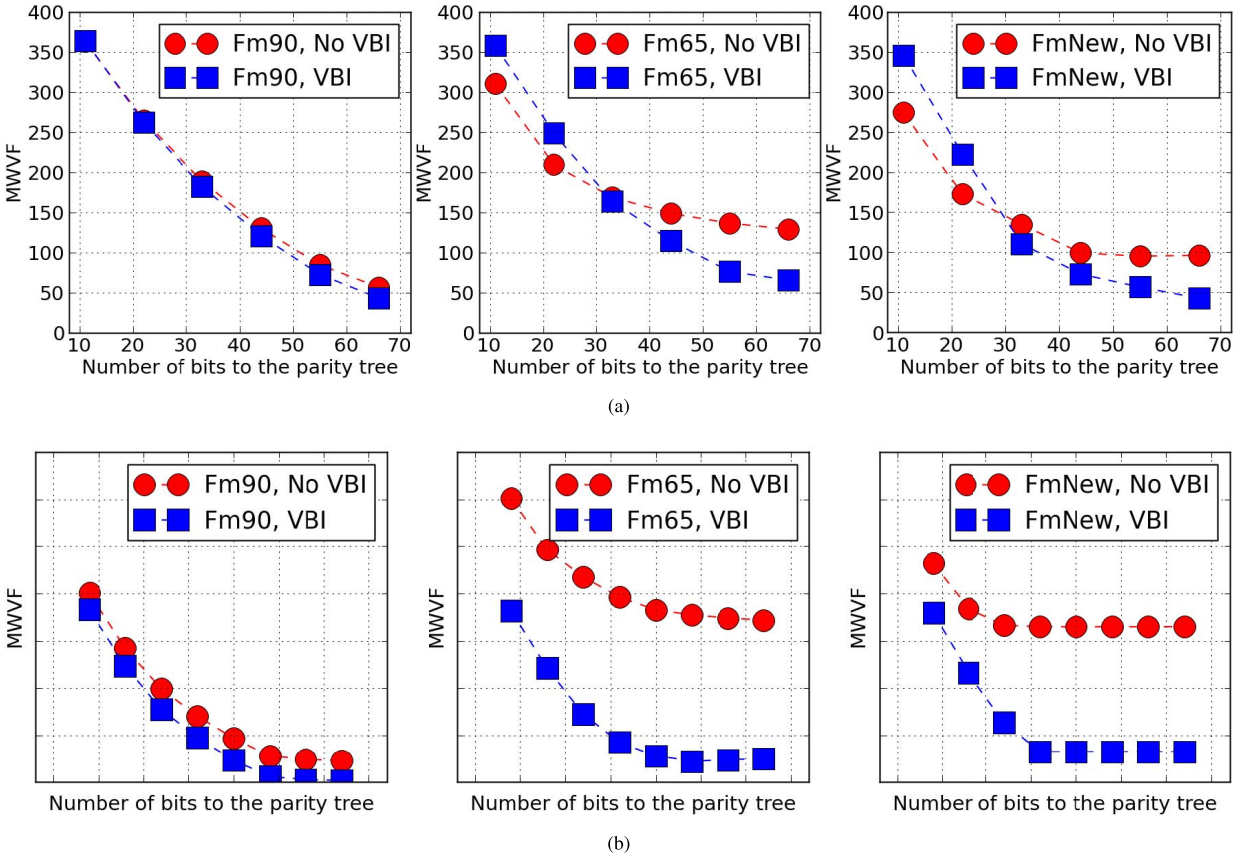


Fig. 11. MVWF reduction by applying VBI. (a) Alpha 21264 instruction queue. (b) Intel P6 RS structure.

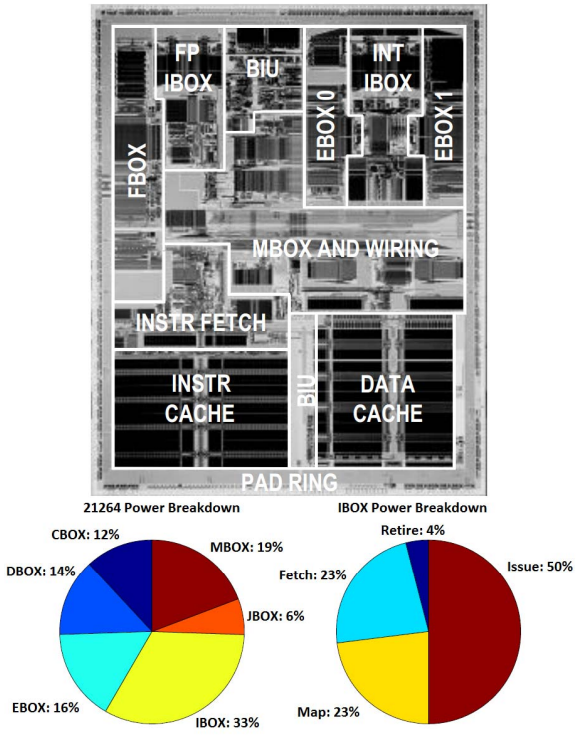


Fig. 12. Alpha IBOX [41].

F. VBI Overhead

Finally, we estimate the overhead of applying VBI to the instruction scheduler of the Alpha 21264 microprocessor. As described in Section V-C, the scheduler features a 32-slot, 219-bit wide instruction queue for storing the incoming instructions (up to 4) from the renaming logic, and can dispatch up to six instructions to the corresponding functional units. We synthesized the Alpha 21264 scheduler using synopsys design compiler and we generated a floorplan and layout using synopsys integrated circuits compiler. The target library was the synopsys generic library, which includes nine metal layers. The floorplanning for the instruction scheduler was initialized to match the actual layout used for the commercial Alpha 21264, as shown in Fig. 12 (INT IBOX). The instruction queue SRAM array is placed on the top of the module, and the scoreboard on the bottom.

We then repeated the process, this time using the new bitline arrangement for the instruction queue, as dictated by the VBI algorithm, while keeping the floorplan and the I/O port location the same. The modified order of incoming bits resulted in an increase in total wire length within the instruction scheduler, as well as extra buffers which the synthesis and layout tools added to meet the timing constraints set forth (in our experiments, the clock frequency was set to 1 GHz). However, the results show that no area overhead was incurred by applying VBI. This can be explained by observing that the instruction scheduler has a large number of I/O ports, due to which the utilization of the control logic area is rather low. This allows for efficient routing and buffer addition even in the case of a completely random rewiring of the incoming bits. The added wires and buffers, however, do cause an increase in

the power consumed by the design. Specifically, after applying the VBI algorithm, the total dynamic power of the control logic increased by 0.09%. Given that the IBOX consumes 33% of the total power of the microprocessor (Fig. 12), this overhead is negligible. Overall, the area and power overhead incurred by the VBI algorithm is minimal and well justified, given the achieved vulnerability reduction.

VII. Conclusion

Recent radiation-induced experiments in contemporary technology nodes reveal a significant increase in MBUs, highlighting the need for revisiting vulnerability analysis and developing novel methods for protecting modern microprocessor in-core memory arrays against MBUs. To this end, we proposed technology-independent AVF-driven selective parity as an efficient method for detecting SBU and MBU, and we introduced an ILP formulation of the parity forest construction optimization problem. Experimentation with several multibit fault distributions injected into in-core memory arrays of the Alpha 21264 and the Intel P6 instruction schedulers elucidated that optimal single tree parity selection can achieve great vulnerability reduction, even when only a small number of bits are added to the parity trees. Furthermore, effective exploration of the solution space, as enabled by the ILP formulation, revealed that the introduction of a second parity tree offers a vulnerability reduction of more than 50% over a single parity tree. Finally, in constrained problem instances where the ILP solver has limited flexibility in selecting the optimal parity tree, application of the proposed VBI method can lead to vulnerability reduction of 86% with minimal overhead. Looking ahead, we expect that the benefits of employing vulnerability-based parity optimization will increase further, as smaller process nodes will exhibit greater vulnerability to MBUs.

References

[1] N. Seifert *et al.*, "Radiation-induced soft error rates of advanced CMOS bulk devices," in *Proc. IEEE Int. Rel. Phys. Symp.*, Mar. 2006, pp. 217–225.

[2] R. A. Reed *et al.*, "Heavy ion and proton-induced single event multiple upset," *IEEE Trans. Nucl. Sci.*, vol. 44, no. 6, pp. 2224–2229, Dec. 1997.

[3] R. Koga, S. D. Pinkerton, T. J. Lie, and K. B. Crawford, "Single-word multiple-bit upsets in static random access devices," *IEEE Trans. Nucl. Sci.*, vol. 40, no. 6, pp. 1941–1946, Dec. 1993.

[4] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *Proc. IEEE Int. Rel. Phys. Symp. (IRPS)*, Apr. 2011, pp. 5B.4.1–5B.4.7.

[5] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, "Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule," *IEEE Trans. Electron Devices*, vol. 57, no. 7, pp. 1527–1538, Jul. 2010.

[6] G. Georgakos, P. Huber, M. Ostermayr, E. Amirante, and F. Ruckerbauer, "Investigation of increased multi-bit failure rate due to neutron induced SEU in advanced embedded SRAMs," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2007, pp. 80–81.

[7] J. Abella, R. Canal, and A. Gonzalez, "Power- and complexity-aware issue queue designs," *IEEE Micro*, vol. 23, no. 5, pp. 50–58, Sep./Oct. 2003.

[8] S. Chaudhry *et al.*, "Rock: A high-performance SPARC CMT processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, Mar./Apr. 2009.

[9] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006.

[10] A. Buyuktosunoglu, D. H. Albonesi, P. Bose, P. W. Cook, and S. E. Schuster, "Tradeoffs in power-efficient issue queue design," in *Proc. ACM Int. Symp. Low Power Electron. Design*, 2002, pp. 184–189.

[11] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 397–404, Sep. 2005.

[12] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2007, pp. 197–209.

[13] M. Maniatakos, M. K. Michael, and Y. Makris, "AVF-driven parity optimization for MBU protection of in-core memory arrays," in *Proc. Design, Autom. Test Eur. Conf. Exhibit.*, Mar. 2013, pp. 1480–1485.

[14] R. Naseer and J. Draper, "Parallel double error correcting code design to mitigate multi-bit upsets in SRAMs," in *Proc. IEEE 34th Eur. Solid-State Circuits Conf.*, Sep. 2008, pp. 222–225.

[15] M. Y. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM J. Res. Develop.*, vol. 14, no. 4, pp. 395–401, Jul. 1970.

[16] K. C. Mohr and L. T. Clark, "Delay and area efficient first-level cache soft error detection and correction," in *Proc. Int. Conf. Comput. Design (ICCD)*, Oct. 2007, pp. 88–92.

[17] S. Almukhaizim, P. Drineas, and Y. Makris, "Entropy-driven parity-tree selection for low-overhead concurrent error detection in finite state machines," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 8, pp. 1547–1554, Aug. 2006.

[18] N. A. Touba and E. J. McCluskey, "Logic synthesis of multilevel circuits with concurrent error detection," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 16, no. 7, pp. 783–789, Jul. 1997.

[19] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar./Apr. 1999.

[20] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2003, pp. 29–40.

[21] S.-K. Lu, S.-Y. Kuo, and C.-W. Wu, "Fault-tolerant interleaved memory systems with two-level redundancy," *IEEE Trans. Comput.*, vol. 46, no. 9, pp. 1028–1034, Sep. 1997.

[22] A. Makhorin, "Modeling language GNU MathProg," Ph.D. dissertation, Dept. Appl. Inform., Moscow Aviation Inst., Moscow, Russia, 2000.

[23] T. Achterberg, "SCIP: Solving constraint integer programs," *Math. Program. Comput.*, vol. 1, no. 1, pp. 1–41, 2009.

[24] M. Maniatakos, N. Karimi, Y. Makris, A. Jas, and C. Tirumurti, "Design and evaluation of a timestamp-based concurrent error detection method (CED) in a modern microprocessor controller," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, Oct. 2008, pp. 454–462.

[25] M. Maniatakos, C. Tirumurti, A. Jas, and Y. Makris, "AVF analysis acceleration via hierarchical fault pruning," in *Proc. 16th IEEE Eur. Test Symp.*, May 2011, pp. 87–92.

[26] K. Osada, K. Yamaguchi, Y. Saitoh, and T. Kawahara, "SRAM immunity to cosmic-ray-induced multierrors based on analysis of an induced parasitic bipolar effect," *IEEE J. Solid-State Circuits*, vol. 39, no. 5, pp. 827–833, May 2004.

[27] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and AVF estimation revisited," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun./Jul. 2010, pp. 477–486.

[28] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt, "Cache scrubbing in microprocessors: Myth or necessity?" in *Proc. 10th IEEE Pacific Rim Int. Symp. Dependable Comput.*, Mar. 2004, pp. 37–42.

[29] P. N. Sanda *et al.*, "Soft-error resilience of the IBM POWER6 processor input/output subsystem," *IBM J. Res. Develop.*, vol. 52, no. 3, pp. 285–292, May 2008.

[30] S. K. Varanasi and S. Mandavilli, "Process variation tolerant SRAM cell design," in *Proc. Int. Symp. Electron. Syst. Design (ISED)*, Dec. 2011, pp. 82–87.

[31] A. J. Bhavnagarwala, X. Tang, and J. D. Meindl, "The impact of intrinsic device fluctuations on CMOS SRAM cell stability," *IEEE J. Solid-State Circuits*, vol. 36, no. 4, pp. 658–665, Apr. 2001.

[32] Y. Takeyama, H. Otake, O. Hirabayashi, K. Kushida, and N. Otsuka, "A low leakage SRAM macro with replica cell biasing scheme," *IEEE J. Solid-State Circuits*, vol. 41, no. 4, pp. 815–822, Apr. 2006.

[33] K. Zhang *et al.*, "A 3-GHz 70-mb SRAM in 65-nm CMOS technology with integrated column-based dynamic power supply," *IEEE J. Solid-State Circuits*, vol. 41, no. 1, pp. 146–151, Jan. 2006.

[34] Y. Ye *et al.*, "Wordline & bitline pulsing schemes for improving SRAM cell stability in low-Vcc 65 nm CMOS designs," in *Symp. VLSI Circuits Dig. Tech. Papers*, 2006, pp. 9–10.

[35] L. Spainhower and T. A. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective," *IBM J. Res. Develop.*, vol. 43, nos. 5–6, pp. 863–873, Sep. 1999.
[36] Intel Corporation. (2011). *Intel Xeon Processor E7 Family: Reliability, Availability, and Serviceability*. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-e7-family-ras-server-paper.pdf
[37] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proc. 29th Annu. Int. Symp. Comput. Archit.*, May 2002, pp. 111–122.
[38] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proc. 29th Annu. Int. Symp. Comput. Archit.*, May 2002, pp. 123–134.
[39] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, 1950.
[40] D. Rossi, N. Timoncini, M. Spica, and C. Metra, "Error correcting code analysis for cache memory high reliability and performance," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, Mar. 2011, pp. 1–6.
[41] M. Matson *et al.*, "Circuit implementation of a 600 MHz superscalar RISC microprocessor," in *Proc. IEEE Int. Conf. Comput. Design*, Oct. 1998, pp. 104–110.

Michail Maniatakos (S'08–M'12) received the B.Sc. and M.Sc. degrees in computer science and embedded systems from the University of Piraeus, Piraeus, Greece, in 2006 and 2007, respectively, and the M.Sc., M.Phil., and Ph.D. degrees from Yale University, New Haven, CT, USA, in 2008, 2009, and 2012 respectively.

He is currently an Assistant Professor of Electrical and Computer Engineering with New York University (NYU) Abu Dhabi, Abu Dhabi, United Arab Emirates, and a Research Assistant Professor with the NYU Polytechnic School of Engineering, New York, NY, USA. He is also the Director of the Modern Microprocessor Architectures Laboratory at NYU Abu Dhabi. He has authored multiple publications in the IEEE TRANSACTIONS and conference papers. His current research interests include hardware security, computer architecture, and industrial control systems security.

Prof. Maniatakos has served as a Program Committee Member for several IEEE conferences.

Maria K. Michael (S'01–M'03) received the B.S. and M.S. degrees in computer science and the Ph.D. degree in electrical and computer engineering from Southern Illinois University, Carbondale, IL, USA, in 1996, 1998, and 2002, respectively.

She was a Lecturer with the Department of Electrical and Computer Engineering at Southern Illinois University from 2001 to 2002, and an Assistant Professor of Computer Science and Engineering with the University of Notre Dame, Notre Dame, IN, USA, from 2002 to 2003. She is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus. Her current research interests include test and reliability of modern digital VLSI circuits, embedded systems and multicore architectures, including on-line/adaptive testing for multicore designs, test and diagnosis for various faults (including timing and other deep-submicrometer/nanometer-induced faults), single-bit and multibit upset analysis and protection, symbolic techniques for test and verification (BDDs and SAT), and parallel methods/algorithms for EDA tools.

Dr. Michael has served on the Technical Program Committees of several international conferences and workshops, and is a reviewer for a number of scholarly journals and international conferences. She was a co-recipient of the Best Paper Award from the 2009 International Conference on Microelectronic Systems Education.

Yiorgos Makris (S'96–M'02–SM'08) received the Diploma degree in computer engineering and informatics from the University of Patras, Patras, Greece, in 1995, and the M.S. and Ph.D. degrees in computer science and engineering from the University of California at San Diego, La Jolla, CA, USA, in 1998 and 2001, respectively.

He joined the University of Texas at Dallas, Richardson, TX, USA, where he is currently a Professor of Electrical Engineering, leading the Trusted and Reliable Architectures Research Laboratory, after spending over a decade with the Faculty of Electrical Engineering, Yale University, New Haven, CT, USA. His current research interests include the applications of machine learning and statistical analysis in the development of trusted and reliable integrated circuits and systems, with a particular emphasis on the analog/RF domain.

Prof. Makris served as the Program Chair of the IEEE VLSI Test Symposium, from 2013 to 2014, and the Test Technology Educational Program from 2010 to 2012. He also served as a Guest Editor of the IEEE TRANSACTIONS ON COMPUTERS and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. He serves as a Topic Coordinator and/or Program Committee Member for several IEEE and Association for Computing Machinery conferences. He was a recipient of the 2006 Sheffield Distinguished Teaching Award and the Best Paper Award from the 2013 Design Automation and Test in Europe Conference.