Online and Operand-Aware Detection of Failures Utilizing False Alarm Vectors

Amir Yazdanbakhsh, David Palframan, Azadeh Davoodi, Nam Sung Kim and Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, USA
Email: {yazdanbakhsh, palframan, adavoodi, nskim3, mikko}@wisc.edu

***Abstract*—This work presents a framework which detects *online* and at *operand level* of granularity *all* the vectors which excite already-diagnosed failures in combinational modules. These vectors may be due to various types of failure which may even change over time. Our framework is flexible with the ability to update vectors in the future. Moreover, the ability to detect failures at operand level of granularity can be useful to improve yield, for example by not discarding those chips containing failing and redundant computational units (e.g., two failing ALUs) as long as they are not failing at the same time. The main challenge in realization of such a framework is the ability for on-chip storage of *all* the (test) cubes which excite the set of diagnosed failures, e.g., all vectors that excite one or more slow paths or defective gates. The number of such test cubes can be enormous after applying various minimization techniques, thereby making it impossible for on-chip storage and online detection. A major contribution of this work is to significantly *minimize* the number of stored test cubes by inserting only a few but carefully-selected "false alarm" vectors. As a result, a computational unit may be *mis*-diagnosed as failing for a given operand however we show such cases are rare while the chip can safely be continued to be used, i.e., our approach ensures that none of the *true-positive* failures are missed.**

I. Introduction

Technology scaling beyond 32nm significantly degrades the manufacturing yield. This degradation is exasperated for emerging technologies such as 3D integration [7]. One way to improve yield is by creating a layout with manufacturing-friendly patterns, for example imposing restrictive design rules or just using regular fabrics [9], possibly in combination with via-configurable logic blocks to make post-silicon corrections [18]. A particular challenge in this approach is to avoid significant degradation in power, area, or performance compared to non-regular and flexible design [13], [19].

Redundancy at various levels can be used as an alternative to improve yield. Many prior proposals suggest exploiting redundancy that *already* exists in high performance processors to facilitate defect tolerance [20], [21], [23]. For instance, faulty execution units or faulty entries in queue-based structures can be disabled. Disabling defective units in this manner results in a functional processor with decreased performance. To avoid the performance penalty due to disabling logic, additional redundancy can be introduced in the form of spare execution units. In case an execution unit is faulty, a spare can be used in its place. This type of redundancy is somewhat coarse-grained, however, and may not be area efficient. For this reason, other proposals suggest the use of fine-grained redundancy. Some of these approaches incorporate redundancy at the granularity of a bit slice [5], [14], [12]. Such approaches are potentially more area-efficient than more coarse-grained redundancy, but they can impose a significant performance penalty, since many multiplexors are required to route around a potentially bad slice of logic. Still other techniques suggest exploiting already existing circuit redundancy. For instance, in [15], a small amount of extra logic is added to take advantage of redundancy in the carry logic of Kogge-Stone adders. Though area efficient, the extra logic is on the critical path and may reduce performance.

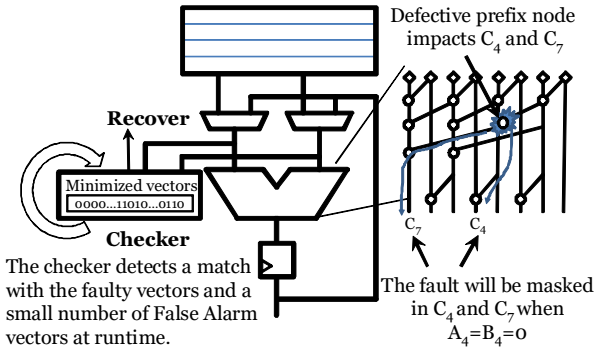

Fig. 1. Our framework for a Han-Carlson adder containing a defective node

A chip can be continued to be used if the replica of its defective module is free of faults. However, the degree of redundancy is limited because of the induced area overhead, and the increasing variety of modules in modern processors (e.g., arithmetic logic, multiplier, floating point, bypass logic, etc.) each of which requires dedicated redundant modules. In fact this may be the reason that so far modern processors have only contained redundant ALUs and not other computational modules. At the same time, the increasing degradation in manufacturing yield requires a higher degree of redundancy. Otherwise, for a given module, its corresponding redundant modules may all be found to contain faults. These factors decrease the effectiveness of a scheme solely based on redundancy to improve the yield in challenging technology nodes.

In this work we introduce a *flexible* framework which detects *online* and at *operand level* of granularity *all* the vectors which may excite already-diagnosed failures in failing combinational modules. Our framework is flexible in the sense that all it needs are the vectors (or cubes) which make a computational unit to fail. Otherwise, it does not care about the source of the failure. For example the failure could be due to a single or multiple stuck-at-faults, or due to timing paths failing because of process variations or aging. Indeed, as new vectors are identified over time, they can be incorporated with the previous ones and detected by the checker unit. Moreover, unlike some previous works, our implementation does not result in adding extra logic on the critical design paths.

Our framework is based on an on-chip "checker" unit implemented as a TCAM which detects online if a vector feeding a defective combinational module causes an observable failure. Figure 1 shows an example when the failing module is a Han-Carlson adder [8]. A defect in an internal node only impacts bit positions C_4 and C_7 and will not yield to an observable failure when the input arguments $A_4 = B_4 = 0$. Therefore the adder may continue to be used when there is no match with a failing vector in the checker.

Upon online detection of the failure, the checker sets a recovery flag which in a microprocessor signals re-issue of the operation. In

the case of a defect, we propose this re-issue to be by scheduling the operation on a redundant computational unit as long as the redundant unit is not failing for the same vector (but may be failing for other vectors). If it is a failing timing path, the recovery could be re-execution on the same unit after scaling down to a lower frequency.

The main challenge in realization of such a framework is the ability for on-chip storage of *all* the (test) cubes which excite an identified failure. The number of cubes that excite a failure (e.g., capturing all vectors that excite a failing path or even a gate) can be numerous even after applying various minimization techniques, thereby making it impossible for on-chip storage and online detection. A major contribution of this work is to significantly minimize the number of stored test cubes by inserting only a few "false alarm" vectors. As a result, a computational unit may be misdiagnosed as failing for a given operand however we show by carefully selecting the false alarms, the number of misdiagnosed cases can be minimized, while the chip can safely be continued to be used and all true failures are guaranteed to be detected.

Our procedure to insert false alarm vectors extends the ESPRESSO tool for two-level logic minimization. It suits a TCAM-based implementation of the checker unit for online checking against test cubes which we also argue is also an area-efficient alternative.

The contributions of this work can be summarized below.

- Proposing the checker unit as a flexible option for online and operand-level fault detection, with the ability to update faulty vectors from various sources over time.
- Design of the unit using a TCAM-based implementation.
- Proposing the use of false alarm vectors which we show significantly minimizes the number of vectors to be checked, thus reducing the TCAM area, with only a slight increase in the number of misdiagnosis, as verified by realistic workloads on microprocessors of different issue widths.

In the remaining sections, we first discuss our framework and various designs of the checker unit in Section II. We then present our false alarm insertion procedure in Section III. Simulation results are presented in Section IV followed by conclusions in Section V.

II. Design of the Checker Unit

Modern microprocessors provide multiple ALUs for wide issue. We explain our checker unit using an example for a 2-issue processor as shown in Figure 2 (a). The figure shows implementation of the checker unit as a TCAM. (The circuitry added for protection is shown in red.) For a set of test cubes stored in the TCAM, the operands of the two ALUs are checked online. In case there is a match, a recovery flag is activated indicating the ALU which is going to fail to process that operand. The recovery activates the process which ensures the incorrectly-processed instruction will not be committed and instead it will be re-issued in the pipeline to the other ALU (assuming the other ALU does not fail for the same operand). The checker can work effectively if the recovery flag is not activated too frequently. Regardless, it allows continuing the use of a failing system, even when potentially two ALUs are failing as long as they don't fail for the same operand. It is also configurable and the set of stored test cubes can be updated if more failures are found over time (as we discuss later in this section).

A. Area Overhead

Implementing the checker in the above example requires MUXes to send the operands of both ALUs to the checker. For the 2-wide processor in this example, we need 2 2-1 MUXes (each for example 32 bits). For a 4-wide processor, we need 2 4-to-1 (32-bit) MUXes.

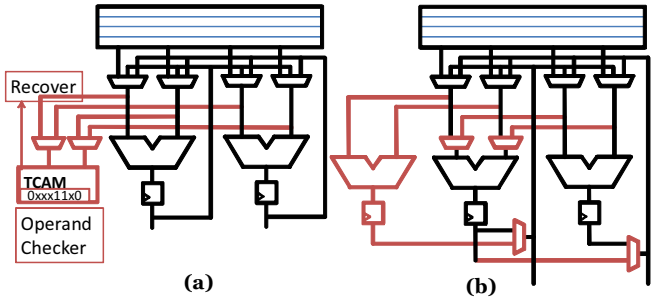

Fig. 2. Alternatives to protect a 2-issue faulty processor using (a) a checker unit; (b) adding a redundant module

To understand this area overhead, we compare against a redundancy-based scheme which can be considered as a more conventional way for protection against failures.

In Figure 2(b), we show how a redundant ALU can be added to the 2-issue processor. To support a redundancy technique for ALUs, we need MUXes that can forward (i) the input vector from any potentially-faulty ALU to the third ALU and (ii) the output vector from the redundant ALU to the faulty ALU. The additional MUXes ensure that in case *any* of the three ALUs is failing, the other two can be used. This MUX network can lead to a significant area overhead. For example for the 2-issue processor in Figure 2(b) we show how the additional (third) ALU is connected. There are 4 2-to-1 (32-bit) MUXes in this case.

The area overhead of (b) and (a) are shown in red. The use of (a) is desirable if it has a lower area than (b). (This is disregarding other desirable features of (a) including online operand-aware detection of failures and providing a mechanism to work with two failing ALUs as long as they are not failing for the same operand.) Moreover, unlike the redundant ALU case, the MUXes in (a) *are not on the critical path*. Thus, these MUXes can be sized much smaller than the MUXes used for the redundancy technique in Figure 2(b). Moreover, by use of techniques such as buffer insertion, the loading impact of the MUXes can be minimized.

Two factors can be considered in allotting the area of the checker at the design stage. First, the checker's area can be decided based on the number and types of modules which need protection, and by comparing with the overhead associated with using other alternatives (e.g., redundancy-based). (In our experiments we show the use of checker is a more area efficient alternative than use of redundant computational modules for microprocessors of various issue-widths.) The second factor limiting the checker's area is the recovery overhead (e.g., total runtime spent on re-execution of instructions).

B. Implementation Options

Here we compare three options: (i) a Ternary Content Addressable Memory (TCAM), (ii) Programmable Logic Array (PLA), and (iii) a Field-Programmable Gate Array (FPGA). We briefly discuss each option in the context of our checker unit.

1) TCAM: Since all the test cubes after the minimization include many don't care bits, our first and most logical implementation choice is using a TCAM that can match a content line with don't-care bits; note that a traditional CAM cannot be used for the checker unit. A conventional TCAM needs to support a random access to a specific entry to update the key value at runtime. This requires a log2(N)-to-N decoder for a TCAM with N entries. Thus, the decoder takes a notable fraction of the area in a conventional TCAM. However, we do not need such a decoder in our use of TCAM, because each entry must be updated only once, every time the chip is turned on.

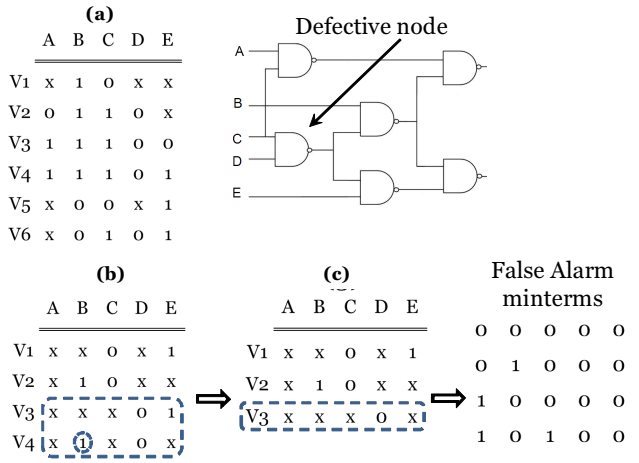

Fig. 3. Example of failing C17 circuit; (a) faulty cubes before minimization; (b) faulty cubes after minimization; (c) cubes after adding false alarms

Therefore, supporting a sequential access to write the test cubes to the TCAM is sufficient, which we implement with a simple counter.

2) PLA: Our second choice is using NOR-NOR-based PLA that can realize an arbitrary combinational logic. In fact, a set of test cubes for a given defect is equivalent to a truth table. Hence, applying it to a (PLA) logic synthesis tool gives us a two-level Boolean function, which can be easily mapped into a PLA. Considering that the one-time programmable fuse cannot be easily integrated with logic devices, we consider the PLA that uses SRAM cells to program each minterm in each test cube. In our case, the PLA using SRAM cells becomes equivalent to TCAM array.

3) FPGA: Our third choice is using an embedded FPGA fabric, which can be implemented with a typical logic device technology. Like a PLA, an FPGA can implement an arbitrary Boolean function. Since our minimized test cubes contain many don't-cares within each cube, synthesizing it to a Boolean function may lead to a very compact circuit that needs only a fraction of FPGA resource.

C. Overhead of Programming the Checker at Startup

In our scheme, after minimizing the test cubes, we must store them on-chip and ensure they are loaded in the checker unit every time the chip is turned on. The first choice is to store the test cube in the on-chip one-time-programming (OTP) memory (a.k.a. fuses). However, the size of each fuse cell is larger than the SRAM cell used for the checker unit which in practice can double the area overhead of the checker unit, when fuses are accounted for. Recently, mounting a flash memory package on a processor package using package-on-package technology began to receive a notable attention [6]. This can provide a large amount of non-volatile storage at a cost far cheaper than on-chip OTP memory integrated with logic device technology, allowing manufacturers to enable new applications and to store a large amount of chip-specific circuit tuning information [6]. Hence, we assume that the programming values of a detector is stored in the flash memory at no extra cost.

In Section IV, we present a case study containing detailed area analysis for TCAM and FPGA-based implementations of the checker unit and compare them with the area overhead of integrating a redundant ALU for processors of different issue widths. (We do not consider PLA-based implementation because it becomes equivalent to TCAM in our scheme.)

Overall, TCAM is the most suitable implementation for matching against test cubes and as we show, the most area efficient alternative.

III. Utilizing False Alarms for Test Cube Minimization

A. Overview

Given a failing path or slow/defective gate, the on-chip checker needs to store all the test cubes exciting the failure. Storing the total number of test cubes is not possible in practice. So in the first step, we aim to minimize the number of test cubes as much as possible. Here we propose to use a 2-level logic minimization tool. This is because it is most suitable for a TCAM-based implementation of the checker which allows storing/parallel-checking against the individual test cubes, containing don't-care bits.

However, even after minimization, the number of test cubes can be prohibitively large. Therefore, in this section we propose the idea of false alarm insertion which allows significant reduction/minimization in the number of stored test cubes. Consider an "alarm" function representing a Boolean recovery signal generated by the checker. The on-set of this function is represented by the minimized test cubes. *By adding false alarms we essentially map a few terms from the off-set back into the on-set in order to further minimize the number of test cubes (corresponding to the TCAM entries) as many as possible.*

Specifically, using false alarm insertion, we aim to reduce the number of test cubes beneath a target threshold corresponding to the maximum number of entries in the TCAM. For example, the threshold is set to *only* 32 test cubes in our simulation setup.

Example: Figure 3 shows these steps for the C17 benchmark circuit from the ISCAS85 suite containing one defective gate. The number of all the test cubes which yield to an observable fault in the outputs are 6 and 4, corresponding to before and after minimization, respectively. Each test cube is shown on one row and may contain don't-cares indicated by x. In Figure 3(b) v4 is expanded in literal 'B' and as a result it can be merged with v3 (or v3 can be eliminated). Expanding v4 introduces 8 additional minterms. However four of these minterms are already included in v3. So only the remaining four minterms will be false alarms which are listed in the figure. Compared to Figure 3(a), the number of test cubes is dropped by half with the insertion of four false alarms. In the remainder of this section, we discuss the details of our procedure for false alarm insertion.

B. Procedure

Figure 4(a) shows the overview of our procedure. The input is a set of already minimized faulty test cubes. Our goal is to reduce the size of this set beneath a given threshold while minimizing the total number of added false alarms.

Our procedure is comprised of the following two core steps which are tightly integrated and repeated at each iteration. The first step expands a subset of the test cubes by identifying for each cube, a single literal which should be dropped in it. Since the input of the algorithm is an already minimized set, further expansion of a cube may likely make it overlap with the off-set of the function, introducing false alarms at this step. However, the expansion in turn may also help reduce the size of the resulting test cubes, as we showed in the example in Figure 3. Therefore, in the second step, we apply logic minimization to reduce the size of the cubes. Specifically, in our implementation, we integrated step one within the main loop of the ESPRESSO logic minimization algorithm [3] by modifying the corresponding source code from [1].

The details are shown in Figure 4(a). To insert false alarms at step one, we add the expandFA function. Step two is then an ordered sequence of standard logic minimization techniques, namely irredundant (for removing a cube fully contained in another cube), reduce (for breaking a cube into more number of cubes),

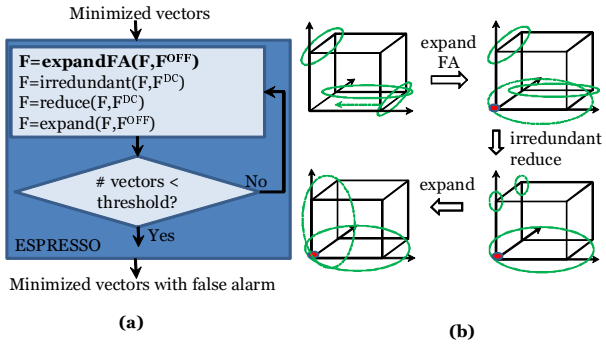

Fig. 4. Integrated false alarm insertion and logic minimization

and `expand` (for expanding a cube in the on-set of the function). After step 2, if the number of test cubes falls beneath the threshold, false alarm insertion is stopped, otherwise the process is repeated.

Note, in the next iteration, a cube which is considered in `expandFA` may already have false alarms so its further expansion by dropping a literal adds more false alarms.

Figure 4(b) shows an example when applying the above sequence for a function of 3 variables. The on-set of the function are the points in the cube which are included in the circles. Each circle represents a test cube. First, the `expandFA` procedure expands a cube and as a result one unit of false alarm is added. The on-set of the function is updated to include the false alarm. The expanded cube now fully contains one of the test cubes which gets identified and removed by the `irredundant` procedure. As a result, the number of test cubes is reduced by one. Next, the `reduce` procedure breaks one test cube into two new cubes which allows its expansion in the on-set of the function (without adding false alarm). The end result is reducing the number of cubes as well as the number of literals.

C. False Alarm Insertion for One Cube

At each call of the `expandFA` function, each cube is visited once and considered for expansion to include some of the points in the off-set. Specifically, at iteration i of the algorithm, only the cubes that have up to i number of don't-cares are expanded. This ordering ensures that the cubes with fewer don't-cares are expanded first. Such a cube introduces fewer false alarms if one of its literals is dropped.

Furthermore, if a cube satisfies the expansion requirement, its don't-care count will be increased by one after dropping one literal. Therefore, in the future iterations, it will continue to be expanded, each time by one literal, until the number of test cubes falls beneath the threshold.

For a given cube, the choice of the literal to be dropped is made such that first, overlap with the off-set is minimized (to ensure inserting fewest number of false alarms), and as a secondary objective, the overlap with on-set is maximized (to increase the opportunities for logic minimization in the subsequent step). We explain our procedure for dropping a literal for a single cube to achieve these objectives using the example in Figure 5. Consider the cube ON1 with value 000x in a four-variable function. The off-set is stored in a minimized way using three cubes as shown in the K-map and represented by a matrix. Each row represents one of the cubes in the off-set and a column gives the literal index. Next, a false alarm matrix is formed where each row corresponds to a row in the off-set matrix. The number of columns is equal to the number of literals. An entry in row i and column j designates the number of introduced false alarms between the off-set cube i and (the expanded) cube when literal j is dropped.

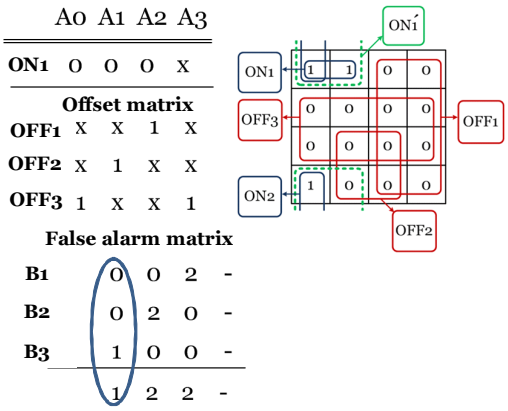

Fig. 5. False alarm insertion for one cube

For example, in Figure 5, dropping literal A0 in ON1, results in its expansion in a new cube denoted by ON1' in the K-map which is shown to overlap with OFF2 in one location, thus for the 1 entry in row 3 and column 1. We then use the summation of each column which gives an upper bound on the total number of false alarms if the corresponding literal is dropped and select the column with minimum summation which identifies the literal that will be dropped.

The upper bound will be tighter if each unit of introduced false alarm is shared among fewer off-set cubes. In our example, the false alarm is only included in OFF2 so the upper bound is equal to the number of false alarms.

In case there is a tie when more than one column has the same minimum summation, the one which overlaps with a higher number of on-set cubes is selected which provides opportunity for logic minimization in the subsequent step.

D. Extensions for Multiple Faults and Faulty Units

We first discuss an extension of the presented procedure to handle multiple failures in a single module. In this case the set of test cubes from multiple failures can simply be combined together and minimized and the same procedure can be applied. An interesting aspect here is how frequent false alarms may happen due to each failure. We plan to extend our procedure from this aspect in the future, for example to balance the false alarm occurrence due to each failure.

The second extension is using the presented procedure for false alarm insertion for multiple faulty modules. We explain with a simple example. Consider two faulty modules (a) and (b). The checker in this case generates two recovery signals (one for each module). Denote dedicated false alarm signals with off-set and on-set denoted by OFF_a and ON_a for module (a) and OFF_b and ON_b for module (b). When module (a) is failing, our false alarm insertion should try to avoid failure in module (b) *due to an inserted false alarm*. This means our false alarm insertion should try to map OFF_a to ON_a (and not to ON_b) as much as possible. Similarly OFF_b should try to be mapped to only ON_a as much as possible.

We handle this extension via a modification in the procedure to insert false alarm for a single cube given in Figure 5. Recall in the false alarm matrix, an entry in row i and column j, designates the number of introduced false alarm minterms between the off-set cube i and (the expanded) cube when literal j is dropped. In our modification, when filling an entry in row i and column j for module (a), if dropping literal j results in overlap with ON_b, we *add* an adjustment $k \times M$ where k denotes the number of minterms due to this overlap with ON_b and M is a very large constant (e.g., upper bounded by the maximum amount that an overlap can be).

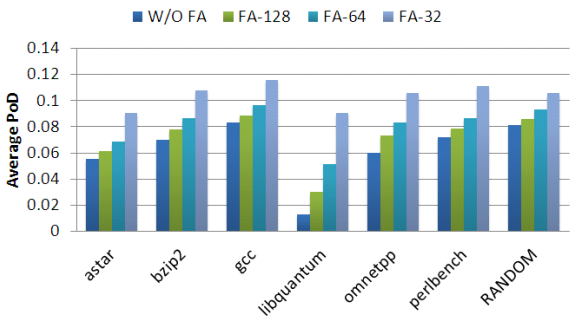

Fig. 6. Comparison of Probability of Detection for a single fault

Note, this amount is an adjustment added to the previous value entered at this entry. Just like before, we select the column with the smallest sum of its entries as the next literal to drop which ensures the column with minimal overlap with ON_b will always have higher priority of selection. Moreover, the procedure ensures that among all the columns having the smallest overlap with ON_b, the one which introduces the smallest number of false alarms in (a) will be selected.

IV. Simulation Results

We divide our validation into three parts, 1) studying the impact of adding false alarms due to a single and due to multiple faults, 2) studying the area overhead, and 3) integration with a redundancy-based technique to improve yield. In part (1), we study a 32-bit Brent-Kung adder [4] by injecting faults at different locations and study the rate of false alarm insertion. In part (2), we assume the adder is part of a 2-issue and 4-issue microprocessor and analyze the area overhead. In part (3) we show our technique can be complimentary to a recent redundancy-based technique [16] using spare bit slices in the adder to further improve yield.

A. Impact of False Alarm Insertion

1) Failure due to A Single Fault: Here we considered single-failure scenarios in various nodes of the adder. For each non-primary input and non-primary output node, we considered two failing cases modeled by a stuck-at-0 and stuck-at-1. For each case, we then generated all the faulty test cubes (which yield to an observable fault in the adder's outputs) using the open-source ATPG tool Atalanta [11]. (Atalanta was able to generate test cubes for about 50% of the nodes in the adder which are the ones considered in our experiments.) The faulty cubes for each case were minimized using the `ESPRESSO` tool [3] using a flow as shown in the example of Figure 3. Minimization reduced the average number of test cubes over the cases from 10,425 to 915 test cubes. Our false alarm procedure was then used for various number of target test cubes (128, 64, and 32 test cubes).

Here we study the impact of false alarm on the frequency of activating the recovery signal. We use simulation for each fault to determine the percentage of the times that the checker activates the recovery signal which we denote by Probability of Detection (PoD). The simulations were done using realistic workloads as well as random input vectors. For the workload-dependent case we considered the benchmarks in the SPEC2006 suite [17]. Specifically, the arguments of the adder were recorded by running each benchmark on an X86 simulator. For the random case, 100K test patterns with uniform probability were generated.

Figure 6 shows the PoD for the following cases: without false alarm insertion (denoted by W/O FA), and with false alarm insertion for target number of test cubes of 128, 64, and 32 bits (denoted by FA-128, FA-64, and FA-32, respectively). For each bar, the reported PoD is averaged over all the corresponding cases. We make the following observations.

- Average PoD degrades with decrease in the number of test cubes (going from 128 to 32). This behavior is expected due to insertion of false alarms.
- Average PoD after inserting false alarms does not degrade significantly in FA-128 or FA-64 or FA-32 compared to W/O FA. (Note the limit on the Y-Axis is only 0.15 probability.)
- The above behavior is true for both workload-dependent and random cases, with the random case typically having a higher PoD especially in the W/O FA case.

We conclude that our false alarm insertion procedure does not degrade the PoD significantly, despite the significant decrease in the number of test cubes (from on average 915 test cubes after logic minimization to 128, 64, and 32 test cubes).

TABLE I
Ratio and Percentage Overhead of False Alarms

# cubes	32		64		128	
	FA	FA-Ag	FA	FA-Ag	FA	FA-Ag
astar	0.23	23%	0.11	54%	0.06	75%
bzip2	0.26	6%	0.19	18%	0.13	32%
gcc	0.20	-3%	0.12	18%	0.07	44%
libquantum	0.38	-20%	0.34	-21%	0.29	-23%
omnetpp	0.28	11%	0.20	22%	0.14	41%
perlbench	0.24	12%	0.14	38%	0.09	54%
RANDOM	0.17	39%	0.10	56%	0.05	76%
Average	0.25	10%	0.17	26%	0.12	43%

We further implemented a variation of our false alarm insertion algorithm in which at each iteration, all the cubes are expanded using the `expandFA` procedure. Recall, in our (default) procedure, the cubes with a lower number of don't-cares are expanded in the earlier iterations, thus for a less aggressive strategy. (See Section III for the description of the default procedure.) We denote the aggressive and default procedures by FA-Ag and FA, respectively.

The FA-Ag procedure results in adding more false alarms per iteration. However, the number of iterations may be smaller because more minimization is possible due to a higher number of expanded cubes per iteration. Therefore, the total number of false alarms induced by FA-Ag may not necessarily be higher than FA if it finishes in a smaller number of iterations. In this experiment, we compare the number of false alarms in two variations of our procedure.

Specifically, we report the fraction of false alarms from the total number of detections. This can be represented by the quantity $\frac{FA}{FA+TP}$ with FA denoting the number of false alarm minterms and TP the number of true positives when a fault is truly happening.

Table I shows this fraction for FA-32, FA-64, and FA-128 in columns 2, 4, 6, respectively. We observe that the fraction of false detections deteriorates from FA-128 to FA-32 which is expected due to insertion of more false alarms. The average ratio of false detections for FA-32, FA-64 and FA-128 are 0.25, 0.17, and 0.12, respectively.

The results for FA-Ag are shown in columns 3, 5, 7, as a percentage of additional false alarms, compared to FA for number of cubes equal to 32, 64, and 128, respectively. The average overheads in false alarms of FA-Ag compared to FA are 10%, 26% and 43%, for 32, 64, and 128, respectively. So on-average FA-Ag results in more overhead with increase in the number of test cubes compared to FA. However, in one case (the `libquantum` benchmark), FA-Ag resulted in fewer false alarms than the FA case. So the aggressive expansion strategy worked out better in this benchmark for this metric.

2) Failure due to Multiple Faults: For this experiment, we generated 1000 random pairings of 2 different faults in the same 32-bit Brent-Kung adder. This corresponds to essentially 1000 different

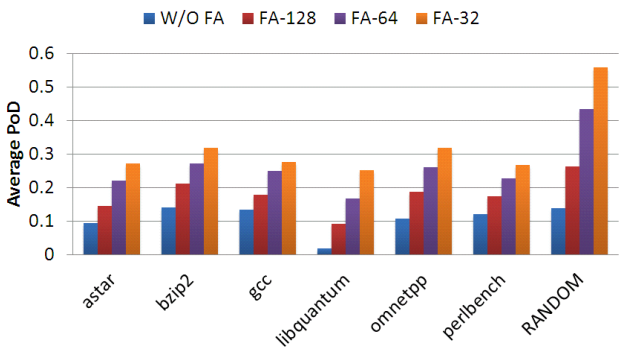

Fig. 7. Comparison of Probability of Detection due to two faults

adders, each containing two faulty nodes. For each adder, we generate (and minimize) all the cubes that excite either fault (or both faults) using the same procedure explained in the previous experiment. We then further minimize each list of cubes using our false alarm insertion algorithm (without aggressive false alarm insertion). Here, again we report the Probability of Detection (PoD) obtained by simulating an X86 processor using various SPEC2006 benchmarks and random workloads, just like the previous experiment. The results are shown in Figure 7. Compared to the previous experiment with a single fault, we observe a similar trend that with increase in the number of test cubes (from 128 to 32 cubes), the PoD degrades, but this degradation is more distinct at each step. We note, for the workload-dependent cases, the PoD is bounded to at most 0.3 for 32 cubes. The random case has a higher PoD, however we note in the presence of multiple faults, the use of realistic workloads is perhaps more appropriate for relevant in our analysis.

B. Comparison of Area and Power

In this experiment, we evaluate the areas of TCAM and FPGA-based implementations of the checker within a 2-wide and 4-wide microprocessor. For the area of the checker unit we consider TCAM and FPGA-based implementation, as explained in Section II for various number of test cubes (32, 48, 64, 128). So we have four variations: 2+TCAM, 2+FPGA, 4+TCAM, 4+FPGA.

We estimate the area of a TCAM-based checker based on the TCAM cell area in 0.18um technology presented in [2] after applying a scaling factor from [22] for 32nm technology.

To evaluate the area of a FPGA-based checker unit, we first derive the truth table from a test cubes set for a failure. Second, we feed Synopsys Design Compiler (SDC) with the derived truth table and a commercial 32nm standard cell library to synthesize the circuit. Based on a prior work comparing area, timing, and power consumption between ASIC and FPGA implementations through an extensive study [10], we use a scaling factor of 35 to compute the FPGA area from the ASIC; an ASIC implementation is 35 times smaller than FPGA implementation for most combinational logic [10].

In our experiments we compared the areas of the following base cases using redundancy:

- 2+1: This case contains two adders (in a 2-issue microprocessor) with one redundant adder. The diagram of this case, including all necessary MUXes is shown in Figure 2(b). The area is taken after synthesis using Synopsys Design Compiler (SDC) using a 32nm Technology library. The area of this base is 4460 μm^2.
- 4+1: This case contains four adders (in a 4-issue microprocessor) with one redundant adder. The area is computed as reported by SDC after synthesis. Our implementation accounted for all the necessary MUXes, similar to the previous case. The area of this base case is 10837 μm^2.

TABLE II
AREA NUMBERS (IN μm^2) OF VARIOUS CHECKER IMPLEMENTATIONS

#cubes	SR	TCAM	FPGA	2+TCAM	4+TCAM	2+FPGA	4+FPGA
32	259	1126	3618	5587	12280	8079	14455
48	389	1552	4973	6013	12706	9434	15810
64	519	1977	6323	6438	13131	10784	17160
128	1038	3732	15563	8193	14886	20023	26400

The above base cases are compared against the following four variations of the checker-based alternative:

- 2+TCAM: This case contains the two adders and instead of one redundant adder, a TCAM is used. The area in this case is computed after synthesizing the two adders and the MUXes for TCAM connection, as shown in Figure 2(a). This post-synthesis area is taken from SDC and is 4682.8 μm^2 which excludes the TCAM area. Next, the TCAM area is also estimated for number of test cubes equal to 128, 64, 48, and 32. The TCAM area for a given size is divided into a memory portion and a shift register (counter) for initial startup, as explained in Section II. For the memory portion, the area is calculated by scaling the TCAM estimate given in [2] for 0.18um technology using the scaling factor from [22] for 32nm technology. The shift register is synthesized using SDC and 32nm library for different number of test cubes. Table II shows the areas of the shift register and the overall area of the TCAM portion in columns 2 and 3 for different number of test cubes. The overall area of 2+TCAM is shown in column 5.
- 4+TCAM: The area of TCAM portion is computed just like the previous case as reported in Table II column 3. The remaining part is composed of 4 adders with proper MUX connections to the TCAM. The area of this portion is 11375 μm^2 after synthesis using SDC. Overall area of 4+TCAM is shown in column 6 for different number of input vectors.
- 2+FPGA: This case contains two adders and one FPGA for the checker unit. The area of the two adders with proper MUXes is 4460 μm^2 after synthesis using SDC. The area of the FPGA portion for different number of test cubes is listed in column 4. It is obtained by synthesizing the corresponding truth tables of test cubes in ASIC and scaling with a factor of 35, based on an extensive recent study [10] estimating the area of combinational logic to be 35X smaller in ASIC. (For different failures, an FPGA was estimated and the worst-case was over all the failure cases was scaled.) The area of 2+FPGA is given in column 7.
- 4+FPGA: The area of the FPGA portion is the same as previous case (given in column 4). The remaining area of the four adders with proper MUX connections is 10873 μm^2 after synthesis using SDC. The overall area of 4+FPGA is listed in column 8.

All adders and MUXes are 32-bit in the above cases.

Table III lists the percentage area overhead of our four variations compared to the areas of the corresponding baselines. The area overheads are listed for the number of test cubes from 32 to 128.

We observe that 2+TCAM and 4+TCAM have 12.5% and 8.8% less area than 2+1 and 4+1 cases, respectively, for 32 number of test cubes. The FPGA-based checker consumes considerably more area than the 2+1 and 4+1 cases. Although the FPGA-based checker for many failures does not require significant area, some failures required a large FPGA, far exceeding the area overhead of integrating a redundant adder. The FPGA-based area can become competitive by not protecting such cases, thus reducing the failure coverage.

Overall, the TCAM-based checker provides similar or lower area for the number of test cubes of 32, 48, and 64.

TABLE III
OVERHEAD OF VARIOUS IMPLEMENTATIONS

	baseline: 2+1		baseline: 4+1	
#cubes	**2+FPGA**	**2+TCAM**	**4+FPGA**	**4+TCAM**
32	26.6%	-12.5%	7.3%	-8.8%
48	47.8%	-5.8%	17.4%	-5.7%
64	69.0%	0.9%	27.4%	-2.5%
128	213.7%	28.4%	96.0%	10.5%

We also discuss the power overhead of the TCAM-based checker. (We do not consider the FPGA case because in our experiments it always had a high area overhead.) For the TCAM case, we observe that most key values are 0s to program don't-cares in TCAM while the columns filled with don't-cares do not discharge the matching lines. For example, for 32, 48, 64 and 128 test cubes stored in TCAM, over all the failure cases, the number of used (non-don't care) bits are 46, 46, 48, and 50, respectively. Therefore, power consumption of the TCAM-based implementation is not significant.

Based on the TCAM model in [2], we estimate that the TCAM-based checker consumes approximately 347fJ, 472fJ, 602fJ, and 1106fJ per cycle (234mW, 318mW, 406mW, and 746mW for 1.2ns cycle time) for 32, 48, 64, and 128 test cubes, respectively; our checker for 48 test cubes consumes 24% and 48% less power than the 2+1 and 4+1 cases.

C. Integration with A Recent Yield Improvement Technique

In our last experiment we show that our scheme is complementary to a redundancy-based scheme for additional improvement in yield.

The work [16] recently proposes a bit-level redundancy scheme in which a data path in a combinational module can be augmented with redundant intermediate bitslices (RIBs) which can be reconfigured after testing to avoid placing nodes that are defective. In this scheme, the datapath is configured to ignore the computation performed in the defective bitslice, and higher-order bits are shifted up, requiring use of additional (one or more) bitslices. The main advantage of this scheme over prior redundancy-based approaches is that the unused bit (and associated logic) remains physically in the datapath, so no additional MUXing is required to route around it. This minimizes area overhead and any impact on delay caused by MUXes.

Using spareRIB for the 32-bit Brent Kung adder, out of the 253 stuck-at-0 and stuck-at-1 defective nodes, spareRIB alone was able to fix 129 nodes. The rest could not be repaired by it.

Next, among the remaining defect cases we identified the ones which had a probability of detection lower than 0.2. We considered such nodes as "repairable" in the sense that they have a sufficiently low runtime overhead to re-execute an instruction when the recovery signal is activated by our checker. Specifically, we first considered all the defect cases and for each one generated its own 32 test cubes using our false alarm insertion algorithm. We then measured the probability of detection in each case by matching its 32 cubes against observed failures in the output during simulation using 100K randomly-generated input vectors. Finally, we identified those defect cases with a probability of detection below 0.2.

After integration of spareRIB with our scheme, we observed that the number of repairable nodes raised from 129 in spareRIB to a total of 168. So 39 additional defects could be repaired. We note, by further increasing the threshold for the maximum probability of detection to a value higher, e.g., 0.3, our framework allows further increase in the number of repairable nodes. This increase can be considered equivalent to yield improvement in the sense that instead of throwing away a faulty chip, we continue its usage but with a higher runtime overhead due to repeating the execution of instructions.

V. CONCLUSIONS

We presented a new framework for online detection of failures at operand level of granularity.The core of our framework is a checker unit which gets programmed after identifying one or more failures within combinational modules; it can also be updated upon observing new failures. For a given failure, if an operand yields to an observable fault, it will be detected by our checker and a recovery flag will be activated at runtime. We presented detailed analysis showing that a TCAM-based implementation of our checker has a smaller area than an alternative using a redundant module. This is with the aid of a proposed algorithm to insert a relatively small number of false alarm test cubes, enabling significant degree of minimization in the number of test cubes to be stored in the checker.

REFERENCES

[1] ESPRESSO. http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/.

[2] B. Agrawal and T. Sherwood. Modeling TCAM power for next generation network devices. In *ISPASS*, pages 120–129, 2006.

[3] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. L. Sangiovanni-Vincentelli. Logic minimization algorithms for VLSI synthesis. *Kluwer Academic Publishers, Boston, MA*, 1984.

[4] R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Trans. on Computers*, C-31(3):260 –264, 1982.

[5] Zhan Chen and Israel Koren. Techniques for yield enhancement of vlsi adders. In *ASAP*, pages 222–229, 1995.

[6] A.N. Doboli and E.H. Currie. Introduction to mixed-signal, embedded design. *Springer Science+Business Media, LLC*, 2011.

[7] S. Garg and D. Marculescu. System-level process variability analysis and mitigation for 3D MPSoCs. In *DATE*, 2009.

[8] Tack-Don Han and David A. Carlson. Fast area-efficient VLSI adders. In *IEEE Symp. on Computer Arithmetic*, pages 49–56, 1987.

[9] T. Jhaveri, L. Pileggi, V. Rovner, and A. Strojwas. Maximization of layout printability/manufacturability by extreme layout regularity. In *SPIE*, 2006.

[10] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE TCAD*, 26(2):203–215, 2007.

[11] H.K. Lee and D.S. Ha. Atalanta: an efficient ATPG for combinational circuits. *Technical Report; Department of Electrical Engineering, Virginia Polytechnic Institute and State University*, pages 93–12, 1993.

[12] R. Leveugle, Z. Koren, I. Koren, G. Saucier, and N. Wehn. The hyeti defect tolerant microprocessor: A practical experiment and its cost-effectiveness analysis. *IEEE Trans. on Computers*, 43:1398–1406, 1994.

[13] Y-W. Lin, M. Marek-Sadowska, and W. Maly. On cell layout-performance relationships in VeSFET-based, high-density regular circuits. *IEEE TCAD*, 30(2):229–241, 2011.

[14] Kazuteru Namba and Hideo Ito. Design of defect tolerant wallace multiplier. In *PRDC*, pages 300–304, 2005.

[15] P. Ndai, Shih-Lien Lu, Dinesh Somesekhar, and K. Roy. Fine-grained redundancy in adders. In *ISQED*, pages 317–321.

[16] D. Palframan, N. Kim, and M. Lipasti. Mitigating random variation with spare RIBs: Redundant intermediate bitslices. In *DSN*, 2012.

[17] A. Phansalkar, A. Joshi, and L.K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *ISCA*, pages 412–423, 2007.

[18] Y. Ran and M. Marek-Sadowska. Designing via-configurable logic blocks for regular fabric. *IEEE TVLSI*, 14(1):1–14, 2006.

[19] N. Ryzhenko and S. Burns. Physical synthesis onto a layout fabric with regular diffusion and polysilicon geometries. In *DAC*, pages 83–88, 2011.

[20] Premkishore Shivakumar, Stephen W. Keckler, Charles R. Moore, and Doug Burger. Exploiting microarchitectural redundancy for defect tolerance. In *ICCD*, pages 35–42, 2012.

[21] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd M. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *ASPLOS*, pages 73–82, 2006.

[22] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The impact of technology scaling on lifetime reliability. In *DSN*, pages 177 – 186, 2004.

[23] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *ISCA*, pages 520–531, 2005.