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
{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 a set of diagnosed failures in combinational modules. The failures may be of various types and may change over time. We propose to utilize this ability to detect failures at operand level of granularity to improve yield, by not discarding those chips containing failing and redundant computational units 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) vectors which excite the set of diagnosed failures. 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 misdiagnosed as failing for a given operand however we show such cases are rare and the chip may continue to be used.

Categories & Subject Descriptors
B.7.2. [Integrated Circuits]: Design Aids
General Terms
Reliability; Algorithms; Design
Keywords
Manufacturing Yield

1. INTRODUCTION
Technology scaling beyond 32nm significantly degrades the manufacturing yield. 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 [8]. A particular challenge in this approach is to avoid significant degradation in power, area, or performance compared to non-regular and flexible design [12].

Alternatively, redundancy at various levels can be used to improve yield. Many prior proposals suggest exploiting redundancy that already exists in high performance processors including [17]. For instance, faulty execution units can be disabled. To avoid the performance penalty due to disabling logic, additional redundancy can be introduced in the form of spare execution units. However this type of redundancy is somewhat coarse-grained and may not be area efficient.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or re-publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GLSVLSI’15, May 20–22, 2015, Pittsburgh, PA, USA.
Copyright © 2015 ACM http://dx.doi.org/10.1145/2742060.2742097.

Figure 1: Overview of our framework
For this reason, other proposals suggest the use of fine-grained redundancy at granularity of a bit slice [5, 13], resulting in potentially more area-efficiency, but imposing a significant performance penalty, since many multiplexors are required to route around a potentially bad slice of logic. Other techniques suggest exploiting existing circuit redundancy. For instance, in [14], 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.

The degree of redundancy which can be exploited in a chip should be constrained 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 its own dedicated redundant module(s). At the same time, the increasing degradation in manufacturing yield requires a higher degree of redundancy. 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 a set of diagnosed failures in failing combinational modules. Our framework is flexible in the sense that all it needs are the vectors which make a computational unit to fail, and it does not care about the source of the failure. 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 in which the failing mod-
ule is a Han-Carlson adder [7]. 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 signals the microprocessor to issue 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) vectors which excite an identified failure. The number of vectors that excite a failure 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. We note, the notion of false alarms in this work is similar to the false triggers in post-silicon debugging in [9].

Our procedure to insert false alarms extends the ESPRESSO tool for two-level logic minimization. It suits a TCAM-based implementation 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.

- Introducing a TCAM-based checker unit as a flexible option for online and operand-level fault detection.
- Proposing the use of false alarm vectors which we show helps significantly reduce the TCAM area, with only a slight increase in the number of misdiagnosis, as verified by realistic workloads on microprocessors of different issue widths.

2. DESIGN OF THE CHECKER UNIT

Consider the example of a 2-issue processor 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 for that operand. The recovery activates the process which ensures the incorrectly-processed instruction will not be committed and instead will be reissued in the pipeline to the other ALU. 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.

2.1 Area Overhead

Implementing the checker in the above example requires MUXes to send the operands of both ALUs to the checker.

Figure 2: Protect a 2-issue faulty processor using (a) a checker unit; (b) a redundant module

For the 2-wide processor in this example, we need two 2-1 MUXes, each for example of 32 bits. For a 4-wide processor, we need two 4-to-1 (32-bit) MUXes. 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. 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 third ALU is connected. There are four 2-to-1 (32-bit) MUXes in this case.

The area overhead of (a) and (b) are shown in red. The use of (a) is desirable if it has a lower area than (b). This is apart from 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).

Two factors may be considered in allotting the area of the checker. 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). The second factor limiting the checker’s area is the recovery overhead (e.g., total runtime spent on re-execution of instructions).

2.2 Implementation Options

Here we compare two options to implement the checker unit: (i) a Ternary Content Addressable Memory (TCAM), and (ii) a Field-Programmable Gate Array (FPGA). We discuss them further in our simulation results.

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. Therefore, supporting a sequential access to write the test cubes to the TCAM is sufficient, which we implement with a simple counter.
Defective node
False Alarm
minterms
(a)
(b) (c)

only 32 test cubes in our simulation setup.

maximum number of entries in the TCAM which we set to

test cubes beneath a target threshold corresponding to the

ing false alarm insertion, we aim to reduce the number of

Specifically, us-

map a few terms from the off-set back into the on-set in order

By adding false alarms we essentially

minimized test cubes.

tion representing a Boolean recovery signal generated by the

alarm function are the points in the cube which are included in

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

2) FPGA: Our second choice is using an embedded FPGA

techology. 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.

2.3 Programming the Checker at Startup

In our scheme, we must store test cubes on-chip and en-

sure they are loaded in the checker unit every time the chip

is turned on. The first choice is to store them 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 and in practice can double the

area overhead. 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 tech-
nology. Hence, we assume that the programming values of

a detector is stored in the flash memory at no extra cost.

3. TEST CUBE MINIMIZATION

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, we propose false alarm

insertion which allows significant reduction/minimization in

the number of stored test cubes. Consider an “alarm” func-
tion 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, us-
ing 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 which we set to

only 32 test cubes in our simulation setup.

Figure 4: False alarm insertion and minimization

Example: Figure 3 shows these steps for the C17 benchmark

circuit from the ISCAS85 suite containing one defec-
tive 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. 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.

3.1 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 identify-

ing 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 re-

duce 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. Specifi-
cally, 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), 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 up-
dated to include the false alarm. The expanded cube now
alarms between the off-set cube and column is equal to the number of literals. An entry in row corresponds to a row in the off-set matrix. The number of dекс. Next, a false alarm matrix is formed where each row the cubes in the off-set and a column gives the literal in-
and represented by a matrix. Each row represents one of
minimized way using three cubes as shown in the K-map
000x in a four-variable function. The off-set is stored in a
example in Figure 5. Consider the cube ON1 with value
a literal for a single cube to achieve these objectives using
subsequent step). We explain our procedure for dropping
(tom increase the opportunities for logic minimization in the
is made such that first, overlap with the off-set is minimized
number of test cubes falls beneath the threshold.

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 con-
tinue 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 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 in-
dex. 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. 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, giving an upper bound on the total number of false alarms if the corresponding literal is dropped, and select the column with minimum summation to identify the dropped literal.

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

3.2 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 algo-


Figure 5: False alarm insertion for one cube

Figure 6: Probability of Detection for a single fault

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.

4. SIMULATION RESULTS

We divide our validation into two parts, 1) studying the impact of adding false alarms, and 2) studying the area over-
head. 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.

4.1 Impact of False Alarm Insertion

4.1.1 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 con-
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 vec-
tors. For the workload-dependent case we considered the
benchmarks in the SPEC2006 suite [16]. Specifically, the
arguments of the adder were recorded by running each bench-
mark 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, respec-
tively). For each bar, the reported PoD is averaged over all
the corresponding cases.

We make the following observations.
strategy worked out better in this benchmark for this metric. False alarms than the FA case. So the aggressive expansion
libquantum
the number of test cubes compared to FA. However, in one
average overheads in false alarms of FA-Ag compared to FA
number of cubes equal to 32, 64, and 128, respectively. The
The average ratio of false detections for FA-32, FA-64 and
32 which is expected due to insertion of more false alarms.
The fraction of false detections deteriorates from FA-128 to FA-
128 in columns 2, 4, 6, respectively. We observe that the
average overheads in false alarms of FA-Ag compared to FA

- Average and variance of 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 and variance 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).

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 3 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 neces-
sarily 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 \( \frac{FA}{FA-Ag} \), with FA denoting the number of false alarm
minterms and TP the number of true positives when a
fault is truly happening.

Table 1 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.

4.2 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 2 for various number of test cubes (32, 48, 64, 128). So we have four variations: 2+T, 2+F, 4+T, 4+F.

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 [18] 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 com-
paring 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 micro-
  processor) 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 \( \mu m^2 \).
- 4+1: This case contains four adders (in a 4-issue micro-
  processor) with one redundant adder. The area is
computed as reported by SDC after synthesis. The im-
plementation accounted for all the necessary MUXes, similar to the previous case. The area of this base case
is 10837 \( \mu m^2 \).

The above base cases are compared against the following four variations of the checker-based alternative:
- 2+T: 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 \( \mu m^2 \) which excludes the TCAM
area. Next, the TCAM area is also estimated for num-
ber of test cubes equal to 128, 64, 48, and 32. The
TCAM area for a given size is divided into a mem-
ory portion and a shift register (counter) for initial
startup, as explained in Section 2. 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 [18] for 32nm technology. The shift
register is synthesized using SDC and 32nm library for
different number of test cubes. Table 2 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+T is shown in column 5.
- 4+T: The area of TCAM portion is computed just
  like the previous case as reported in Table 2 column
3. The remaining part is composed of 4 adders with
proper MUX connections to the TCAM. The area of
this portion is 11375 \( \mu m^2 \) after synthesis using SDC.

<table>
<thead>
<tr>
<th># cubes</th>
<th>FA</th>
<th>FA-Ag</th>
<th>FA</th>
<th>FA-Ag</th>
<th>FA</th>
<th>FA-Ag</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.23</td>
<td>23%</td>
<td>0.11</td>
<td>54%</td>
<td>0.06</td>
<td>75%</td>
</tr>
<tr>
<td>64</td>
<td>0.26</td>
<td>6%</td>
<td>0.19</td>
<td>18%</td>
<td>0.13</td>
<td>32%</td>
</tr>
<tr>
<td>128</td>
<td>0.20</td>
<td>-3%</td>
<td>0.12</td>
<td>18%</td>
<td>0.07</td>
<td>44%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>library</th>
<th>RANDOM</th>
<th>gcc</th>
<th>g++</th>
<th>battpi2</th>
<th>astar</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.20</td>
<td>39%</td>
<td>0.10</td>
<td>56%</td>
<td>0.05</td>
</tr>
<tr>
<td>64</td>
<td>0.24</td>
<td>12%</td>
<td>0.14</td>
<td>38%</td>
<td>0.09</td>
</tr>
<tr>
<td>32</td>
<td>0.28</td>
<td>11%</td>
<td>0.20</td>
<td>22%</td>
<td>0.14</td>
</tr>
</tbody>
</table>

| Average | 0.25 | 10% | 0.17 | 26% | 0.12 | 43% |

<table>
<thead>
<tr>
<th>libquantum</th>
<th>bzip2</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.07</td>
<td>56%</td>
<td>0.12</td>
</tr>
<tr>
<td>0.14</td>
<td>38%</td>
<td>0.10</td>
</tr>
<tr>
<td>0.19</td>
<td>18%</td>
<td>0.12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>omnetpp</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>76%</td>
</tr>
<tr>
<td>0.08</td>
<td>54%</td>
</tr>
<tr>
<td>0.11</td>
<td>41%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gcc</th>
<th>bzip2</th>
<th>libquantum</th>
<th>omnetpp</th>
<th>perlbench</th>
<th>RANDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>76%</td>
<td>0.08</td>
<td>54%</td>
<td>0.11</td>
<td>41%</td>
</tr>
<tr>
<td>0.08</td>
<td>54%</td>
<td>0.11</td>
<td>41%</td>
<td>0.19</td>
<td>18%</td>
</tr>
<tr>
<td>0.07</td>
<td>56%</td>
<td>0.12</td>
<td>43%</td>
<td>0.11</td>
<td>41%</td>
</tr>
</tbody>
</table>
Overall area of 4+T is shown in column 6 for different number of input vectors.

- **2+F:** This case contains two adders and one FPGA for the checker unit. The area of the two adders with proper MUXes is 4460 \( \mu \text{m}^2 \) 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+F is given in column 7.

- **4+F:** 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 \( \mu \text{m}^2 \) after synthesis using SDC. The overall area of 4+F is listed in column 8.

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

Table 3 lists the percentage 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+T and 4+T 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.

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 347J, 472J, 602J, and 1106J 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.

### 5. 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.

### 6. REFERENCES


