CMOS technology. Fig. 6 shows the layout of this FIR processor with a die size of 6.3 $\times$ 6.3 mm^2. The power consumption is around 3.8 W at a clock frequency of 200 MHz, where the power and functionality of the proposed processor were analyzed using the PowerMill and TimeMill tools. The throughput rate is 50 MHz for 8-bit input data and 25 MHz for 16-bit input data. There are 241 742 transistors in the design, and the circuit density is around 164 μm^2 per transistor. When considering 8-bit input data and (anti)-symmetric filter coefficients, the proposed FIR processor can have a maximum of 64 taps with a computational power of 12.8 billion multiplication-accumulation operations/s. The specifications of the proposed FIR processor are listed in Table IV. Additionally, performance comparisons between the proposed FIR processor and processors presented in other work are listed in Table V, where the factors of area per 10×12-bit multiplier A(mult) and power per 10×12-bit multiplier P(mult) are defined using a 0.5 μm CMOS technology, a 3.3 V supply voltage, and a 50-MHz clock rate [1]–[4]. The memory-based FIR processor with the 8-bit input data and 9-bit filter coefficients dissipates a higher power [4]. The CSD FIR processor has the lowest power consumption and the smallest area because values of filter coefficients are rounded off by a few nonzero digits [3]. As compared to the other Booth-algorithm FIR processors [1], [2], the proposed processor is highly flexible, and has a good throughput rate, a fair die size and a reasonable power consumption in a 5 V standard cell implementation point of view.

V. Conclusion

Based on the radix-4 Booth algorithm, this work has successfully developed an FIR architecture with programmable dynamic ranges of input data and filter coefficients. Notably, the proposed architecture employs only data-path controls to accomplish programmable operations. A practical FIR processor with 8-bit and 16-bit dynamic ranges of input data and filter coefficients was also implemented by using the TSMC 5 V 0.6 μm CMOS technology. Moreover, the processor is optimized for odd or even (anti)-symmetric and asymmetric filter coefficients in ten operation modes. This programmable FIR processor can be operated at a clock frequency of 200 MHz to produce throughput rates of 50 M and 25 M samples/s for 8-bit and 16-bit input data of various industrial applications, respectively.

Acknowledgment

The authors would like to thank the reviewers for their valuable comments and suggestions.

References

[1] C. Nicol, P. Larsson, K. Azadet, and J. O'Neill, "A low-power 128-tap digital adaptive equalizer for broadband modems," *IEEE J. Solid-State Circuits*, vol. 32, pp. 1777–1789, Nov. 1997.

[2] J. Choi, S. Jeong, L. Jang, and J. Choi, "Structured design of a 288-tap FIR filter by optimized partial product tree compression," in *Proc. IEEE CICC*, May 1996, pp. 79–82.

[3] K. Khoo, A. Kwentus, and A. Willson, Jr., "A programmable FIR digital filter using CSD coefficients," *IEEE J. Solid-State Circuits*, vol. 31, pp. 869–874, June 1996.

[4] C. Golla, F. Nava, F. Cavallotti, A. Cremonesi, P. Piacentini, and G. Casagrande, "A 30M samples/s programmable filter processor," *Digest IEEE Int. Solid-State Circuits Conf.*, pp. 116–117, Feb. 1990.

[5] W.-L. Liu and O. T.-C. Chen, "A highly-scaleable symmetric/asymmetric FIR processor," in *Proc. IEEE Int. Conf. Acoustic, Speech, Signal Processing*, Mar. 1999, pp. 1917–1920.

[6] J. Yuan and C. Svensson, "New single-clock CMOS latches and flipflops with improved speed and power savings," *IEEE J. Solid-State Circuits*, vol. 32, pp. 62–69, Jan. 1997.

On-Line Test for Fault-Secure Fault Identification

Samuel N. Hamilton and Alex Orailoğlu

***Abstract*—In an increasing number of applications, reliability is essential. On-line resistance to permanent faults is a difficult and important aspect of providing reliability. Particularly vexing is the problem of fault identification. Current methods are either domain specific or expensive. We have developed a fault-secure methodology for permanent fault identification through algorithmic duplication without necessitating complete functional unit replication. Fault identification is achieved through a unique binding methodology during high-level synthesis based on an extension of parity-like error correction equations in the domain of functional units. The result is an automated chip-level approach with extremely low area and cost overhead.**

***Index Terms*—Faults, reliability, reliable, test, ULSI, VLSI.**

I. Introduction

As the sophistication and complexity of modern electronics increases, so does our reliance on it. Utilization in critical areas such as medicine, navigation, and transportation is already high and continues to rise. More and more pilots, surgical patients, and even everyday motorists bet their lives daily on the reliability of their electronics. Thus, the need for reliable computational equipment is both immediate and increasing.

Due to the dearth of efficient approaches to fault identification, designers frequently resort to complete duplication of a set of functional units in order to detect errors. Replication allows two units to compare duplicate calculations for consistency. If the calculations are run on disjoint hardware, the design is *fault secure* at the single fault level, as no single fault can corrupt output without detection. When an error does occur, it still must be determined which of the two calculations is correct. The consequent hardware required for fault identification entails additional complexity.

To avoid the cost associated with complete hardware duplication, there has been an increasing focus in the fault-tolerance literature on high-level synthesis [1]. Tailoring high-level synthesis routines for compatibility with efficient hardware solutions facilitates the production of more compact, reliable designs.

In this paper, we propose an algorithmic approach to fault identification that introduces error encoding schemes into high-level synthesis. By endowing scheduling and binding routines with the capacity to embed generic error correction codes, we enable efficient implementation of calculation duplication via load balancing while avoiding hardware complexity associated with traditional fault identification techniques. By adopting algorithmic duplication, we also introduce the capacity for fine grain reliability/cost tradeoffs similar to performance/cost tradeoffs inherent in high-level synthesis while simultaneously guaranteeing fault security for single faults. In addition, unlike approaches based on more restricted fault models, a voting or function unit fault cannot result in erroneous behavior. The proposed technique represents the first fault-secure methodology for fault identification to forgo triplication in favor of the significantly lower cost duplication entails.

Manuscript received November 23, 1998; revised February 12, 1999.

The authors are with the Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114 (e-mail: hamilton@cs.ucsd.edu; alex@cs.ucsd.edu).

Publisher Item Identifier S 1063-8210(00)01036-2.

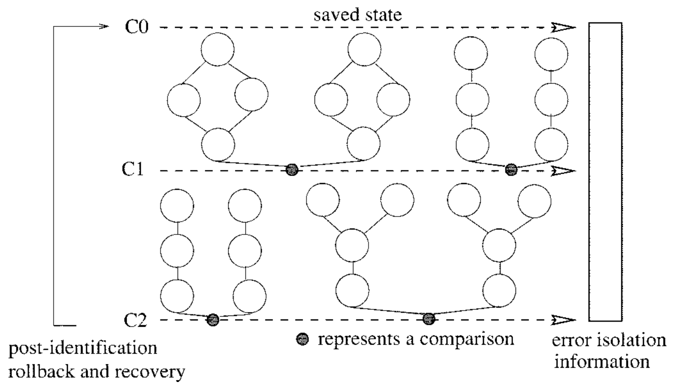
saved state
C0
C1
C2
post-identification rollback and recovery
represents a comparison
error isolation information

Fig. 1. Overall fault isolation and recovery structure.

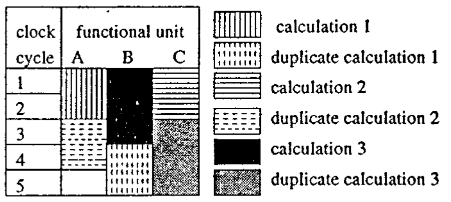

Fig. 2. Fault detection through algorithmic duplication.

TABLE I
HARDWARE FOR TRACKS REPRESENTED IN FIG. 2

track reporting error	potentially faulty units
1	A B
2	A C
3	B C

II. Overall Approach

The algorithmic approach we propose aims to combine an efficient interjection of a small set of hardware with carefully tailored high-level synthesis routines to achieve fault tolerance with low area overhead. Detection is achieved through algorithmic duplication, which can avoid full functional unit duplication through load balancing. Fault identification is accomplished through the insertion of error codes during high-level synthesis. Fig. 1 outlines the operation of a system utilizing our technique.

Errors are detected through comparison of all calculations and duplicate calculations at periodic checkpoints, represented as C0–C2 in our diagram. Note that as shown in Fig. 1, there can be significantly less than one comparison per operation. Information reflected by reported errors is collected and used for fault identification. When identification has occurred, the system is reconfigured to remove the faulty unit, and rollback to the last saved state is initiated.

In this paper, we do not address rollback insertion or reconfiguration methodology, but concentrate on fault identification. For efficient checkpoint insertion and reconfiguration techniques, refer to [1] and [2], respectively. Our fault identification methods adopt the following assumptions.

1) As in most fault models in this domain, during fault identification we assume only a single functional unit is faulty. Multiple faults can be handled, however, as long as they do not fall within the same fault identification period.
2) We adopt a Byzantine fault model, wherein faults are not assumed to produce consistently erroneous behavior. This fits with data from testing literature, which indicates that many faults are neither catastrophic, nor covered by limited fault models [3].
3) We restrict our exposition of fault tolerance techniques to within basic blocks. Consequently, neither conditional branches nor loops are addressed in our description. This is not a fundamental restriction, but is made to simplify the presentation of our main ideas.

Initially, we will also assume fault-free voting hardware. This assumption is waived through incorporation of voting hardware into the fault model in Section IV.

A. Error Detection

The basic approach to error detection is outlined in Fig. 2. Each calculation is duplicated, allowing an error to be detected through comparing the results of identical strings of computations. Of course, a calculation and its duplicate cannot share functional units. If they did so, a fault in a single unit could corrupt both the calculation and its duplicate. If this corruption occurred such that both erroneous results agreed, the erroneous behavior would escape detection. For subsequent legibility, we denote a string of operations as a *string*, while a string and its duplicate is referred to as a *track*.

By ensuring that strings and their duplicates utilize disjoint sets of units, we ensure that erroneous results cannot remain undetected. Of course, this is reliant on a single faulty unit assumption. Otherwise, a fault could occur in sets of units such that a string and its duplicate both arrive at the same incorrect result. Since the agreement between a string and its duplicate is interpreted as an *error-free* condition, the erroneous result would escape detection. Note that any system reliant on duplication for error detection implicitly adopts this assumption, since it is always possible for a functional unit performing a calculation and the functional unit checking that calculation to produce erroneous results simultaneously.

B. Fault Identification

While error detection is useful, our overall goal is faulty unit identification. Error detection through calculation duplication does not accomplish this. The error detection process does, however, contain information that can be useful for fault identification: if a track reports an error, one of the units active in that track is faulty. Thus, as can be seen in Table I, if track 1 reports an error, either unit A or unit B must be faulty.

Note that the information present in any one track is only adequate to narrow the set of potentially faulty units to two. By combining the information provided by multiple tracks, however, we can still achieve complete identification. For example, if track 1 and track 2 from Table I both report errors, it can be deduced that unit A is faulty, since it is the only unit used in both tracks. Similarly, if tracks 1 and 3 report errors, unit B must be responsible.

A subtle issue raised by Fig. 2 is the possibility of *error masking*. Since strings can contain more than one operation, it is possible that a faulty unit will produce incorrect results for an operation, but a subsequent operation in the same string will correct the error before the string is compared with its duplicate. Although error masking can result in faults remaining undetected that would be detected if every operation were checked, it cannot result in undetected errors in the output. This is because every output is the culmination of a calculation, and by definition masked errors produce correct results at the end of output. Thus, our system remains completely fault secure despite reduced comparator requirements.

Error masking is not the only way a faulty unit can remain undetected. Since many faults only produce incorrect results under a limited set of circumstances, they cannot always be detected through calculation duplication. This is particularly true with hard-to-test faults, where the fault only rarely produces erroneous behavior. Tracks reflecting

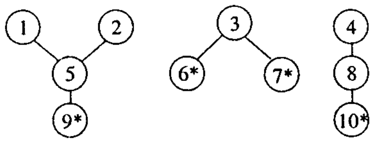

Fig. 3. Data flow graph, with asterisk denoting output nodes.

TABLE II
POSSIBLE SCHEDULING OF DATA FLOW GRAPH IN FIG. 2 AND ITS DUPLICATE USING SEVEN UNITS. A QUOTE DESIGNATES A DUPLICATE NODE

time	A	B	C	D	E	F	G
clk 1	1	2	1'	2'	4'	4	3
clk 2	5	8'	5'	3'		8	6
clk 3	9	10'	7'	9'	6'	10	7

TABLE III
TRACKS FOR THE SCHEDULE PRESENTED IN TABLE II. STRINGS WITHIN A TRACK ARE SEPARATED BY A SLASH

track	nodes	units utilized
1	1 2 5 9	A B / C D
2	3 6	G / D E
3	3 7	G / C D
4	4 8 10	F / B E

TABLE IV
INCLUSION PROPERTY OF UNITS IN TRACKS LISTED IN TABLE III

	unit included in track						
	A	B	C	D	E	F	G
tracks	1	1 4	1 3	1 2 3	2 4	4	2 3

correct behavior by a faulty unit are referred to as *false negatives*, since they fail to signal the presence of a fault. False negatives interfere with the identification process, since they reduce the number of tracks reporting useful information to the fault-identification process.

C. Motivating Example

Fig. 3 shows a sample data flow graph for which we illustrate some of the advantages inherent in algorithmic duplication, as well as the challenges false negatives present.

The data flow in Fig. 3 can be scheduled in three clock cycles using four functional units, indicating that hardware duplication would result in eight units. As Table II shows, however, with algorithmic duplication the data flow in Fig. 3 can be scheduled using only seven units, without increasing the number of clock cycles. Thus, algorithmic duplication can be achieved with less-than-complete hardware duplication.

The tracks inherent in the schedule in Table II are presented in Table III. In order for a specific functional unit to be identified, it needs to be in a unique set of tracks. Otherwise, it could not be differentiated from units with identical track sets. As can be seen in Table IV, the tracks in Table III satisfy this property.

Due to the possibility of false negatives, however, it is not sufficient for each unit to have a unique set of tracks. If the set of tracks that include unit a are a subset of the tracks including unit b, ambiguity can arise. The problem is that if a's tracks report errors, it cannot be determined whether unit a is faulty, or whether b is faulty but some of its tracks are not reporting errors. For example, in Table IV, unit A is included in track 1, which is a subset of the tracks including units B, C, and D. If track 1 reports an error, either A is faulty, or B, C, or D is faulty with false negatives in other tracks.

TABLE V
HAMMING TRACKS FOR SEVEN UNITS

track	units utilized
1	A B C D
2	A B E F
3	A C E G

TABLE VI
DIFFERENTIATION FOR HAMMING CODE USING SEVEN UNITS

unit	number of times differentiated from						
	A	B	C	D	E	F	G
A	-	1	1	2	1	2	2
B	0	-	1	1	1	1	2
C	0	1	-	1	1	2	1
D	0	0	0	-	1	1	1
E	0	1	1	2	-	1	1
F	0	0	1	1	0	-	1
G	0	1	0	1	0	1	-

Consequently, there is no reliable way of identifying unit A as faulty without an alternative track decomposition. To enable identification of any faulty unit within a set of tracks, the fault behavior of each unit must be *differentiated* from all other units. Unit a is considered differentiated from unit b if a track exists that utilizes a, but not b. Thus, in track 1 of Table III, A, B, C, and D are differentiated from units E, F, and G. Note that differentiation is not a symmetrical property, since adding a track prevents units in the track from being subsets of excluded units, but does not prevent excluded units from being subsets of included units. Thus, while unit B in Table III is differentiated from unit F by track 1, the tracks of F are still a subset of the tracks of B.

III. ENCODING ERROR IDENTIFICATION PROPERTIES

A common error encoding technique for fault identification is Hamming code [4]. There are some subtle differences, however, between the application of Hamming code to bit correction, and to functional unit fault identification.

Following is an analysis of the properties of Hamming code, followed by methods to extend it to allow complete identification. While there is no straightforward method to apply the theoretical track sets derived to complex data flow diagrams, the analysis of these sets supplies useful techniques for implementing tracks for any data flow.

A. Hamming Track Sets

Hamming tracks are based on a logarithmic technique, where every track differentiates one half of the units from the other half. Table V shows Hamming tracks for seven units.

When applying Hamming to faulty unit identification instead of bit correction, the method of error detection is quite similar. In Hamming tracks, an erroneous track is detected using a comparison unit, which is assumed to be fault-free.[1] In Hamming bit correction code, the hardware used to check parity is also assumed to be fault-free. The hardware associated with checking parity (XOR), however, is significantly simpler than the voting hardware required for checking tracks (a comparator).

The simpler hardware requirements result from the addition of a parity bit to each parity equation. Since each parity bit is itself subject to error, n parity equations can only correct $2^n - n - 1$ bits of information. Hamming tracks, however, have no equivalent requirements, and can therefore include up to $2^n - 1$ functional units in n tracks.

Table VI shows the differentiation properties of the Hamming tracks shown in Table V. The entry in row A, column B of Table VI indicates

[1]This assumption is waived in Section IV.

TABLE VII
MIRROR HAMMING TRACKS FOR EIGHT FUNCTIONAL UNITS

track	units utilized
1	A B C D
2	A B E F
3	A C E G
4	E F G H
5	C D G H
6	B D F H

TABLE VIII
DIFFERENTIATION FOR MIRROR HAMMING CODE USING EIGHT UNITS

unit	number of times differentiated from							
	A	B	C	D	E	F	G	H
A	-	1	1	2	1	2	2	3
B	1	-	2	1	2	1	3	2
C	1	2	-	1	2	3	1	2
D	2	1	1	-	3	2	2	1
E	1	2	2	3	-	1	1	2
F	2	1	3	2	1	-	2	1
G	2	3	1	2	1	2	-	1
H	3	2	2	1	2	1	1	-

that there exists one track where A is present and B is not (track 3 from Table V). The zero entry in row B, column A shows that no tracks contain B without including A. Thus, B is not differentiated from A. Consequently, if the tracks containing B (1 and 2) report errors, it cannot be determined whether unit B is faulty or whether A is (since track 3 could have a false negative). Since not all units are differentiated from all other units, Hamming tracks are insufficient for fault identification.

B. Extensions to Hamming

Despite the flaws in Hamming track sets, they possess a useful characteristic: all chart entries above the diagonal are greater than zero. Using this fact, we can derive a new encoding scheme that fulfills our differentiation requirements. The new code is called *mirror Hamming*, because it is derived by combining Hamming with its complement. The complement will by definition produce a mirror image of the differentiation chart, since for each track every unit previously missing is differentiated from all units previously present. Therefore, Hamming combined with its complement is fully differentiated.

Note that when implementing complementation, the set of units used in n Hamming tracks is 2^n, not the $2^n - 1$ units originally included. As alluded to previously, Hamming is derived using a logarithmic approach to creating unique track sets for each unit. The unit assigned the empty set of tracks cannot be included, of course, because by definition it is not used in any of the tracks. During complementation, however, the corresponding unit is included in all new tracks, and can therefore be utilized. Table VII shows mirror Hamming tracks for eight units.

From the differentiation chart in Table VIII, it is clear that complete differentiation has been achieved. Not only that, but the number of tracks required using this technique is $2(\lg n)$, which allows functional units to be fully differentiated by relatively few comparisons.

Complete differentiation such as that shown in Table VIII is indicative of a *minimum differentiation* of one, i.e., the smallest entry in the differentiation chart is one. Minimum differentiation is equivalent to the number of false negatives required to create an ambiguity. This is implicit in the definition of differentiation, since every differentiation of a unit a from another unit b represents an additional track where a false negative would be required for unit a's tracks to become a subset of unit b's tracks.

Hamming code is not unique in its capacity to supply complete differentiation when mirrored. The mirror of any error correction code is sufficient for complete differentiation. This is because error correction codes require a unique set of bits to be included for any fault, which is sufficient to fill the upper half of the triangle in a differentiation chart as Hamming code does in Table VI. The mirror then covers the other half. While tangential to the fault identification work described herein, it is interesting to note the reverse is also true. Any code that fills the upper half of a differentiation table is sufficient for error correction.

To reduce the chance of an ambiguity arising, it is desirable to have track sets with high minimum differentiations. Refinement of theoretical track sets such as mirror Hamming is an impractical solution, however, as data dependencies and timing constraints limit straightforward track formation. Section VI addresses derivation techniques for realistic track sets within the context of high-level synthesis utilizing the analysis techniques developed in this section.

IV. FAULTY VOTING HARDWARE

The previous analysis has been based on the assumption that voting hardware exhibits fault-free behavior. In this section, we incorporate voting hardware into our fault model through a functional analysis of potential voting faults. The two possible fault behaviors of a voting unit are:

- the voter fails to report an error in the checked track;
- the voter reports an error in the checked track when no error exists.

Note that by the single fault assumption, if the checked track contains an error, the voting unit cannot be faulty. Thus, the only fault condition that needs to be considered is the report of an error when the strings of the checked track are in agreement.

To identify faults in voters that incorrectly report errors, we can include the voters in the set of units they check. Through inclusion in the differentiation chart, faults in voters can be identified through the previously established techniques.

V. EXTENSIONS TO MULTIPLE SEQUENTIAL FAULTS

The techniques described above supply complete fault security in a single fault environment. In fact, as long as only one fault occurs at a time, the fault identification system remains fault secure under multiple sequential faults. For example, if multiple spares are included, our system could identify and replace multiple faults as long as only one occurred at a time.

This is a very reasonable fault model for permanent faults, as any chip which passes manufacturing testing will likely develop permanent faults very slowly, allowing each to be caught and identified separately. If only one fault occurs at a time, the chance of the fault being detected and identified approaches 100% as the time period between faults approaches infinity, given random input.

The only place this model loses realism is in the case of dedicated comparators. In the case of comparators, the input is not random. In fact, if a zero output from a comparator indicates no fault, only input resulting in a zero output will occur unless there is a noncomparator fault. This means if a voting unit were to develop a stuck-at-zero output fault, that fault could remain undetected indefinitely. While the application would continue to function correctly, the prolonged period without detection increases the likelihood of another fault occurring. These two faults could then interact such that the faulty voting unit masks errors in the second faulty unit. Note that this cannot happen with voting units containing stuck-at-one output faults, as the frequent error reports will enable quick identification.

One solution to this problem is the use of nondedicated comparators. That is, check other units using spare cycles in adders or other units that can conduct comparisons. Since these units are responsible for other operations, these other operations will detect stuck-at-zero faults. In addition, this reduces overhead by reducing dedicated comparator requirements.

VI. Synthesis

While Section III sets the theoretical foundation for on-line fault identification, it does not cover synthesis issues. The main issues to address are how to schedule and bind a dataflow graph into tracks, and how to store track information with minimal overhead. In this section, we address these issues, and describe our implementation of a high-level synthesis algorithm. Application of our algorithm to a benchmark from our results section is shown step by step to clarify and illustrate our techniques.

A. Scheduling and Binding

Current high-level synthesis scheduling and binding algorithms do not address differentiation issues. Consequently, modifications are required. When adjusting scheduling and binding algorithms to facilitate differentiation, our priority is to avoid sacrificing performance.

Binding is the fundamental component in determining fault identification properties, as it determines what units are included in each track. As it is rarely possible to bind according to *a priori* track sets such as mirror Hamming, we do not attempt such a mapping. Instead, we judge each track's contribution to a differentiation chart, and use this as a criterion to bind nodes in a greedy fashion. In our implementation, binding is interweaved into the scheduling process. We do this to enable more intelligent comparison insertion during scheduling, as insertion of a comparison operation dictates the formation of a track.

As we prioritize performance over differentiation, if binding becomes disruptive to scheduling (due to the need to maintain disjointness), a rebinding algorithm is called, which unbinds relevant nodes and comparisons. The nodes are then rebound and comparisons inserted in a simplified fashion, where track sizes are minimized by reusing functional units between parents and children. If this is not sufficient, the program delays scheduling the relevant node.

For most applications, it is neither possible nor desirable to fit all operations into several long tracks. Long tracks limit minimum differentiation potential and increase error latency. The solution is to *cut* the length of tracks and, after checking for errors, introduce a new set of tracks. These checks are done in parallel with the next clock cycle of operations. We introduce cuts when functional units capable of a comparison are idle. In applications utilizing adders or other units capable of comparison, this reduces overhead by eliminating dedicated comparator hardware.

Fig. 4 shows the basic structure of our scheduling and binding algorithm. The application of that algorithm results in the schedule in Table IX. The node number scheme follows a breadth first ordering of the auto regressive (AR) filter dataflow presented in [5]. The hardware input consists of seven functional units capable of multiplication and addition, and two functional units capable of addition. Note that since addition can be used for comparison purposes, all functional units in this example may be scheduled to do comparisons. For simplicity of illustration, each functional unit is given a uniform amount of time to complete an operation.

The ability of each functional unit to do a comparison creates an abundance of opportunities for track formation. In fact, it is possible in this example to compare all 28 replicated data nodes, creating 28 tracks, without increasing the number of functional units. There are only 19 tracks implicit in Table IX, as shown in Table X, however, which raises the question why more idle functional units were not used for comparison. The answer is that a comparison may potentially stretch the lifetime of the relevant register. For example, the result for node 1 is calculated in clock cycle 1, and used in clock cycle 2 by node 5. If node 1 were compared in clock cycle 6, the lifetime of the register storing its value would be significantly stretched. We reduce the effect of this phenomenon by taking register lifetime increases into account during our derivation of the best comparison insertion.

```
High_Lvl_Synth(Graph, FUs, Regularize?) {
  Graph = Duplicate(Graph);
  CC = 0;
  while (∃ uncompared leaf in Graph) {
    while (∃ schedulable node) {
      if (!bind(Graph.bestSchedulable(CC,FUs))) {
        if (!rebind(Graph.bestSchedulable(CC,FUs)))
          Graph.restrictBestSchedulable(CC);
      }
    }
    while (∃ free FU for comparison) {
      new_compare = Graph.bestCompare(CC,FUs);
      Graph.insertComparison(new_compare);
      Tracks.addTrack(new_compare);
    }
      CC++;
  }
  if (Regularize?) Tracks.feedBleedBreed();
}
```

Fig. 4. Fault detection through algorithmic duplication.

TABLE IX
Schedule and Binding of AR Filter in Ten Clock Cycles. A Quote Designates a Duplicate Node, and an Asterisk Represents a Comparison

	multiplier/adders							adders	
time	A	B	C	D	E	F	G	H	I
clk 1	1	1'	2	2'	3	3'	4		
clk 2	4'	6	15	15'	16	16'	17	5	5'
clk 3	7'	17'	8	18	7	18'	23	23'	6'
clk 4	10'	11'	24	9	24'	10	11	6*	8'
clk 5	9'	12	12'	8*	13	7*	10*	24*	18*
clk 6	13'	14'	20	11*	12*	21	9*	14	
clk 7	20'	19'	19	22	21'	22'	14*	13*	
clk 8	26	26'	21*	19*	20*		22*	25'	25
clk 9	27	28'	25*	26*				28	28'
clk 10		27*							28*

B. Track Storage

The tracks in Table X have a minimum differentiation of 2, as shown in Table XI. To allow online fault identification, the track information must be encoded so that a comparator can trigger the union of the relevant track set and the ambiguity set.

A simple approach to storing track binding is to embed n bits of information for each checking operation, where n is the number of hardware units. These bits represent the units utilized by each track, and can be intersected with the ambiguity set upon detection of an error.

For applications with a large number of tracks, this approach can entail significant overhead. One technique to decrease this overhead is to limit the set of potential track bindings. A multiplexer stores these bindings, which are selected according to bits of information embedded in the checking operation. As long as the number of possible tracks is no greater than 2^{n-1}, this will result in a reduction in the bits of information that must be stored with the checking operation. In our example, the number of bits per comparison reduces from nine to five, as there are 19 tracks.

Since the size of the multiplexer and the number of bits of information required for binding identification are dependent on the number of possible track bindings, it is desirable to minimize them. This minimization must be done within the context of maintaining differentiation

TABLE X
TRACKS FOR INSERTED COMPARISONS

track	node compared	units utilized
1	6	A B E F G H I
2	18	D F I
3	24	B C E G H
4	8	C D I
5	7	A B C D E F H I
6	10	A F G
7	9	A D G
8	11	B D G
9	12	B C E
10	13	A E H
11	14	B G H
12	21	C E F
13	19	B C D
14	20	A C E
15	22	D F G
16	25	C H I
17	26	A B D
18	27	A B C D E F G H I
19	28	B H I

TABLE XI
DIFFERENTIATION TABLE FOR IRREGULAR TRACKS

	A	B	C	D	E	F	G	H	I
A	X	4	5	4	3	4	4	4	5
B	6	X	5	5	5	7	5	4	6
C	6	4	X	5	3	6	7	5	5
D	5	4	5	X	7	5	5	7	5
E	3	3	2	6	X	4	5	3	5
F	3	4	4	3	3	X	3	4	3
G	4	3	6	4	5	4	X	4	6
H	4	2	4	6	3	5	4	X	3
I	4	3	3	3	4	3	5	2	X

properties. One technique is to *regularize* the track set. This involves deriving a small core set of track bindings, and mapping checking operations into this core set. During mapping, it is important to ensure that checking operations are only mapped to core tracks if the track binding of the checking operation is a subset of the binding of the core track. As long as this subset property holds true, a faulty hardware unit cannot be incorrectly eliminated from the ambiguity set.

The disadvantage of regularization is that differentiation is weakened. For example, if a check using units (A B D) is mapped to the core track (A B C D), unit C would not be eliminated from the ambiguity set despite not having been utilized in the original track. Thus, differentiation is redefined such that all units in the original track are differentiated from all units not in the core track being mapped to. For applications with numerous tracks, however, differentiation is already likely to be quite good. Since this is also the application group which derives the most benefit from regularization, the technique should be seriously considered for any implementation with a large number of tracks.

The main challenge involved in the regularization of tracks is the selection of a core track set. Once this is done, a greedy algorithm can be used to map each check to the core track which improves differentiation the most.

To select x core tracks, we utilize a genetic algorithm approach which incorporates elements of simulated annealing. Following the generation of a set of seed tracks, it iterates through the following steps until only x tracks remain.

1) Select a group of x tracks.
2) *Feed* each track based on the differentiation properties of the group as well as individual contributions.

TABLE XII
REGULAR TRACKS AND MAPPING FROM IRREGULAR TRACKS

track	units	subsumed irregular tracks
1	D F I	2
2	A E H	10
3	B G H I	11 19
4	B C E F G	9 12
5	A C D E F G	6 7 14 15
6	C D H I	4 16
7	A B C D F H I	13 17
8	A B C D E F G H I	1 3 5 8 12 18 19

TABLE XIII
DIFFERENTIATION TABLE FOR REGULAR TRACKS

	A	B	C	D	E	F	G	H	I
A	X	4	1	1	1	1	2	3	4
B	3	X	2	3	4	2	2	1	1
C	4	3	X	2	3	2	3	3	3
D	2	4	1	X	4	1	4	3	2
E	2	2	1	3	X	1	1	3	4
F	2	3	1	1	1	X	1	4	3
G	1	3	1	1	1	1	X	3	3
H	3	2	3	3	3	4	2	X	1
I	4	3	2	1	4	3	3	1	X

3) *Bleed* the group by decreasing their food. Any tracks with no food remaining are discarded from the set.
4) *Breed* remaining tracks in the group by giving each a chance of creating a duplicate containing a minor mutation.

Seed tracks include mirror Hamming codes as well all possible tracks with 0, 1, or 2 units excluded. The possibility of adding all original tracks as well was considered, but discarded due to the likelihood that these tracks represent local maxima. Simulated annealing techniques were interjected to gradually increase bleeding and decrease breeding. Consequently, the number of tracks initially swells, followed by a slow decline.

The selected set is saved if its differentiation properties are better than any previously generated. If upon completion the algorithm has failed to achieve the minimum differentiation desired, the process is retried, relying on random elements in the algorithm to explore new possibilities.

Running the *Feed, Bleed, and Breed* algorithm on the tracks in Table X results in the core tracks in Table XII. Mapping the tracks from Table X into the core tracks results in the differentiation properties shown in Table XIII. Note that encoding the original track set requires five bits as there are 19 tracks, while the core set of

VII. EXPERIMENTAL RESULTS

The scheduling and binding implementation was tested on an AR filter, a discrete Fourier cosine transformation (DFCT), and an elliptic filter, under a variety of conditions. The hardware requirements for these benchmarks are presented in Table XIV.

Each benchmark was tested through a wide spectrum of performance/cost tradeoffs, comparing the functional unit requirements of standard implementations for fault identification through triple modular redundancy, and fault security through modular duplication without fault-identification methodology. As expected, our technique entails significantly less overhead than triple-modular redundancy (43% savings). In addition, our results indicate that despite the interjection of fault-identification properties, our techniques compare favorably to modular duplication (11% savings), despite the fact that modular duplication lacks fault-identification capabilities.

The differentiation properties of each implementation are shown in Table XV. The result of applying our regularization algorithm is also

TABLE XIV
EXPERIMENTAL RESULTS

DFG	clock cycle	number of gates in datapath		
		ours	duplication	triplication
AR	8	2242	2296	3552
	9	1940	2296	3552
	10	1382	1496	2352
	11	1268	1322	2064
DFCT	5	2230	2608	4128
	6	1878	2148	3384
	7	1544	1918	3012
	8	1616	1764	2808
EL	14	1274	1496	2352
	15	1074	1096	1752
	16	868	922	1464
	17	814	922	1464

TABLE XV
RESULTS FOR IRREGULAR AND REGULAR TRACK SETS

DFG	clock cycle	irregular		regular	
		diff.	bits	diff.	bits
AR	8	3	5	2	4
	9	2	5	1	3
	10	3	5	2	4
	11	4	5	2	4
DFCT	5	1	5	1	4
	6	1	5	1	4
	7	3	5	1	4
	8	1	5	1	5
EL	14	3	5	2	4
	15	5	6	1	3
	16	3	6	2	3
	17	5	6	2	3

shown. While the small benchmark sizes reduce the effectiveness of regularization, it can be seen that regularization can effect a decrease in control at the cost of losing some differentiation capacity. Thus, for applications where reduced error latency supplied by high differentiation values is crucial, irregular tracks remain the methodology of choice. For applications where control size is more relevant, however, our results show that even in small applications a differentiation of one can be maintained during regularization, thereby facilitating smaller control logic while maintaining viable differentiation.

VIII. CONCLUSIONS

We have described a comprehensive approach to fault identification, which provides full fault security under a broad fault model. Fault identification is achieved without triplication through the interjection of parity-like error correction equations for functional units during high-level synthesis. Furthermore, load-balancing techniques are utilized to reduce functional unit and voting hardware requirements. In addition, regularization algorithms are presented to reduce control logic area. The product is the first automated, chip-level synthesis technique to achieve fault-secure fault identification with only modest increases in area.

REFERENCES

[1] A. Orailoğlu and R. Karri, "Automatic synthesis of self-recovering VLSI systems," *IEEE Trans. Comput.*, vol. 45, pp. 131–142, Feb. 1996.

[2] W. Chan and A. Orailoğlu, "High-level synthesis of gracefully degradable ASICs," in *Proc. Eur. Design and Test Conf.*, Mar. 1996, pp. 50–54.

[3] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Rockville, MD: Computer Science, 1990.

[4] P. Vera, *Introduction to the Theory of Error-Correction Codes*. New York: Wiley, 1989.

[5] R. Jain, A. Parker, and N. Park, "Module selection for pipelined synthesis," in *Proc. Design Automation Conf.*, June 1988, pp. 542–547.

Evolutionary Algorithms for the Synthesis of Embedded Software

Eckart Zitzler, Jürgen Teich, and Shuvra S. Bhattacharyya

***Abstract*—This paper addresses the problem of trading off between the minimization of program and data memory requirements of single-processor implementations of dataflow programs. Based on the formal model of synchronous dataflow (SDF) graphs [1], so-called single appearance schedules are known to be program-memory optimal. Among these schedules, buffer memory schedules are investigated and explored based on a two-step approach: 1) An evolutionary algorithm (EA) is applied to efficiently explore the (in general) exponential search space of actor firing orders; 2) For each order, the buffer costs are evaluated by applying a dynamic programming post-optimization step (GDPPO). This iterative approach is compared to existing heuristics for buffer memory optimization.**

***Index Terms*—Dataflow, evolutionary algorithms, memory management, software synthesis.**

I. INTRODUCTION

Software synthesis has become an important component of the implementation process for embedded VLSI systems due to flexibility and time-to-market considerations.

Synchronous dataflow (SDF) [1] is a restricted form of dataflow in which the nodes, called *actors*, have a simple firing rule: The number of data values (*tokens, samples*) produced and consumed by each actor is fixed and known at compile time. The SDF model is used in industrial DSP design tools, e.g., SPW by Cadence, COSSAP by Synopsys, as well as in research-oriented environments, e.g., Ptolemy [2], GRAPE [3], and COSSAP [4]. Those systems include code generation tools with code (usually optimized assembly code) stored for each actor in a target-specific library. Typically, code is generated from a given schedule by instantiating actor code in the final program by code inlining. Subroutine calls may have unacceptable overhead, especially if there are many small tasks.

With this model, it is evident that the size of the required program memory strongly depends on the number of times an actor appears in a schedule, and so-called *single appearance schedules*, where each actor

Manuscript received November 27, 1998; revised July 26, 1999. The work of E. Zitzler was supported by the Swiss National Science Foundation. The work of S. S. Bhattacharyya was supported by the U.S. National Science Foundation under CAREER Grant MIP9734275.

E. Zitzler is with the Computer Engineering and Networks Laboratory (TIK), ETH Zürich, Zürich, Switzerland.

J. Teich is with the Computer Engineering Laboratory (DATE), University of Paderborn, Paderborn, Germany.

S. S. Bhattacharyya is with the Department of Electrical and Computer Engineering, and the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 USA (e-mail: ssb@eng.umd.edu).

Publisher Item Identifier S 1063-8210(00)01026-X.