Self Recovering Controller and Datapath Codesign

Samuel N. Hamilton
Computer Science and Eng.
Univ. of California, San Diego
La Jolla, CA 92093, USA

Andre Hertwig
Fachbereich Elektrotechnik
Univ. GH Siegen
57068 Siegen, Germany

Alex Orailoğlu
Computer Science and Eng.
Univ. of California, San Diego
La Jolla, CA 92093, USA

Abstract

As society has become more reliant on electronics, the need for fault tolerant ICs has increased. This has resulted in significant research into both fault tolerant controller design, and mechanisms for datapath fault tolerance insertion. By treating these two issues separately, previous work has failed to address compatibility issues, as well as efficient codesign methodologies. In this paper, we present a unified approach to detecting control and datapath faults through the datapath, along with a method for fault identification and reconfiguration. By detecting control faults in the datapath, we avoid the area and performance overhead of detecting control faults through duplication or error checking codes. The result is a complete design methodology for self recovering architectures capable of far more efficient solutions than previous approaches.

1 Introduction

The increasing need for fault tolerance has motivated the development of a variety of fault tolerance techniques. For datapath faults, techniques such as triplication with voting [1], checkpointing with rollback [2] [3], and concurrent error recovery [4] have all been explored. Recovery from permanent faults has been addressed through sparing [5], and graceful degradation [6]. For control faults, the literature concentrates on detection through duplication or error codes, followed by rollback and recovery [7] [8].

While the ideas presented regarding fault tolerant datapath and controller design are useful, the disjoint nature of this research limits the practicality of some of the techniques due to compatibility issues. For example, datapath rollback techniques potentially introduce a significant number of state transitions, which can seriously impact the effectiveness of methodologies for control design that rely on sparse control flow graphs.

Even more relevant is the effect that separate controller and datapath design can have on overall design efficiency. By treating each separately, potential combinations of fault detection and identification circuitry are ignored.

Our approach addresses these issues by combining the error detection and fault isolation requirements of the controller and the datapath into a single system. For datapath error detection, we rely on duplication on disjoint hardware, followed by comparison, as in traditional rollback schemes for transient errors. We extend this as suggested in [9] by encoding error identification properties during high-level synthesis capable of pinpointing datapath faults.

The real novelty in our system is that we are actually able to identify control faults in the datapath. In a traditional controller, fault-secure fault detection through datapath duplication is not possible, let alone fault identification. This is because a single control fault may lead to multiple erroneous signals to the datapath. Thus, when the duplicated calculations are compared, their mutual corruption could be masked. To avoid this, we introduce a partitioning scheme which splits the controller into several sub-machines. Duplicated calculations are then issued by disjoint sub-machines, thereby ensuring fault-security assuming a single fault.

By detecting faults through the datapath, we avoid costly error detection hardware within the controller. Not only does this avoid the significant overhead of controller duplication or error encoding hardware, it also eliminates the necessity of checking circuitry in the controller. As checking hardware usually contributes heavily to the critical path of the controller, and the controller critical path is often crucial in designs with strict timing requirements, this is a significant benefit.

An additional novelty is our proposed method of post-identification reconfiguration. Our technique is based on the observation that since every calculation is duplicated on disjoint hardware, a single fault is either in the duplicate or the original calculation, not both. Thus, upon identification of a fault, any calculation including that fault can be disabled without effecting performance. The circuitry required to support this methodology is minimal compared to sparing, and avoids the performance loss of graceful degradation. Instead, degradation is in terms of error-detection capacity. Of course, sparing and graceful degradation are fully compatible with our approach, though their application to the controller is by no means trivial.

The combined approach to fault detection, isolation, and reconfiguration, represents an extremely efficient solution to fault tolerant controller/datapath codesign.

2 Overall Approach

The standard approach to fault isolation is to perform each operation three times, compare these results, and if one disagrees label the associated unit faulty [1]. While triplication is fault secure assuming at most one fault, it entails massive area and power overhead in addition to lengthening the critical path with voting hardware.

Due to this expense, rollback approaches to recovery from transient faults have forgone triplication in favor of duplication [2] [3]. The reason permanent faults were not addressed is that while duplication comparison can detect faults within a single fault assumption, it cannot determine which calculation contains the faulty unit.

Our approach to fault isolation is based on the observation that while calculation duplication does not immediately identify the fault, it does supply crucial information: hardware issuing or used in a calculation/duplicate pair reporting an error is faulty.

Assuming a single fault, it is known the faulty hardware is shared by all calculation/duplicate pairs reporting an error. To pinpoint the fault, we therefore intersect the hardware of all calculation/duplicate pairs reporting an error. When the set of potentially faulty hardware is reduced to a cardinality of one, the fault has been identified.

To prevent a single fault from corrupting both a calculation and its duplicate, potentially masking the fault, it is important that the calculation and its duplicate utilize disjoint hardware. This is ensured in the datapath through scheduling and binding restrictions. Traditional controllers foil this scheme, however, as a single fault in the controller could potentially effect both a calculation and its duplicate. To avoid this effect, we have devised a partitioning scheme which splits the controller into distinct submachines. This decomposition allows calculations and their duplicates to be issued from disjoint submachines, thereby maintaining fault security. Figure 1 shows the basic system organization.

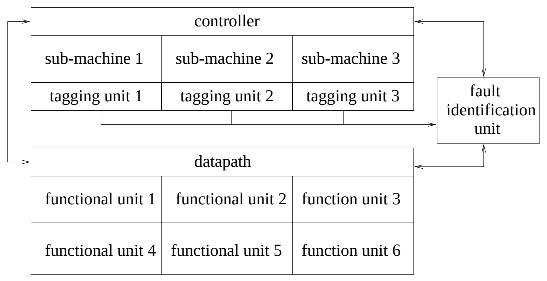

Figure 1: System Overview.

To support this methodology, comparison operations that check for errors must be labeled with the hardware set utilized by the calculations they are comparing. This is achieved through *tag* generation, done by disjoint tagging units within the controller. Tags containing the set of potentially faulty hardware are sent to the fault identification logic to be used for fault isolation in the event of an error.

Following fault identification, rollback restores a safe state. Reconfiguration is triggered using a simple, yet effective technique. As each calculation is performed twice on disjoint hardware, calculations using faulty hardware can be suspended without interfering with functionality. This is implemented by allowing instructions to be issued from each sub-machine only when the fault identification unit indicates the hardware utilized is fault free, as shown in figure 2. This technique avoids the performance loss of graceful degradation without the substantial hardware cost of sparing.

3 Terminology and Assumptions

Several definitions are given here to simplify future references: a *checkpoint* is a set of timesteps during which a calculation and its duplicate are performed and compared. Operations contained by a calculation within a single checkpoint are referred to as a *string*, while a string and its duplicate are referred to as a *track*. The set of potentially faulty hardware, as determined by the intersection of tracks reporting er-

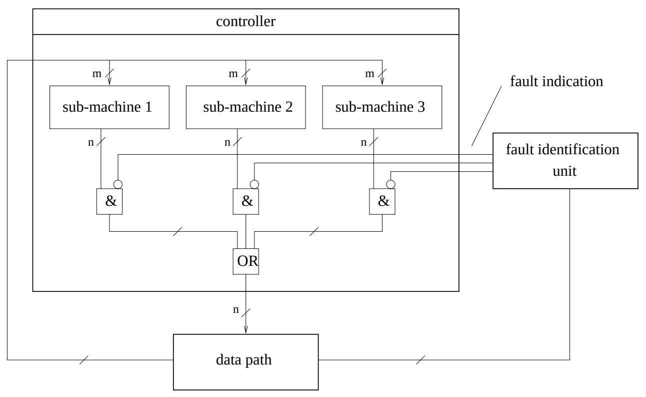

Figure 2: Controller Structure

rors, is the *ambiguity set.* A singleton ambiguity set denotes the faulty unit.

In this paper, we also adopt the following assumptions:

1. During fault isolation, we assume at most one discrete hardware segment is faulty.
2. We adopt a Byzantine fault model, wherein faults are not assumed to produce consistently erroneous behavior. A faulty segment producing a correct value is a *false negative.*

4 A Small Example

The following example illustrates the fundamental technique proposed. Table 1 shows a simple FSM state transition graph. Each output bit corresponds to the activation of one of three functional units in the datapath, labeled A, B, and C. Table 2 shows the accompanying datapath track information. Note that the original string and duplicate string use disjoint hardware. What remains to be done is partitioning of the FSM issuing logic into sub-machines such that each string is issued by disjoint logic.

Note that table 2 completely defines the datapath fault identification characteristics. For example, if tracks 1, 2, 4, and 5 all report errors, the intersection of the unit sets for these tracks identifies the faulty unit {A, B} $\bigcap$ {A, C} $\bigcap$ {A, B} $\bigcap$ {A, C} = {A}. Due to false negatives, however, a fault in A does not always ensure that all tracks including A will report faults. By building redundancy into the track set, we can alleviate this threat. Thus, if A is faulty, and tracks 1 and 5 exhibit false negatives, identification is still possible due to tracks 2 and 4 {A, C} $\bigcap$ {A, B} = {A}.

It is always possible that numerous false negatives will prevent fault identification. For example, if A is

input bits	from state	to state	output bits A	B	C
1	s0	s0	0	0	0
0	s0	s1	1	1	1
0	s1	s2	1	1	1
0	s2	s3	1	1	1
0	s3	s4	1	1	1
0	s4	s5	0	1	1
0	s5	s6	1	1	1
0	s6	s7	1	1	1
0	s7	s8	1	1	1
0	s8	s9	1	1	1
0	s9	s10	0	1	1
0	s10	s10	0	0	0

Table 1: State Transition Graph.

Track	Original String	Duplicate String
1	A(s0,s1)	B(s3,s4)
2	A(s2,s3)	C(s0,s1)
3	B(s0,s1,s2)	C(s2,s3,s4)
4	A(s5,s6)	B(s8,s9)
5	A(s7,s8)	C(s5,s6)
6	B(s5,s6,s7)	C(s7,s8,s9)

Table 2: String Binding and Active States.

faulty, and tracks 2 and 5 exhibit false negatives, a fault in B cannot be ruled out {A, B} $\bigcap$ {A, B} = {A, B}. Of course, assuming random input the probability of this approaches 0 as the number of tracks processed approaches infinity. To maximize resistance to false negatives, we generate datapath tracks such as those in table 2 using the *differentiation* heuristic suggested in [9]. In essence, this technique maximizes resistance to false negatives by maximizing the number of tracks that include a and exclude b, for any two distinct hardware segments a and b.

The differentiation heuristic is also used for control logic partitioning. By utilizing this heuristic instead of using a constructive approach, we ensure compatibility with advanced partitioning methodologies for control logic synthesis [10] [11]. A central idea in these works is to break the controller into smaller portions before utilizing automated synthesis techniques. As automated synthesis can find much more effective solutions on smaller problems, the overall solution can be dramatically improved. The partitioning method chosen effects how evenly area and speed is distributed between sub-machines, as well as the amount of optimization possible. By using a hybrid heuristic weighting area, speed, and differentiation, it is possible to tailor controller partitioning to specific application requirements.

Sub-Machine 1	Sub-Machine 2	Sub-Machine 3
string 1		string 1 dup
string 2	string 2 dup	
	string 3	string 3 dup
	string 4	string 4 dup
string 5		string 5 dup
string 6	string 6 dup	

Table 3: String Partitioning into Three Sub-Machines.

Table 3 shows sub-machine partitioning using the differentiation heuristic. New state transition graphs can now be generated for each sub-machine. Since output signals are ORed lines from sub-machines (see figure 2) sub-machines not responsible for issuing a string may encode the relevant output bits as *don't cares*, resulting in substantial savings during logic synthesis. Using this methodology, the state transitions in table 4 can be generated. This description can be fed into a standard synthesis tool to generate the final control logic.

input bits	from state	to state	M1 output	M2 output	M3 output
1	s0	s0	000	000	000
0	s0	s1	1-1	-1-	—
0	s1	s2	1-1	-1-	—
0	s2	s3	—	11-	–1
0	s3	s4	—	1–	-11
0	s4	s5	0–	0–	011
0	s5	s6	-11	1–	—
0	s6	s7	-11	1–	—
0	s7	s8	-1-	–1	1–
0	s8	s9	—	–1	11-
0	s9	s10	0–	0-1	01-
0	s10	s10	000	000	000

Table 4: Sub-Machine 1-3 State Transition Graphs.

5 Controller Faults

Figure 2 shows the target control structure. Control logic is partitioned into several sub-machines, whose output determines datapath activity. If a fault has previously been identified, that information is supplied by the fault identification unit, which prevents strings utilizing the faulty hardware from being issued.

The following criteria are required to ensure the controller is fault secure:

1. A string and its duplicate are issued by disjoint sets of sub-machines.

2. Decomposition forms at least three distinct sub-machines.

3. Sub-machines do not communicate.

4. Input/Output lines do not have one-to-many relationships with datapath components.

The first requirement prevents a single sub-machine fault from corrupting a string and its duplicate. A more subtle issue is that the controller must be partitioned into at least three sub-machines. Two is not sufficient, because each track would be required to include both sub-machines to maintain disjointness between strings. Thus, regardless of which tracks report errors, a fault in the first sub-machine is not distinguishable from a fault in the second.

It is also important to ensure sub-machines do not communicate. If communication did occur, fault isolation could not determine if a specific sub-machine was faulty, or if the fault was in a sub-machine it received data from. This can lead to replication of some functionality that otherwise might have been shared. For most applications, this replication has little effect on area, as the simplification of the synthesis problem more than compensates for this replication, since smaller sub-machine definitions allow synthesis tools to perform significantly more optimization. This is particularly true when the majority of the controller is dedicated to operation issuing, as in data-dominated applications.

An additional restriction is that a single control output line must not lead to multiple datapath components. The importance of this becomes clear when analyzing the effect of interconnect faults. If an output line is faulty, and controls multiple datapath components, then that fault can propagate through multiple resources potentially resulting in error masking. If each output line controls at most one component, however, a fault in an output line is equivalent to a fault in the receiving component. Thus, fault security and identification capacity is maintained. Similarly, faults in the AND gates shown in figure 2, and other propagation logic associated with output signals do not jeopardize fault security. Since the logic associated with propagation of each output signal does not interact with other output signals, a fault in this logic can be considered equivalent to a fault in the output line. Thus, the controller is fault secure.

6 Fault Identification

The hardware associated with fault identification consists of a set of comparators, a set of flip-flops to maintain the ambiguity set, and intersection logic for ambiguity set reduction.

In previous literature, comparators, or voting units, have often been assumed to be fault free. Our approach does not rely on fault free comparator units. Comparators are included in tracks as any other unit is, with the restriction that the comparator cannot be a member of either string.

Theorem 1 *Let c be a comparator which checks a track t utilizing a disjoint set of hardware from t. c reports an error if and only if faulty behavior exists.*

Proof: Since track t and comparator c utilize disjoint hardware, by the single fault assumption a fault cannot exist in both the hardware utilized by t and c. Thus, there are three possible conditions:

1. t displays faulty behavior, c does not.
2. c displays faulty behavior, t does not.
3. t and c display correct behavior.

Under condition 1, an error is reported, since c correctly detects the faulty behavior of track t. Under condition 2, an error is reported, since c can only display faulty behavior when a track is correct by reporting it is incorrect. Under condition 3, no error is reported, since c will correctly report that t is correct. Therefore, under all possible conditions, if a track or comparator displays faulty behavior, a fault is reported. ■

Faults associated with tag generation or ambiguity set storage are also safe under the single fault assumption. If this information is incorrectly generated or stored due to a fault, then the datapath and controller components responsible for string calculation are fault free, and the fault is never triggered.

7 Recovery Mechanism Faults

Faults in the recovery mechanism are different from faults in identification hardware, as there is the potential to rearrange multiple calculations. The recovery mechanism we propose, however, does not rebind or reschedule operations as in many purely datapath solutions. Our technique simply disables strings and comparisons which utilize faulty hardware. Therefore, in order to maintain functionality despite a fault in the recovery mechanism, we simply need to ensure that if a string is disabled, its duplicate is not.

The ambiguity set is stored using a flip-flop for each hardware segment. The disable line is on for segment a if segment a's flip-flop is on, and no other flip-flop is on. In this organization, a single fault can only turn on the disable line of a single segment. Since strings are disjoint, if a string is disabled, its duplicate is not.

8 Implementation and Results

We implemented scheduling and binding algorithms for the datapath, maximizing differentiation through a greedy algorithm. The algorithm acts to maintain disjointness and maximize resistance to false negatives for registers, functional units, and comparators. As comparator scheduling implicitly creates tracks, track information and its effect on differentiation was updated following each comparison scheduled. If increasing the number of comparators would decrease the critical path, the number of comparators was iterated, and comparator scheduling redone. This differs from the technique specified in [9], where the number of tracks were minimized with the assumption that this would minimize the number of comparators. As comparators are explicitly included in our model, we attempted full utilization of available comparators to improve fault identification.

The state transition graph generated is sent to our partitioning algorithm, which partitioned tracks into three sub-machines through greedy assignment, prioritizing differentiation. Following partitioning, *don't cares* were inserted according to the criteria outlined in section 4.

As standard high-level synthesis benchmarks are quite small, and devoid of interesting control characteristics, we incorporated several benchmarks into a single architecture. The resulting architecuter could repeatedly iterate on a FIR(finite-length impulse response) filter, an Elliptic fileter, a Discrete Cosine Transformation, and an AR(autoregressive) filter. It could also loop through each in turn.

As expected, the datapath had significant area savings over triplication (33%), comparable to results from previous implementations utilizing this fault identification technique [4] [9]. We synthesized the controller portion using both SIS and Synopsys. Figure 3 shows the result of compilation using our partitioning method with Synopsys, while figure 4 shows the result of triplication.

The area savings over triplication were 47% for both

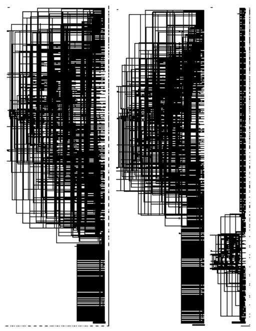

Figure 3: Control Logic Synthesis Using Our Methodology.

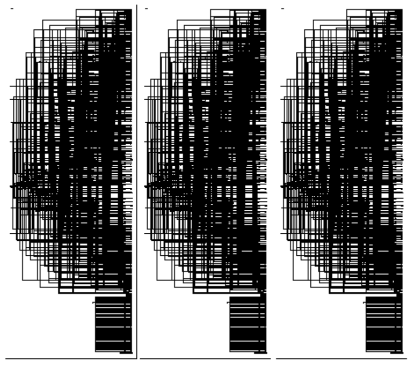

Figure 4: Control Logic Synthesis Using Triple Modular Redundancy.

tools (3.034 mm^2 vs. 1.596 mm^2 in SIS), while the speedup was 18% (24.68 ns for triplication vs. 20.27 ns for our method.)

It must be noted that since area was not a partitioning criterion the area distributions between submachines varied drastically, as can be seen in figure 3. This resulted in uneven propagation delay through the three sub-machines (20.27 ns, 18.71 ns, 8.81 ns.) As the fastest machine must wait for the slowest machine, this uneven distribution results in a slower overall machine than a more even partitioning would have. While outside the scope of this paper, it is important to note that this effect can be avoided through partitioning heuristics that prioritize area distribution and timing constraints [10] [11]. As the controller often contributes to the critical path, this adjustment can have very beneficial results.

9 Conclusion

We have presented the first approach to self-recovering ASIC design to combine datapath and controller issues into a single coherent synthesis system. We do so by identifying both datapath and control faults in the datapath, thereby eliminating error checking hardware from the controller. The result is an efficient methodology which severely reduces controller area requirements, while maintaining compatibility with state of the art datapath and controller design methodologies.

References

[1] B. Iyer and R. Karri, "Introspection: a low overhead binding technique during self-diagnosing microarchitecture synthesis," in *Proceedings of Design Automation Conference*, June 1996, pp. 137–142.

[2] D.M. Blough, F.J. Kurdahi, and S.Y. Ohm, "Optimal algorithms for recovery point insertion in recoverable microarchitectures," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol. 16, no. 9, pp. 945–955, September 1997.

[3] A. Orailoğlu and R. Karri, "Automatic synthesis of self-recovering VLSI systems," *IEEE Transactions on Computers*, vol. 45, no. 2, pp. 131–142, February 1996.

[4] S.N. Hamilton and A. Orailoğlu, "Concurrent error recovery with near-zero latency in synthesized ASICs," in *Design, Automation and Test in Europe*, 1998, pp. 604–609.

[5] K. Kim, R. Karri, and M. Potkonjak, "Configurable spare processors: a new approach to system level fault tolerance," in *Proceedings of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, November 1996, pp. 295–303.

[6] W. Chan and A. Orailoğlu, "High-level synthesis of gracefully degradable ASICs," in *Proceedings of European Design and Test Conference*, March 1996, pp. 50–54.

[7] Y. Tamir and M. Tremblay, "High-performance fault-tolerant VLSI systems using micro rollback," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 548–554, April 1990.

[8] Andre Hertwig, Sybille Hellebrand, and Hans-Joachim Wunderlich, "Fast self-recovering controllers," in *Proceedings of the 16th IEEE VLSI Test Symposium*, April 1998.

[9] S.N. Hamilton and A. Orailoğlu, "Microarchitectural synthesis of ICs with embedded concurrent fault isolation," in *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, June 1997, pp. 329–338.

[10] P. Ashar, S. Devadas, and A. R. Newton, "Optimum and heuristic algorithms for an approach to finite state machine decomposition," *IEEE Transactions on CAD*, vol. 10, no. 3, pp. 296–310, March 1991.

[11] Andre Hertwig and Hans-Joachim Wunderlich, "Fast controllers for data dominated applications," in *Proceedings of the European Design and Test Conference*, March 1997, pp. 84–89.