# Concurrent Error Recovery with Near-Zero Latency in Synthesized ASICs*

Samuel Norman Hamilton and Alex Orailoğlu

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

**Abstract**

*The importance of fault tolerant design has been steadily increasing as reliance on error free electronics continues to rise in critical military, medical, and automated transportation applications. While rollback and checkpointing techniques facilitate area efficient fault tolerant designs, they are inapplicable to a large class of time-critical applications. We have developed a novel synthesis methodology that avoids rollback, and provides both zero reduction in throughput and near-zero error latency. In addition, our design techniques reduce power requirements associated with traditional approaches to fault tolerance.*

## 1 Introduction

As chip density continues to rise, so does its vulnerability to faults. In critical military, medical, and transportation applications an undetected fault can have costly or even life threatening results.

While triple-modular redundancy combined with sparing is an effective method for concurrent fault isolation and recovery, its tremendous overhead in both area and power has prompted a search for more efficient solutions.

High-level synthesis approaches, which ameliorate the cost of calculation duplication with techniques such as rollback and load balancing, have proven particularly effective [1] [2] [3] [4]. These techniques entail significant error latency, however, making them impractical in applications with strict timing requirements. Furthermore, in order to circumvent overhead, many approaches provide only partial error checking capacity, thereby limiting resilience [5] [6] [7].

In this paper, we present an automated synthesis methodology which provides zero reduction in throughput, near-zero error latency, and guarantees fault security. Rollback is replaced with a novel error recovery system, which upon detection of an error utilizes encoded error isolation information to identify which units are not involved. Based on a single faulty unit assumption, duplication is temporarily suspended on these safe units, freeing resources to conduct recalculation in parallel with subsequent operations. Furthermore, our system actually eliminates recalculation in many cases by utilizing error isolation information to deduce correct results. Thus, throughput is never compromised, while error latency is often zero, and is otherwise limited to the recalculation of a few operations. In addition, our high-level synthesis approach is designed such that error recovery introduces only minor control logic and interconnect overhead.

By avoiding rollback, bulky state storage hardware is avoided. In addition, our technique utilizes less recalculation than retry. Thus, compared to rollback schemes, power usage is reduced both during normal operation by avoiding state storage and during recovery by limiting recalculation. Compared to triple-modular redundancy the reduction in power consumption is even greater, cutting the number of operations by almost a third.

The remainder of the paper is organized as follows: Section 2 outlines the proposed scheme. Section 3 discusses synthesis issues, while section 4 describes our high-level synthesis implementation. Section 5 shows our results on a variety of benchmarks. Section 6 summarizes our work and results.

## 2 Overall Approach

The algorithmic approach we propose maintains fault security while avoiding throughput reduction and error latency usually associated with fault detection. This is achieved through a modest interjection of hardware combined with carefully tailored high-level synthesis routines. Figure 1 outlines the operation of a system utilizing our technique.

*This work is supported by the National Science Foundation under grant number MIP-9308535.

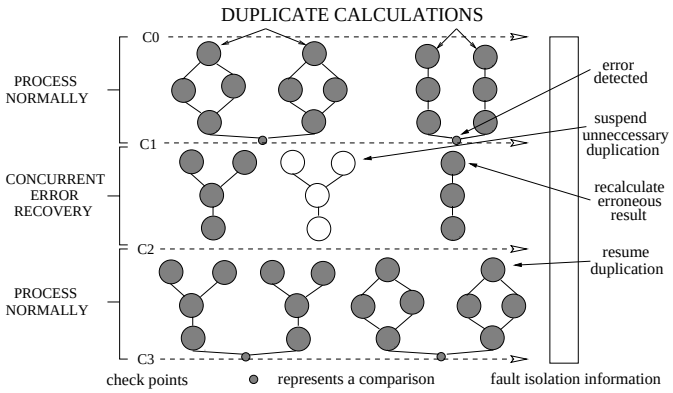


Figure 1: Fault Isolation and Recovery Structure.

Errors are detected through comparison of all calculations with duplicate calculations at periodic checkpoints. When an error is detected, calculation duplication is temporarily suspended. The resources freed are then used for recalculation of the erroneous operations while processing continues. Calculation duplication is immediately resumed following recalculation. Thus, as long as the single fault assumption is valid during recalculation an undetected error cannot occur, since the encoded error isolation information enables the system to avoid reliance on potentially faulty units during duplication suspension.

The encoded error isolation also allows a reduction of the set of potentially faulty units during the report and correction of errors through a single fault assumption. Upon isolation, spares are used to replace the faulty unit, since the strict timing constraints of time-critical applications disallow more efficient solutions such as graceful degradation. The single fault assumption is then lifted, supplying the system resiliency in the face of multiple, concurrent faults.

### 2.1 Algorithmic Duplication

Error detection is achieved through calculation duplication and result comparison. A calculation may involve several operations before comparison. This can reduce comparators as well as reduce registers required to store uncompared results. For subsequent clarity, we denote a string of operations as a *string*, while a string and duplicate pair is referred to as a *track*.

As pointed out in [4], the information inherent in error detection can be used for error isolation: when a track reports an error, it is known that one of the units active in that track is faulty. In fact, by the single fault assumption we can deduce that all units not included in the track are fault free. The set of units which might be responsible for a detected error is referred to as the *ambiguity set*; a singleton ambiguity set denotes the faulty unit.

Notice that the information present in any one track may not be adequate to narrow the set of potentially faulty units to one. One way to further reduce the ambiguity set is through exploiting information inherent in the error resolution process. After recalculation of the erroneous track, result comparison will reveal which string contains the error. The ambiguity set is then limited to the units in that string.

Since strings may contain multiple units, recalculation may not result in a singleton ambiguity set. In such cases, complete ambiguity resolution is achieved through repetition of the process. When the next error is detected and recalculated, the additional information can be combined with the current ambiguity set to facilitate further reduction of the ambiguity set.

### 2.2 Error Recovery

Time-critical applications impose three challenges to error recovery:

- error latency associated with recovery from an error or isolation of a faulty unit must be minimal.
- reconfiguration after isolation of a faulty unit cannot result in degraded performance.
- in applications where outside input is regular and frequent, the system must be able to process input at the same speed it is supplied. Thus, throughput cannot be compromised.

Post-reconfiguration performance can be preserved through simple sparing. Minimizing latency and maintaining processing speed, however, is extremely challenging without resorting to functional unit triplication. To achieve this goal, after an error is detected, operations must continue to be processed while the fault is simultaneously isolated and the error corrected. This approach limits error latency to the recalculation time, while maintaining throughput.

By freeing resources through the temporary suspension of calculation duplication, this imposing task can be achieved. The single fault assumption assures only units in the ambiguity set can be faulty. Thus, calculations done in parallel with recalculation do not require duplication as long as they exclude units in the ambiguity set. In addition, by reducing the number of operations through suspension of duplication, power consumption actually decreases during recalculation.

If numerous tracks require recalculation, the additional resources provided by duplication suspension could prove insufficient. To prevent this, tracks can be designed such that at most one track is recalculated after any given checkpoint. This is achieved by utilizing the error isolation information implicit in error detection, as seen in figure 2.

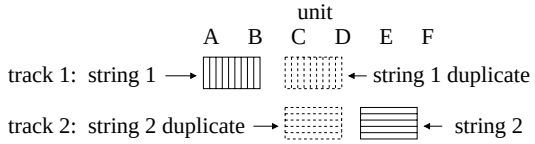


Figure 2: Track set where a maximum of one string requires recalculation.

If track 1 reports an error, the ambiguity set would be ABCD, and a string would have to be recalculated using units EF. Consider, however, if track 1 and 2 both report an error. Due to the single fault assumption, the ambiguity set would be ABCD & CDEF = CD. Since neither string 1 nor string 2 use these units, we can conclude that these strings have produced correct results. Thus, in this case, no recalculation is required. In fact, through careful track design, it can be ensured that multiple errors will always allow identification of a safe string in each erroneous track. Of course, if only one error is reported, no such deductions can be made, and recalculation must occur.

While limiting recalculation and suspending duplication provides resources for continuing calculation, data dependencies between an unresolved error undergoing recalculation and the calculations done in parallel must be considered. Of course, this is only problematic for *dependent tracks*, which rely on the result of the unresolved error. *Independent tracks* may be calculated in parallel with recalculation without fear of introducing errors.

To ensure throughput is maintained, dependent tracks must utilize *calculation splitting*, wherein two calculations are made, one for each possibly correct predecessor string. Immediately following recalculation, the correct string is identified and calculations based on the incorrect string halt.

Note that while calculation splitting introduces redundancy, the resources devoted to that calculation are no greater during error recovery than during normal operation. This is because for each dependent track the redundancy added by calculation splitting is equivalent to the redundancy eliminated by duplication suspension.

# 3 Synthesis

In order to minimize control logic and interconnect overhead, it is important to have as much similarity in behavior before, during, and after the detection of an error. In addition, the properties alluded to previously must be maintained in order to limit recalculation requirements. There are two major synthesis issues to address within this context:

1. Overall control flow both during normal operation and directly after the detection of an error.
2. High-level synthesis of track sets that enable error recovery and are compatible with the desired flow of control.

## 3.1 Control Logic

The flow of control should be as simple as possible in order to minimize control logic. To achieve this, there are several behavioral aspects to the design which must be succinctly captured by a single control flow. The main parts are:

- control logic for normal operation.
- control logic for recalculation.
- control logic for tracks processed in parallel with recalculation.
- control logic for ambiguity resolution.

Ambiguity resolution can be supplied by including a bit for each unit with comparison operations. Upon detection of an error, these bits are intersected with the ambiguity set. Resolution of an error results in intersection of the bits from the incorrect string with the ambiguity set. If the ambiguity set is reduced to one, identification has occurred, and the faulty unit is replaced by a spare.

Recalculation control logic can be achieved by inserting an additional string of operations for each track directly after that track's checkpoint. A control mechanism is consequently required to ensure these operations are only activated by the error condition associated with them. Our solution is to give each operation a set of *condition bits*, one for the error condition of each track in the previous cut. An operation's condition bits indicate whether the operation should be done under each error condition, and can be used to ensure recalculation strings are only enabled when the corresponding track reports an error. Of course, a condition bit for the no error condition also exists, which is on for all strings that are not recalculations, and off for all strings involved in recalculation. In this way, control flow for recalculation operations and normal operations is identical.

The methodology utilized for limiting the number of recalculations to one string per checkpoint imposes an additional hardware requirement. As alluded to previously, tracks are designed such that correct strings can be deduced from the ambiguity set. To ensure that subsequent operations utilize correct data, tracks that report errors but contain strings without units from the ambiguity set receive special treatment. The register storing the correct data is copied into the

register with the incorrect data, and the error is ignored. Thus, if multiple errors occur, recalculation is avoided while maintaining ambiguity resolution.

In addition to simplifying recalculation, condition bits also supply an efficient framework to process dependent and independent tracks in parallel with recalculation. For independent tracks, duplication is waived. Thus, one string in each independent track is not processed, and the comparison operation that detects errors is skipped. This is captured by setting the condition bit off on strings and comparisons that are not processed.

For dependent tracks, the scenario is almost as simple. Though duplication is waived, calculation duplication requires that both strings continue, each based on different assumptions regarding which string in the erroneous track is correct. The simplest approach is to set up dependent strings such that they draw data from different registers, corresponding to the string they draw data from. If there is no error, these registers will contain identical data, so no consistency problems arise. When an error is detected, by setting the condition bits of dependent strings on, we ensure both strings are calculated and draw data from the appropriate register. Thus, dependent tracks continue to process strings as they would during normal operation. The only limitation is that after recalculation, the string reliant on faulty data should be ignored. This can be achieved using the register transfer hardware already suggested to enforce consistency during recalculation minimization.

Overall, this control flow efficiently captures the complex behavior of our technique. The only significant overhead is the condition bits, and the operations introduced for recalculation. Since an operation and duplicate pair requires only one recalculation operation, the increase over simple duplication in number of operations is 50%. As mentioned previously, minor hardware support is also required to maintain the ambiguity set and transfer between registers. In addition, recalculation operations may increase interconnect, though the static nature of dependent and independent track binding limits this effect.

### 3.2 High-Level Synthesis

The efficient control logic and recovery methodology described above are reliant on tracks maintaining several specific properties. These properties must be introduced during high-level synthesis, and have the following characteristics:

1. tracks with the same checkpoint must be designed such that multiple errors allow for deduction of correct strings, thereby minimizing recalculation.
2. independent tracks must contain a string that does not utilize units from the ambiguity set.
3. after recalculation eliminates the incorrect string in a dependent track, the remaining string must not contain units in the ambiguity set.

To minimize recalculation, for any track $a$ with checkpoint $C$, and any other track $b$ with checkpoint $C$, at least one string in $a$ must not include units in $b$. This is referred to as the *independent string property*. For track groups that hold this property, if more than one track reports an error, each of the tracks will contain a string whose result can be trusted, thereby avoiding recalculation. Recalculation is consequently restricted to single track errors only.

Occasionally, simultaneous tracks must violate the independent string property. Violation occurs when tracks share an operation, since all tracks that share an operation share the units that operation and its duplicate are bound to. Tracks that share an operation in this way are referred to as *joined tracks*, and are treated as a single track with multiple output. Thus, these tracks share a condition bit.

Independent tracks can be successfully encoded by enforcing the independent string property across a checkpoint. This ensures that each independent track contains a safe string, which can then be used without fear of using a unit in the ambiguity set. Of course, holding this property across all tracks would be extraordinarily difficult. Fortunately, the independent string property only needs to hold during recalculation of an erroneous track. After recalculation, errors are detected through algorithmic duplication as specified previously, allowing utilization of units in the ambiguity set without risking undetected output errors.

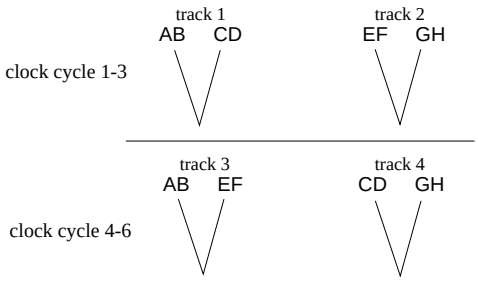


Figure 3: Independent tracks design.

Figure 3 illustrates this principle. If track 1 reports an error, the ambiguity set ABCD is created. Both tracks 3 and 4 contain strings which do not utilize units from this ambiguity set. These strings can be used during error recovery, while their duplicates are eliminated. Tracks 3 and 4 are similarly equipped if track 2 reports an error.

The calculation splitting methodology utilized for dependent tracks requires a less stringent property. Since only one string is kept, only that string must avoid using faulty units. This can be ensured by holding the *limited dependence property*, which ensures that if string $a$ is dependent on data from string $b$, then it cannot use units in the duplicate of $b$. Since the ambiguity set after recalculation is limited to those units used in the incorrect string, it is guaranteed that the string kept will not use any units from the ambiguity set. Note that during calculation, dependent strings can use units from the string they receive data from, even if those units are in the ambiguity set. This is justified by the fact that if one of those units is faulty, recalculation will identify the string containing the faulty unit and copy over its results.

## 4 Implementation

To test the feasibility of our techniques, we implemented scheduling and binding algorithms utilizing the encoding specifications previously presented. Our goal was to introduce fault tolerance while keeping the number of clock cycles constant.

First all operations are duplicated and scheduled. Ignoring binding considerations during scheduling ensures that the binding constraints introduced through enforcement of the independent string property and the limited dependence property do not result in an increase in the number of clock cycles.

After scheduling, a cut size is chosen, which acts as the maximum string size for all tracks. The maximum string size is imposed to introduce regularity, which significantly simplifies enforcement of the independent string property and the limited dependence property. To further simplify enforcement, recalculation occurs within one cut, thereby limiting the enforcement of the aforementioned properties as well as the error latency to one cut.

Binding is achieved through a variation of simulated annealing. The cost function is heavily weighted against shared units in strings and duplicates, and is lightly weighted against violations of the independent string and limited dependence properties. In addition, a small penalty is assessed if the cut following any track does not have sufficient free units for recalculation of that track. If at the end of annealing the cost is nonzero, the best schedule previously considered is randomly perturbed, and the process retried.

## 5 Results

The scheduling and binding implementation was tested on several benchmarks under a variety of conditions. Hardware requirements for homogeneous architectures using only ALUs are shown in table 1, and results for heterogeneous designs are shown in table 2. Each benchmark was tested in a wide spectrum of performance/cost tradeoffs.

As expected, our technique requires significantly less functional units than triple-modular redundancy (38% less for homogeneous designs, 39% less for heterogeneous designs). In addition, our results indicate that despite the variety of properties required to avoid error latency and state saving hardware associated with rollback, our techniques result in even less units than simple hardware duplication (7% less for homogeneous designs, and 8% less for heterogeneous designs).

Note our fault tolerance techniques require a minimum of three units, otherwise following error detection, no units would be available for recalculation. While this became relevant for three of our heterogeneous implementations of an elliptic filter, it is of little significance, since area is rarely critical in ICs requiring less than three units.

Overall, our results did not suggest a clear superiority of homogeneous or heterogeneous implementations of our techniques. Homogeneous implementations use less overall units, and are unlikely to be limited by the three unit minimum. On the other hand, heterogeneous implementations allow slight improvements in load balancing, and allow for greater flexibility in design. Of course, fault tolerance is only one of numerous criteria, and tradeoffs between homogeneous and heterogeneous design must be made within a broader context.

## 6 Conclusion

We have described the first efficient synthesis methodology for fault-secure fault identification with both zero reduction in throughput and near-zero error latency.

Not only do our methods satisfy the stringent requirements of time-critical applications, they do so with only modest increases in area. Our methodology limits control logic and interconnect overhead through regularized flow of control, and limits power requirements through reducing redundancy of calculation during error conditions.

## References

[1] 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.

| DFG | CC | no fault tolerance ALU units | fault tolerance ALU units | savings over duplication | savings over triplication |
|---|---|---|---|---|---|
| DFCT | 5 | 6 | 11 | 8% | 39% |
| | 6 | 5 | 9 | 10% | 40% |
| | 7 | 4 | 8 | 0% | 33% |
| | 8 | 4 | 7 | 13% | 42% |
| AR | 8 | 4 | 8 | 0% | 33% |
| | 9 | 4 | 7 | 13% | 42% |
| | 10 | 3 | 6 | 0% | 33% |
| | 11 | 3 | 6 | 0% | 33% |
| FIR | 5 | 8 | 16 | 0% | 33% |
| | 6 | 6 | 11 | 8% | 39% |
| | 7 | 4 | 8 | 0% | 33% |
| | 8 | 4 | 7 | 13% | 42% |
| EL | 13 | 4 | 7 | 13% | 42% |
| | 14 | 3 | 6 | 0% | 33% |
| | 15 | 3 | 5 | 17% | 44% |
| | 16 | 3 | 5 | 17% | 44% |

Table 1: Experimental results for homogeneous architectures.

| DFG | CC | no fault tolerance | | | fault tolerance | | | saving s over | savings over |
|---|---|---|---|---|---|---|---|---|---|
| | | add | sub | mult | add | sub | mult | duplication | triplication |
| DFCT | 5 | 3 | 4 | 3 | 6 | 6 | 6 | 20% | 47% |
| | 6 | 2 | 2 | 2 | 4 | 4 | 4 | 0% | 33% |
| | 7 | 2 | 2 | 2 | 4 | 3 | 4 | 7% | 39% |
| | 8 | 2 | 2 | 2 | 3 | 3 | 3 | 25% | 50% |
| AR | 8 | 2 | 0 | 4 | 4 | 0 | 8 | 0% | 33% |
| | 9 | 2 | 0 | 4 | 4 | 0 | 5 | 25% | 50% |
| | 10 | 2 | 0 | 2 | 3 | 0 | 4 | 13% | 42% |
| | 11 | 2 | 0 | 2 | 3 | 0 | 4 | 13% | 42% |
| FIR | 5 | 8 | 0 | 8 | 16 | 0 | 16 | 0% | 33% |
| | 6 | 4 | 0 | 4 | 8 | 0 | 8 | 0% | 33% |
| | 7 | 3 | 0 | 3 | 6 | 0 | 6 | 0% | 33% |
| | 8 | 2 | 0 | 2 | 4 | 0 | 4 | 0% | 33% |
| EL | 13 | 3 | 0 | 2 | 6 | 0 | 4 | 0% | 33% |
| | 14 | 3 | 0 | 1 | not enough units for efficient FT | | | | |
| | 15 | 2 | 0 | 1 | not enough units for efficient FT | | | | |
| | 16 | 2 | 0 | 1 | not enough units for efficient FT | | | | |

Table 2: Experimental results for heterogeneous architectures.


[2] W. Chan and A. Orailoğlu, "High-level synthesis of gracefully degradable ASICs," in *EDAT*, March 1996, pp. 50–54.

[3] S.Y. Ohm, D.M. Blough, and F.J. Kurdahi, "High-level synthesis of recoverable microarchitectures," in *EDAT*, March 1996, pp. 55–62.

[4] S.N. Hamilton and A. Orailoğlu, "Microarchitectural synthesis of ICs with embedded concurrent fault isolation," in *FTCS*, June 1997, pp. 329-338.

[5] A.T. Dahbura, K.K. Sabnami, and W.J. Hery, "Spare capacity as a means of fault detection and diagnosis in multiprocessor systems," *IEEE Transactions on Computers*, vol. 38, no. 6, pp. 881–891, June 1989.

[6] D. M. Blough and A. Nicolau, "Fault tolerance in super-scalar and VLIW processors," in *IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, July 1992, pp. 193–200.

[7] B. Iyer and R. Karri, "Introspection: a low overhead binding technique during self-diagnosing microarchitecture synthesis," in *DAC*, June 1996, pp. 137–142.